Tool Construction for Process-Centred Software Development Environments based on Object Databases

Schriftliche Arbeit
zur
Erlangung des Grades
„Doktor der Naturwissenschaften"
im Fachbereich Mathematik/Informatik
der
Universität-Gesamthochschule Paderborn

vorgelegt von

Wolfgang Emmerich
aus
Südlohn/Westfalen

Abstract

The aim of this thesis is to discuss the construction of tools for process-centred software development environments (PSDEs). Our main contribution is the proof that object database systems are a very suitable basis for improving the functionality of software development tools and for integrating them in PSDEs. We set out to prove this hypothesis following engineering principles rather than in an analytic or empiric way. We, therefore, first discuss the functionality that software developers require from tools contained in a PSDE. Starting from these requirements, we take the position of a tool builder and delineate requirements for a database system for document management purposes. We then review how well existing database systems satisfy these requirements. This results in the selection of object database systems as the most promising systems to take. We then propose a tool architecture that is based on object databases. We classify the components of this architecture into components that are common to any tool and thus can be reused and components that vary from tool to tool. We shall see that the most important varying component is the schema of the database. We propose tool specification languages that are capable of describing the tools' schemas as well as the other varying tool components at different levels of abstraction. These different levels of abstraction will provide the tool builder with the flexibility to define arbitrary syntax-directed tools. Then we discuss the construction of tools for the various languages identified above and their integration in GENESIS, an integrated environment for tool specification. After that, we outline the design and the implementation of compilers for our languages that generate executable tools. Finally, we evaluate the approach suggested in this thesis on the basis of two scenarios. These are tools for the GENESIS environment and tools for C++ class library development and maintenance, which we developed for the reuse departement of British Airways.

Acknowledgements

First of all I would like to thank my advisor Prof. Wilhelm Schäfer. He has guided my research, taught me to teach and improved my presentation and writing skills. Thanks a lot Wilhelm! Moreover, I would like to thank my co-supervisors Prof. Carlo Ghezzi and Prof. Uwe Kastens for their valuable feedback on earlier drafts of this thesis.

The research that made its way into this thesis was inspired by fruitful discussions with the members of the Merlin project. I would, therefore, like to thank the former and current knights of the round table Jörg Brunsmann, Jens Janke, Gerald Junkermann, Prof. Petr Kroha, Christian Luecking, Olaf Neumann, Burkhard Peuschel, Sabine Sachweh, Dr. Stefan Wolf and Dr. A. Zündorf as well as our guests, namely David Jacobs, Kelvin Ross and Prof. Jim Welsh.

I was particularly stimulated by supervising a number of master students. In his thesis Frank Buddrus showed the major drawbacks of current tool generators and suggested using an object database system as a basis for improving tool construction. Martin Kampmann implemented the initial prototype of the Groupie system and demonstrated that cooperative software development can be improved significantly by exploiting conventional database transactions. Dirk Schnorpfeil explored the limitations of relational database systems. He suggested denormalisation techniques that can be used for the efficient management of documents in terms of abstract syntax graphs in a relational database. He also demonstrated the very high costs that environment builders have to pay in terms of database schemas which are no longer comprehensible. Jörg Brunsmann showed that version and configuration management of documents can be built on top of the version manager of an object database system, thus proving that our requirements on databases are necessary in that respect and that they are sufficiently met by the database that we used in our implementation. Wiebke Reimer implemented efficient algorithms to version indexes on collections of objects and to index versioned objects within the kernel of the O_2 object database system. By supervising her work, I obtained valuable insights into the kernel of the database system that afterwards helped me to exploit it better. Jens Jahnke suggested semantic rules as they are presented in this thesis and implemented the incremental evaluation algorithm. Werner Beckmann demonstrated how import/export relationships between object database systems as well as object-oriented views, can be used for tool integration purposes. In particular he proved our requirements regarding views are necessary and that they are sufficiently addressed. Boris Gesell explored the fact that the tool specification languages we have suggested in this thesis for textual languages can, in a straight forward manner, be extended to graphical languages and that the tool architecture only has to be changed in a very limited way. Finally, Mike Wagener is currently extending the tool specification languages to support the definition of tool specific services that will then be used by the process engine for tool integration at a fine granularity. I thank you all for your contribution. I have enjoyed working with you!

The first prototypes for the compilers for our languages were built within a student's project by a group of graduate students. I, therefore, thank Werner Beckmann, Christian Bittner, Jörg Brunsmann, Heiko Köster, Cordula Köster, Wiebke Reimer, Hans-Dieter Schneider, Rolf Schneider, Thorsten Wystrychowski and Mike Wagener for their support.

The proof-reading done by Heather Hambloc, Wiebke Reimer and Andreas Baumkötter significantly contributed to getting the presentation into shape. Through Heather's work on this thesis I learned how bad my English still is. Without her help I could not have submitted my thesis in English!

Last, but not least, I would like to thank my parents Helmuth and Sigrid Emmerich and my girl friend Birgit Osterholt, whose encouragement and love helped me to overcome tiredness, lack of power and loss of self-confidence that are inherent in any PhD project.

Contents

Chapter 1

Introduction

A *software development environment* (SDE) includes tools that support most of the software life-cycle phases, that is the construction and analysis of the corresponding documents and document interdependencies [SW89]. A sophisticated integrated environment should allow the incremental, intertwined and syntax-directed development and maintenance of these documents (c.f., for example, [HN86, Lew88, ELN$^+$92]). The tools check for consistency between documents of the same and different types, visualise inconsistencies or even automatically preserve consistency during changes. To reflect the history of a software system, sophisticated environments also manage different revisions and variants of documents and support construction of different configurations (c.f., for example, [Fei91, Wes91, Dar92]).

A *process-centred software development environment* (PSDE) is an SDE that also has a tool called a *process engine*. The process engine maintains knowledge about the software process, the particular state a development project currently has and sometimes even the evolution of development states over time. In doing so, it can guide developers through tasks they are obliged to perform, automate particular tasks and most importantly control the way multiple users cooperate. Examples for PSDEs are Merlin [PS92, PSW92], Melmac [DG90], SPADE [BFG93a] and Marvel [BK90].

Current PSDEs do not support users as much as they could. This is mostly due to tools that have not been built for use within a process-centred environment. Firstly, tools do not allow multiple users to work concurrently on the same set of documents. Hence PSDEs have to work around this either by applying strict exclusive locking policies to documents to disable concurrent access completely or by enabling concurrent access to different versions of a document. This defers the problem to merging different document versions, which takes significant human effort. Secondly, PSDEs seldom include those tools that are tailored and integrated towards a particular process' or even project's needs, but only those tools that happen to be available. Finally, hardly any tool provides well-defined services the process engine could use for process automation.

In this thesis, we develop solutions to the above problems. To do so, we follow an engineering rather than an analytical or empirical approach. In an engineering tradition, we reuse already existing solutions. We exploit object database systems for document management purposes and message-based inter-process communication for integrating tools and process engines. We then develop in a requirements-driven manner architectures and languages for the specific problem of tool construction. As we shall see, this specificness will allow us to make particular assumptions under that problems become solvable that can hardly be solved in general. We follow a further engineering principle, namely to evaluate the quality of the suggested solutions with case studies. We, therefore, specify a number of software development tools and generate executable tools from the specifications. Among the most complex of these are tools for editing and maintaining libraries of reusable C++ classes. These tools are contained in a PSDE that has been developed for British Airways within the GOODSTEP research project [GOO94].

The particular research contributions of this thesis are the following:

- We delineate the relevant requirements on software development tools.
- We elaborate on design rationales for these tools and in particular discuss their impact on a database system for tool construction.
- We show that object database systems are reasonably well-suited to tool construction and indicate why other database management systems such as relational or structurally object-oriented systems are inappropriate.
- We develop a tool architecture based on object database systems and identify reusable and tool-specific architecture components. Tools constructed with this architecture will be improved with respect to
  - maintenance of inter-document consistency constraints,
  - concurrent and distributed execution,
  - version management of documents,
  - robustness against software and hardware failures,
  - openness, and
  - efficiency.
- We define specification languages to develop the tool-specific components, such as the database schema, at appropriate levels of abstraction.

This thesis will further be structured as follows: In the next chapter, we discuss software development tools from a software developer's and environment builder's point of view. Their requirements determine a tool builder's requirements on a database system as the main building block of software development tools. These requirements are discussed in Chapter 3. In Chapter 4, we review different classes of available database systems. While doing so, we particularly focus on database performance since it dominates the tool's performance and, therefore, is a critical factor for tool acceptance. This review will result in the selection of object database systems as the most promising systems to take. We then present a tool architecture that is based on using object database systems in Chapter 5. We use this architecture to distinguish tool components into reusable and tool-specific components. Chapter 6 presents various tool specification languages for defining the varying components of a tool. The languages will support the definition of different concerns at the appropriate levels of abstraction and allow reuse of specification components based on the object-oriented paradigm. Chapter 7 is devoted to discussing requirements for an environment for tool specification based on these languages. In particular we define consistency constraints between the languages in order to define requirements for tool integration. In Chapter 8, we outline the design and implementation of compilers for the languages, which will generate the tool-specific architecture components. The approach taken in this thesis has been evaluated on the basis of two scenarios. The evaluation results are presented in Chapter 9. Finally, Chapter 10 summarises the main results and indicates open problems and further work that remains to be done.

Chapter 2

Tool Requirements

A process that develops and maintains a software system, which we hereafter refer to as a *software process*, consists of a number of different tasks. It usually includes a *requirements analysis* task where requirements of future customers of a software system are elicited. These requirements are transformed into an *architectural design* where the different components of a software system and relationships among them are identified. The export and import interfaces of these components are designed in more detail in a *component design* task. The components are implemented according to the design decisions in a *component implementation* task, and they are tested in a *component test* task. The different component implementations are integrated and an *integration test* task is performed. Different kinds of documentation are produced including a *technical documentation* and a *user manual*. The suggestion of the Waterfall model [Roy70] that these tasks be performed in a strictly sequential order has been proved infeasible [Boe88]. Instead the tasks are often carried out in an intertwined manner.

The purpose of each task is to produce a set of *documents*. Each document is written in a formal graphical or textual language that determines the *document type*. The document types that are typically used during requirements analysis include *data flow diagrams* [dM78], *entity relationship diagrams* [Che76] and *Petri Nets* [GL81]. Architectural design might be performed in an object-oriented manner using the *Booch design* language [Boo91] or the *object modelling technique (OMT)* [RBP+91]. More conventional languages for architectural design are the *INSCAPE* notation [Per89] or the Π-Language [CFGGR91]. Module interfaces may be defined in languages like the ones suggested by [Lew88] or in the Groupie module interface language [ES94]. Implementations are then carried out using programming languages like C [KR78], C++ [Str86] or Eiffel [Mey92]. Informal documents, such as user manuals or technical documentation, are often written in mark-up languages, like *SGML* [ISO86].

A particular mix of document types that is appropriate in one process need not be appropriate for another. A process developing a real-time application, for instance, should use a requirements' definition language that can express response time constraints, but such a language might be unnecessarily complicated for customers of a banking application where response time constraints need not be expressed. This means that it is impossible to find **the** mix of document types that could be used in arbitrary software processes. One of the main goals during *software process modelling* is, therefore, to identify those document types that are most appropriate for the different tasks that have to be carried out during a particular kind of software process (For the tasks that have to be performed during process modelling, c.f for instance [JPSW94]).

There are a number of different consistency constraints that documents must obey. Apart from static semantic constraints of the formal languages, there are also consistency constraints between different documents. These *inter-document consistency constraints* are not confined to documents of the same type but frequently exist between documents of different types. An important factor for the quality of a software system is then whether these constraints have been defined properly and are respected by the documents produced during the process.

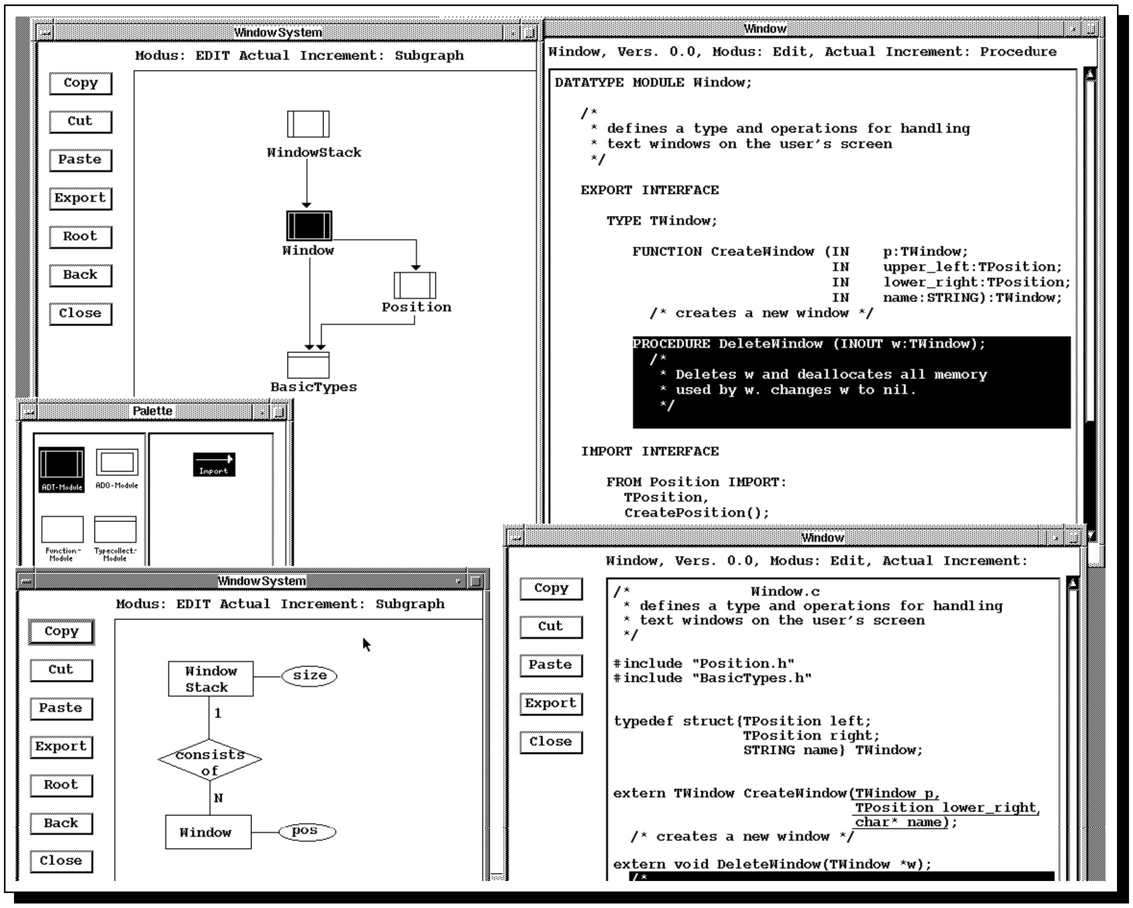
This chapter is further structured as follows. In the first section we introduce a scenario that provides a number of examples for inter-document consistency constraints. We will use the scenario for illustration purposes throughout this thesis. In the second section we delineate requirements that arise due to the fact that tools are used in a multi-user environment, namely a process-centred software development environment (PSDE). We consider inter-document consistency constraints in more detail. Then we discuss requirements that are concerned with the integration of tools with a process engine. After that we outline a number of requirements that trace back to the fact that multiple users use tools concurrently to develop related documents. The consequences of these requirements are that users must modify the syntactic structures of documents incrementally. In the second section we then delineate requirements that follow on from this consequence. We delineate requirements for document editing, analysing and browsing and consider persistence of documents and integrity preservation. Then we discuss the response time requirements that tools should meet. In the third section we identify the need for supporting the efficient construction of tools that meet our requirements and finally, in the last section, we discuss related work.

2.1 An Illustrating Scenario

Figure 2.1 displays four different documents of four different types. Starting from the bottom left, there are in clock-wise order an entity relationship diagram, an architectural definition that identifies different kinds of modules as components of a software system[1], a module interface specification that identifies exported types and operations as well as an import interface, and a module implementation that implements the exported types and operations of a module in the C programming language. There are a number of inter-document consistency constraints between these document types. Entities of the entity relationship diagram, for instance, must be refined in terms of abstract data type modules in the architecture diagram. Modules in these diagrams, in turn, must be refined by a module interface definition, that defines the export and import interface in detail and independent of a particular programming language. Each arrow of the architecture diagram should appear as entry in the import interface of the module interface definition. Module interfaces must then be implemented, in this example in the C programming language. Operations and types that have been identified in the export interface must be properly implemented in the C document. Therefore, parameter lists and result types should match each other. Moreover, import interfaces are refined by pre-processor `#include` statements. Vice versa, there should be no such statements when the design does not include the respective entry in the import interface, otherwise there would be dependencies among source code components that are not properly reflected in the design.

The need arises to assist software developers in the production of documents that meet inter-document consistency constraints like the ones outlined above. Developers, therefore, require a *tool* for each document type. Such a tool should then support the editing of multiple documents

[1]The detailed notion is of no concern here and we refer to [ES94].



Window, Vers. 0.0, Modus: Edit, Actual Increment: Procedure

```
DATATYPE MODULE Window;

   /*
    * defines a type and operations for handling
    * text windows on the user's screen
    */

   EXPORT INTERFACE

      TYPE TWindow;

         FUNCTION CreateWindow (IN    p:TWindow;
                               IN    upper_left:TPosition;
                               IN    lower_right:TPosition;
                               IN    name:STRING):TWindow;
           /* creates a new window */

         PROCEDURE DeleteWindow (INOUT w:TWindow);
           /*
            * Deletes w and deallocates all memory
            * used by w. changes w to nil.
            */

   IMPORT INTERFACE

      FROM Position IMPORT:
        TPosition,
        CreatePosition();
```

Window, Vers. 0.0, Modus: Edit, Actual Increment:

```
/*               Window.c
 * defines a type and operations for handling
 * text windows on the user's screen
 */

#include "Position.h"
#include "BasicTypes.h"


typedef struct{TPosition left;
               TPosition right;
               STRING name} TWindow;


extern TWindow CreateWindow(TWindow p,
                            TPosition lower_right,
                            char* name);
  /* creates a new window */

extern void DeleteWindow(TWindow *w);
  /*
```

Figure 2.1: Inter-Document Consistency Constraints

of that type, it should be supportive in analysing static properties of documents, browsing to semantically related documents and most important, it must check for inter-document consistency. In fact, the example of Figure 2.1 displays the user interfaces tools contained in the *Groupie* environment [ES94]. The tool in the lower-left corner can be used to edit, analyse and check entity relationship diagrams. Then there is a tool for architecture definitions, a module interface definition tool and a programming tool for the C programming language. To be able to check inter-document consistency constraints the tools must be integrated. We refer to a set of integrated tools that support all document types of a software process as a *software development environment* (SDE).

Most software processes are conducted by multiple rather than single software developers. That means that different developers use tools to produce different documents. Different *versions* of documents must be managed to facilitate independent document development. However, due to inter-document consistency constraints, the development cannot be performed in complete isolation. At some point in time, the documents produced by one developer must become consistent with documents produced by other developers. Consider in the above example, that a requirements engineer uses the entity relationship tool to define the information model of a software system, while a system architect is in charge of the architectural design of the system. Their documents should be consistent with each other before implementation begins, otherwise significant effort might be wasted during implementation if, for instance, wrong names are used or it turns out that an implemented module is obsolete.

2.2 Multi-User Support

A great number of documents have to be produced during a software process. We, therefore, require a central instance that keeps track of the states of these documents and of the responsibilities of different developers for the documents. We refer to this instance as *process engine*. It advises multiple developers as to which activities they should perform in order to proceed with the overall software process. A software development environment whose tools are used under control of a process engine is referred to as a *process-centred software development environment* (PSDE) hereafter. The software developers that use a PSDE are referred to as *users* in the following. We now discuss requirements that stem from the fact that multiple users participate in a software process.

2.2.1 Inter-document Consistency Constraints

Different levels of granularity have to be considered for the specification of inter-document consistency constraints and their validation. From a process point of view, on a coarse-grained level, we are concerned with which set of documents a particular document is consistent with or not. A document that is not consistent with related documents requires further attention. Hence the process engine must be aware of consistency constraint violations when it schedules user activities (c.f. Section 2.2.2 below). Tools must consider inter-document consistency constraints on the level of syntactical units of documents that we refer to as *increments*. This fine-grained level is required to be able to detect or preserve inter-document consistency constraints and display appropriate error messages to users.

The fact that multiple users might be involved in achieving inter-document consistency has an impact on how tools have to handle constraint violations. Different strategies can be considered for that. A tool might handle a constraint violation in a lazy way and only visualise an inconsistency to the user as soon as it has been introduced. This visualisation might be achieved by the use of colours or by underlining. Users might want to have a more detailed error message, but this should be provided on demand in order not to overload the document representation. A tool might also follow an eager approach and reject user interactions that would violate an inter-document consistency constraint. A tool might even automatically correct erroneous increments. Upon a change of one increment, it can, for instance, automatically modify related increments in other documents in such a way that consistency is retained. We refer to these automatic modifications of related increments as *change propagations*.

In order to discuss examples of these different strategies let us revisit Figure 2.1. The lazy approach of only visualising inconsistency is, for instance, applied to the consistency checks between parameter lists in operations of the module interface and the respective implementation. As the parameter lists of operation `CreateWindow` in the interface and the implementation do not match, the list in the implementation is underlined. A constraint whose violation should not be tolerated is the consistency between module names in the architecture, the interface design and the implementation. The module interface tool and the implementation tool, therefore, might prevent interface designers or programmers from changing the name. The chief architect, however, might want to be able to make such a change. In that case all appearances of the module name should be changed consistently in all documents that refer to the name. The change may become effective immediately or be deferred until a time when the user responsible for an affected document decides to incorporate the change.

During maintenance, a software system might have to be extended to meet additional customer requirements. Then not only the source code, but also the requirements definition, the architectural and component design, the user manuals and the technical documentation have to be updated in such a way that all inter-document consistency constraints are preserved. Even during the initial development it happens that a design decision turns out as unfortunate only when the respective component is implemented or tested. Then the design has to be improved and the consistency of depending documents like implementation, source code and test plans either has to be preserved in terms of change propagations or the resulting inconsistency has to be visualised. Users might want to see the affected documents at the same time. Tools must, therefore, facilitate the *intertwined development* of documents and display the effect of a change immediately. In order not to destroy unaffected parts of other documents, tools should support the *incremental development* of documents. In particular, a propagation should change only those increments of other documents that have to be changed.

In order to be able to check inter-document consistency constraint checks and even preserve them in the way outlined above, tools have to be integrated. We distinguish *a-priori* and *a-posteriori tool integration*. A-priori tool integration implies that tools are newly constructed in such a way that they can inter-operate with companion tools to provide the required functionality for inter-document consistency. This might not always be feasible, for instance if existing tools have to be reused. Then tool integration must be achieved a-posteriori and be controlled externally. The tools must, therefore, be open and provide an interface that offers a subset of the incremental editing functionality as *services* to the external instance that controls a-posteriori tool integration.

We note that the most appropriate strategy for handling violations of a constraint depends on the process model and even on the process state. If a change propagation does not affect other users' documents it can be performed without any problem. If other users are affected, there may have to be negotiations about the change among the different users involved. Whether or not to perform propagation, therefore, depends on the allocation of tasks to PSDE users. Likewise, a constraint violation that is tolerable in an early stage of a software process might become intolerable when the process reaches a certain deadline where documents must become consistent. As an example, consider again the above scenario. An import from a non-existent module in a module interface definition is quite tolerable during the design. If the tool did not tolerate such errors bottom-up design would be enforced, which is not always appropriate. The constraint may, therefore, be violated temporarily. If implementation of the module has started, however, the constraint should not be violated because the implementation of the module might depend on a type that might never be implemented and if that is detected too late a significant amount of effort is wasted. Therefore, the need arises to integrate tools with the process engine that drives the software process.

2.2.2 Integration of Tools and Process Engine

A large number of documents are produced during a software process. An architectural specification of a complex software system will identify from several hundred to several thousand different components. For each component other documents like an interface definition, an implementation and a test plan have to be developed. The process engine will have to maintain document states because they change concurrently due to the work of multiple users, and users will not be able to keep track of that. Therefore, the process engine should guide users to those documents that require further attention based on document states.

The user interface that the process engine uses for this guidance is some sort of personal *agenda*. It visualises the user's current duties [PSW92, DG90, BFG93a]. The agenda is computed based on the responsibilities users have for documents and on document states. If a document is not yet complete, has been rejected during a review or contains inter-document consistency constraint violations it should be included in the agenda of the user who is responsible for the document. As an example, consider Figure 2.2, which depicts the agenda of the Merlin PSDE called *working context*. It contains icons for each document that a user needs to complete his or her duties. The user can select one of these icons and push a mouse button to obtain a menu that offers a list of *activities*. In the example, the working context offers an editing, viewing or printing activity for document `module_impl` to user Miller.

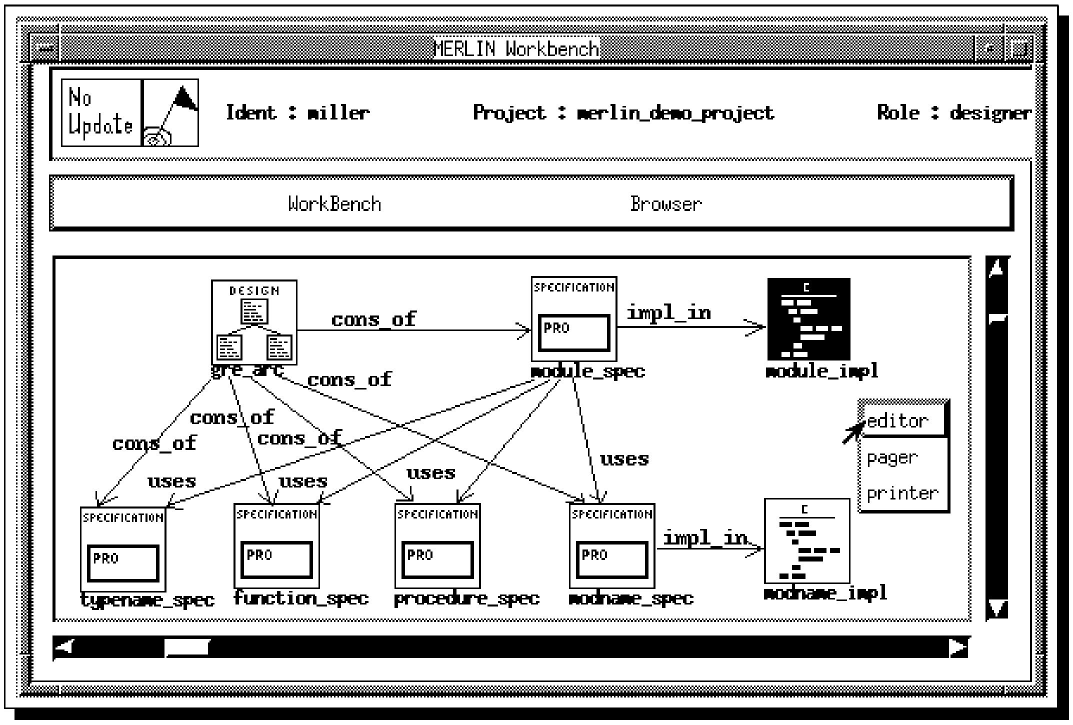


Figure 2.2: Working Context of the Merlin PSDE

The process engine must then react to the selection of one of the offered activities. In the above example, it must use the C programming tool to display the document implementation of `module_impl` in an edit window or to compute a textual representation that the process engine can pipe into a paging or printing command. The need for a communication link between process engine and tools arises. The communication link will be used by the process engine to invoke *services* from tools and the tools, in turn, will communicate the result of a service execution back to the process engine. Services can be classified into two categories: Generic vs. tool-specific services and synchronous vs. asynchronous services. As we shall see in Chapter 5, we can use this classification to identify the different service providers in the tool's architecture.

Generic services are those any tool should offer. Examples of these are creation of a new document, opening a document, computation of a printable representation of a document and so on. Beyond those services, any tool may offer the process engine *tool-specific services* that implement specific tasks for the tool. As an example, imagine a module interface editor that offers a service for a-posteriori integration to add or delete import relationships.

When requesting a *synchronous service*, the process engine awaits service execution and then continues. Computation of a printable document representation is an example of this. Unlike synchronous services, the process engine cannot wait for the tool to handle an *asynchronous service*. An edit service is an example. If the process engine waits for its completion, all other concurrent tasks will be blocked. After having handled an asynchronous service, the tool has to communicate termination of the service execution to the process engine.

Tools in PSDEs perform different kinds of accesses to documents which must be in-line with the *access rights* defined in the process model. In [PSW92], tool accesses are classified as read, write and execute accesses. Read accesses are performed, for instance, with browsers, static analysers or printing tools, whereas write accesses can only be performed with editors. Execute accesses are performed with tools that can run programs such as a debugger, which we do not consider any further. Thus, we regard an execute access as a kind of read access.

In order to enable the process engine to communicate the kind of document accesses granted to a user, we introduce the notion of *open modes*. The service used to open a document then gets the open mode as an additional parameter. The parameter can take the values *update* and *view*. A document opened in *view mode* cannot be changed. When opening a document in this mode, modifications of the document must be inhibited. Documents opened in *update mode* can be changed.

Some actions that users perform with tools not only modify a document, but also the document state on which the agenda computation is based. We refer to these situations as *process events*. As an example, consider the deletion of an exported operation in the scenario of Figure 2.1. There the states of those modules that import the operation and now have become inconsistent have to be reverted so that the modules are included in the agendas of the users responsible for these modules. Therefore, tools must inform the process engine of process events in order to enable the agenda to be recomputed and to reflect the new document states.

The question arises whether all process definitions should still be implemented solely by the process engine or partly by the tools. In the above example, determining access rights could also be achieved by tools. The process engine would then have to inform tools whenever owners of documents were changed. Tools could then store ownership information of documents persistently. To determine the access rights of a particular user to a document the tool could then check whether he or she was the owner and only then provide the full tool functionality, otherwise it might restrict the applicable functionality to read-only operations. Whether the process engine or tools are more appropriate to implement a particular process definition depends on the probability that the definition changes. Process programming languages that are executed by process engines are defined in such a way that process programs can easily be changed, even without terminating their execution [PS92, BFG93a]. Tools, however, are much more difficult to change. Hence, any process definition that is likely to change should be implemented by the process engine rather than a tool. On the other hand, tools might be able to execute particular definitions more efficiently and the process model will become simpler if it can focus on the relevant concerns. If a particular definition is unlikely to change, for instance because it has been successfully used in a number of other processes, the process programs will get simpler and the PSDE might perform more efficiently when the definition is implemented in the tool. Thus we refer to these tools as *process-sensitive tools*.

2.2.3 Versions

The multiple users that perform a software process edit, analyse, browse and check documents concurrently. Documents are the unit of granularity for assigning responsibilities to users. Usually only one user is responsible for a document at the same time. Then this user is the only one allowed to edit the document in update mode and other users do not edit the document in this mode concurrently. Due to inter-document consistency constraints, however, the activities of one user might interfere with concurrent activities of other users. Versions are used to allow users to edit documents in *isolation* for a certain period of time. At some point, however, all the document versions that belong to a configuration must become consistent with each other. This is, for instance, the case when a configuration of a software system has to be delivered. Then users have to share their versions and edit them cooperatively in order to obtain a state of inter-document consistency. To facilitate this *cooperation* the effect of an update of one user must become visible as soon as possible. It must even become visible when other users concurrently edit affected document versions. We now discuss in more detail the functionalities that are required from tools to achieve isolation and cooperation.

We distinguish two notions of document versions, *revisions* and *variants*. Revisions are linearly ordered in the development history of a document. A later revision improves a predecessor revision and will, therefore, replace it at some point in time. Revisions are not only needed to facilitate the above required isolated development, but also to establish checkpoints. A user might then "undo" a set of modifications by reverting to a previous revision if they turn out to be inappropriate. Variants emphasize different aspects of the development [Tic85, SS95]. A user may, for example, have to develop a document in different ways in order to meet the requirements of different customers. Unlike revisions, whose development history is linearly ordered in time, variants are two or more parallel branches that coexist for a certain period of time. If it is not important to distinguish between revisions and variants, they are subsumed under the term *version*.

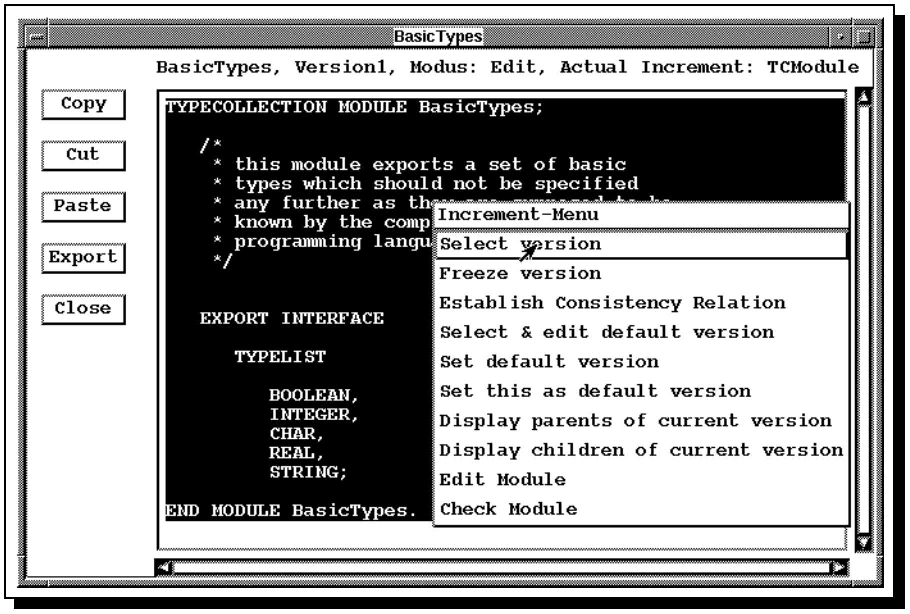
Upon creation of a document an initial or *root version* is created and selected. Tools should then offer facilities for deriving new versions from a selected version. This derivation defines a predecessor/successor relationship between versions. The version from which another version is derived becomes the *predecessor*. The derived version is the *successor*. Before a version can be derived from a predecessor version, the predecessor must be *frozen* so that its contents cannot be changed any further, otherwise the predecessor/successor relationship would not reflect the development history of the document.

The tool must offer facilities for browsing through the development history. It should, therefore, provide ways of displaying a document's version with its predecessor/successor relationships. The user may then want to select one of these versions as the *current version*. The tool should then apply all changes to that version only. The tool must respect the status of that version. If it is frozen, it must prevent the user from modifying it.

Users may want to establish a *default version* of a document, which is the version that is seen as long as no current version has been designated. The tool must store, for each document, the default version information persistently.

Sometimes different variants must be *merged*. If we reconsider the example given above, the variants would have to be merged if an implementation were found that would meet several customer requirements. Merging cannot be achieved fully automatically. If an increment is changed in two different variants, only the user can decide which variant of the increment should be taken in the common successor version. Therefore, tools have to support the interactive process of merging variants.

Finally, users may want to reduce the number of revisions stored in the development history. The tool must, therefore, offer ways of deleting any version except the root version. During such a deletion, the predecessor/successor relationships must be redirected accordingly.


BasicTypes
Copy
Cut
Paste
Export
Close
Increment-Menu
Select version
Freeze version
Establish Consistency Relation
Select & edit default version
Set default version
Set this as default version
Display parents of current version
Display children of current version
Edit Module
Check Module

BasicTypes, Version1, Modus: Edit, Actual Increment: TCModule

```
TYPECOLLECTION MODULE BasicTypes;

   /*
    * this module exports a set of basic
    * types which should not be specified
    * any further as th
    * known by the comp
    * programming langu
    */

   EXPORT INTERFACE

      TYPELIST

         BOOLEAN,
         INTEGER,
         CHAR,
         REAL,
         STRING;

END MODULE BasicTypes.
```

Figure 2.3: User Interface for Version Management in Groupie

Figure 2.3 continues our example and displays how version management functionality can be offered at the user interface. We assume a style of interaction where users select an increment of the document. Then the tool deduces a set of possible *tool commands* that are applicable to this increment and presents them in a *context-sensitive menu* to the user. If the user selects a complete document and the document is under version control, the computed menu contains all the commands for version management.

Different users will have to cooperate, for instance, to achieve inter-document consistency of the document versions they are responsible for. This implies that they share their document versions. To achieve tight cooperation, users then want to see the impact of each other's updates as soon as possible. Note that an update might not necessarily involve changes to the contents of a document. We will also have to consider the correction or introduction of an inter-document consistency as an update, since this changes the internal state of increments. Tight cooperation then requires an update to a document version to be done in such a way that all tools concurrently displaying the document version are informed of the update as soon as possible. They should then reflect the update as well. In the above example of inconsistent parameter lists, a designer might remove the inconsistency by deleting the additional parameter. If a programmer is accessing the implementation document that corresponds to the implementation, he or she should see, as soon as possible, that the inconsistency has been resolved and requires no further attention. The shared and cooperative updates of document versions must, therefore, not be disabled by exclusive locking of document versions as suggested in the check-in/check-out model [Tic85].

Concurrent changes of multiple users, however, cannot be performed in a totally unrestricted way. This is due to the *lost update* and *inconsistent analysis* problems, known from concurrency control in database systems [Dat86]. Assume that two programmers concurrently create two new module implementation documents. Assume further that both documents have to be inserted into a shared table of available module implementations in order to be able to check for existence and uniqueness of module implementation names. The tool of the first programmer, therefore, reads the table from some shared memory, inserts the document into the table and writes the updated table back to shared memory. If concurrency is not restricted the table update might now be lost due to the fact that the second programmer's tool read the table before the other update was written, inserted the other new document and wrote its copy of the table back after the other tool. For the inconsistent analysis problem, consider the following scenario. A programmer changes the name of an implementation document. A concurrently working programmer creates a new `#include` statement referring to the old module implementation name. During that time, the included document name is searched in the table of available modules. An inconsistent analysis problem occurs if this search is performed after the other tool has checked for `#include` statements to be marked as erroneous and before the table has been updated with the new document name. Then the `#include` statement will not be displayed as inconsistent although the referenced module no longer exists.

Now we have encountered the dilemma that we cannot lock document versions exclusively while they are being edited without hampering cooperation. On the other hand, we must perform locking to avoid the lost update and inconsistent analysis problems. The dilemma is solved by requiring tools to decrease granularity with respect to both the subject that performs locking and the objects that are being locked. This means that tool sessions must be considered as sequences of shorter execution units, each of which is executed in isolation from concurrent execution units. The units achieve isolation by locking objects in a traditional way [Gra78]. An object is locked in shared mode when the object is read and in exclusive mode when it is updated. While shared locks are compatible to each other, any other combination reveals a concurrency control conflict that must be resolved by the tools. To decrease the probability of concurrency control conflicts, execution units should not lock the complete representation of a document version, but only the representation of those increments that are being accessed or updated during the execution of a unit. In the examples that encounter lost updates or inconsistent analysis problems, we would then obtain a concurrency control conflict. Tools should react to these conflicts by delaying the execution of one unit to await completion of the conflicting unit, that is until conflicting locks have been released.

Summary

In short, we require from tools to handle isolated and cooperative updates of documents in different ways. In early stages of document development editing should be supported in such a way that other users do not interfere with an update. Versioning is required for that purpose. When inter-document consistency among different users has to be achieved, the same document version must be concurrently updatable by multiple users to facilitate the required cooperation. Locking must be performed to avoid lost updates and inconsistent analysis. The granularity of the subject that performs locking should be decreased from sessions to shorter execution units and the granularity of objects that are being locked should be decreased from documents to increments.

2.3 Document Production Support

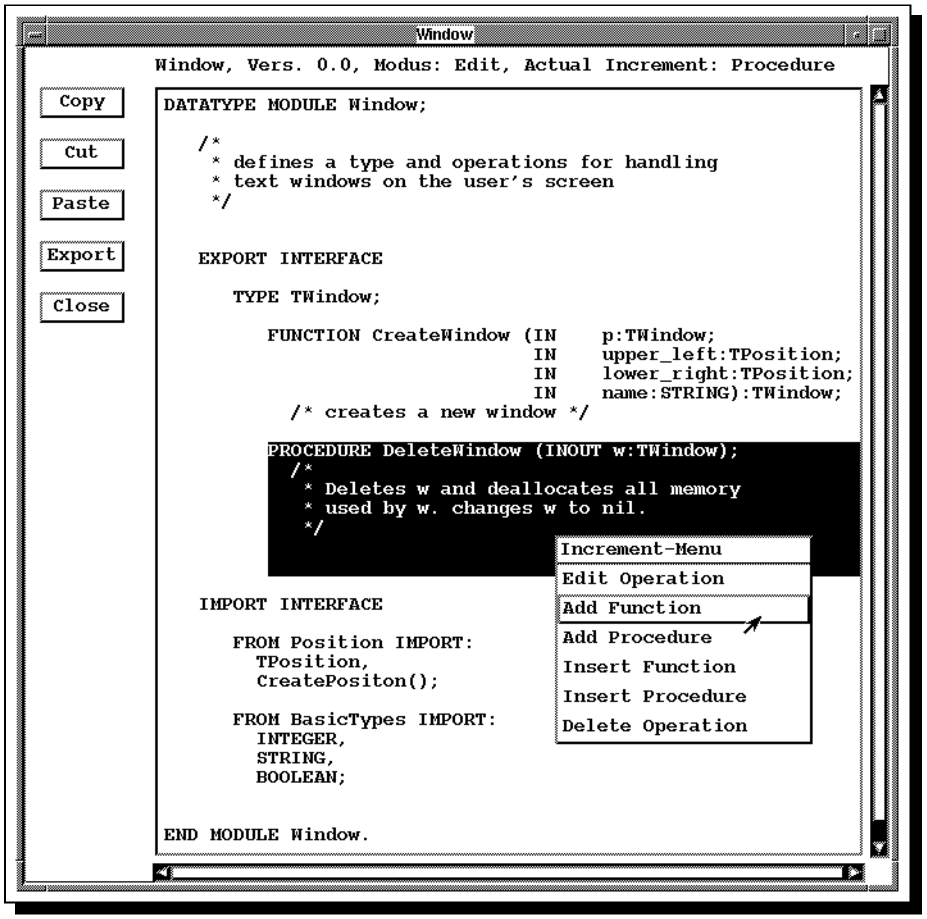
Checking and preserving inter-document consistency constraints requires the access and modification of documents based on their syntactic structure. Moreover, concurrent accesses and updates of the same document version require the access and update of the fine-grained syntactic structure of the document in order to avoid unnecessary concurrency control conflicts. Tools should, therefore, provide users with the means to access and update the syntactic structure of documents incrementally [ELN+92] in order to facilitate the required incremental and intertwined document production.

2.3.1 Editing, Analysing and Browsing

We refer to such tools as *syntax-directed tools* [RT81, DGKLM84, Rei84, Nag85, HN86]. Document accesses and updates are issued in terms of *commands*. Commands in syntax-directed tools are directed towards the syntax of the underlying language. This is done by relating each command to an increment. A user can use a pointing device such as the mouse of a graphical workstation to select a particular increment as the *current increment*. The tool can then use this selection to deduce possible commands and present them to the user in a menu. This *structure-oriented mode* of editing particularly supports the incremental and intertwined development of documents. The effect of consistency-preserving change propagations, for instance, are displayed immediately in all opened documents. Also constraint violations, if tolerated, are immediately visualised in all documents that are displayed.

As an example, Figure 2.4 displays the user interface of the Groupie module interface editor, which is part of the environment depicted in Figure 2.1. By means of a click into the window with a mouse button, the tool selects the current increment and highlights its textual representation. A push with another mouse button causes the tool to deduce a set of possible commands applicable to the selection and offers them in a pop-up *menu*. The user may then select a command, for instance, in the above situation `Add Function`. The tool will then insert a new function template after the current increment. A change propagation might then also insert a template for a C function into the implementation document.

We note that the editing commands, as introduced above are an appropriate choice for the execution units that are to be performed in isolation from each other. They are considerably short and most often only access a small fragment of a document. When commands are performed in isolation from each other, only small portions of documents are locked and this only for a very short period of time. Concurrent users then see the effect of a change as soon as the command that changed a document version has been completed.



```
Window, Vers. 0.0, Modus: Edit, Actual Increment: Procedure

DATATYPE MODULE Window;

   /*
    * defines a type and operations for handling
    * text windows on the user's screen
    */


   EXPORT INTERFACE

      TYPE TWindow;

         FUNCTION CreateWindow (IN    p:TWindow;
                                IN    upper_left:TPosition;
                                IN    lower_right:TPosition;
                                IN    name:STRING):TWindow;
           /* creates a new window */

         PROCEDURE DeleteWindow (INOUT w:TWindow);
           /*
            * Deletes w and deallocates all memory
            * used by w. changes w to nil.
            */

   IMPORT INTERFACE

      FROM Position IMPORT:
        TPosition,
        CreatePositon();

      FROM BasicTypes IMPORT:
        INTEGER,
        STRING,
        BOOLEAN;

END MODULE Window.
```

Figure 2.4: Structure-oriented Syntax-directed Editing

The structure-oriented mode of editing a document is not always appropriate. In some cases, it has drawbacks compared with conventional text editing. It is inferior if

- many template insertions, compared with the number of keystrokes in conventional text editors, are required in order to fully expand an increment. This is, for instance, the case with identifiers or expressions.
- complex increments must be changed in a way that cannot be anticipated. Suppose, for instance, that a large `while` statement is to be transformed into an if statement without affecting the body.
- documents only available in a textual representation must be reused. This is always the case if development started with conventional text editors that are later replaced by more sophisticated tools, like the ones discussed here.
- experienced users who master the language are capable of typing very fast. Such users may be faster typing an increment than repeatedly moving their hands from the keyboard to the mouse, selecting an increment, popping up a menu and choosing a command.

To address these problems, users expect from syntax-directed tools not only support for structure-oriented editing, but also facilities for *free textual input* of increments with conventional text editors. To ensure syntactic correctness, tools must parse new texts as soon as the user has finished editing. If a text contains syntax errors, they must be brought to the user's attention.

Besides inter-document consistency constraints, documents also have to obey the *static semantics* of the respective languages. In principle the same approach as with inter-document consistency constraints can be applied to check, visualise and handle static semantic errors. Unlike inter-document consistency constraint violations, static semantic errors only affect a single document. Since normally, only a single user is responsible for the document, it is appropriate to preserve static semantic correctness by means of change propagations.

To improve the quality of documents further, users may require *static analysis* capabilities that go beyond analysis of static semantics. Users of a programming language editor, for instance, may wish to find obsolete variable declarations, unreachable statements or use of uninitialised variables [ES89]. During the maintenance of a system, users benefit significantly when requirements analysis or design tools offer *cross-reference analysis* facilities that can be used for change impact analysis. Therefore, for those increments that are related to other increments, tools should offer commands that display the other increments.

Particularly during maintenance, there is a need to support users that are not familiar with a document in understanding the document and its relationships to other documents. Beyond merely displaying related increments, this requires browsing along relationships in order to read the context of the related increment. A set of documents can be considered as a *hypertext* where the relationships between increments define hypertext links. As a hypertext viewer can follow hypertext links, tools should offer *browsing facilities* that open documents containing related increments.

2.3.2 Persistence and Integrity

Editing sessions are interrupted for three different reasons. Usually the interrupt is due to an explicit decision taken by the user, for instance to perform some other task or to leave off work. Secondly, there are situations where the process engine has to interrupt a user's editing session, for example if a designer deletes a module in an architecture while a programmer is concurrently implementing that module. According to [Wol94], a consistency preserving transaction must then stop the editor used by the programmer. Thirdly, editing sessions may be interrupted accidentally. Consider as examples a loss in the power supply or a failure in the operating system that causes a reboot.

In order to anticipate these situations, documents must be stored persistently so that the editing session can be resumed without any significant loss of effort. In the first two cases, the user or the process engine could take explicit measures to save a document. While we could expect the user to save every document before quitting a tool, the process program enacted by the process engine would be unnecessarily overloaded if it had to define these saving procedures. Dealing with accidental interrupts is even more complicated. Such an interrupt could occur during execution of a tool command that changes a set of documents. Then the *integrity* of these documents, that is their immediate usability by the same or other tools, will be violated if some parts are changed while changes to other parts are omitted due to the failure.

To preserve effort from accidental interrupts and to relieve the process programmer of having to define document saving procedures, we require *persistence* of documents to be transparent to both user and process model. Persistence and integrity should be achieved as follows. We require of tools that the effect of a command execution is persistent if and only if it is completed. Moreover, tool commands must be designed in such a way that the integrity of documents is guaranteed whenever a command execution is completed. In the case of a hardware or software failure, we require tools to automatically *recover* to the state after the last completed command execution. This reduces any potential loss of effort to the amount spent during the last command whose execution was not completed and ensures integrity of documents against any failures.

2.3.3 Efficiency

Among the chief-factors for user acceptance of a tool is the tool's *performance* during command execution. If the user has to wait for tool commands to be executed, the tool will hardly ever support the user in efficient document production. We, therefore, discuss the performance required from tools.

Very fast typists can type about 300 characters per minute. In this case, the time between successive keystrokes is about 200 milliseconds. Thus any response time of a tool that is below 200 milliseconds is non-critical, since users will not recognise it as a delay [ESW93].

Unlike secretaries, users do not type continuously. They frequently pause between two command executions, for instance to use a pointing device to select the next current increment or to choose an interaction from a menu. If the non-trivial processing that a tool carries out is aligned with these natural breaks between the execution of two commands, response times up to a second are acceptable.

In other rare circumstances, users may accept higher response times if they can be justified by the complexity of the task concerned [WBK91]. No user, for instance, would refuse to use a compiler just because it needs more than a second to compile a source.

2.4 Tool Generation Required

Depending on a particular process and even a particular project, a large variety of different languages and corresponding documents may exist. As the design and implementation of syntax-directed tools is mainly driven by the syntax of the language, they have to be built anew for each language. Designing and implementing a tool is a significant task.

Moreover, often companies do not use a language as defined, but have their own guidelines that extend a language or only use a subset. Unless there is an instance that enforces usage of these guidelines, users will hardly ever obey them. These guidelines can easily be enforced, if syntax-directed tools used for document production are confined to these guidelines and only enable the defined language subset to be edited. As a consequence, however, tools have to be built anew for each company that has different guidelines.

The handling of inter-document consistency as required in Section 2.2.1 depends on the process specific mix of languages. If tools have to enforce inter-document consistency constraints, they must be customised whenever different languages are used in different software processes. Moreover, the construction of process-sensitive tools might require customisations whenever the tool is used in a different process. In the example given above, the process-sensitive tool checking access rights might have to be customised if it is to be used in a process with a different notion of access rights.

As a consequence, there is a need for tool construction or at least tool customisation for each company, for each different process model and since process models change [MS91], even for different process states. Our concern is to ease this tool construction and successive tool customisations as much as possible. We, therefore, focus on *generating* integrated syntax-directed tools from appropriate high-level specifications. Customisation of tools can then be done by changing the specifications and generating the tools anew.

2.5 Related Work

One could use compilers [ASU86, WG84] for each textual language that is used in a software process. The compiler would then check syntax, static semantics and inter-document consistency constraints between textual documents of the same type. For programming languages, we can assume that compilers are available. For languages where compilers are not available they could be generated with compiler generators like Eli [GHL$^+$92] or Cocktail [GE90]. During code generation, a compiler might translate a document of one type into a document of another type in order to start off the development of documents of the other type. This approach is followed in the early versions of the *ProMod* SDE [Hru87] or in the *Software through Pictures* environment [WP87]. In the example of Figure 2.1, a compiler for the module interface definition language could, for instance, generate an initial C code frame that then had to be refined further by programmers.

A number of our requirements, however, will remain open. A compiler can detect static semantic errors, but it will not be possible to prevent a user from introducing such errors into documents. The cases outlined above, where errors must not be tolerated, cannot be dealt with. Moreover, compilers only consider a single input language and cannot check for consistency constraints between documents that are written in different languages. Also the translation of documents into other types cannot be applied in general in order to facilitate incremental propagation of changes. In the example above, where a name has to be consistently changed throughout all documents, the transformational approach will fail. If we compiled changed module interface documents anew in order to generate consistent C code frames that incorporate the new module name, all the refinements that programmers had already made to the implementations of these modules would be lost. This problem could be solved, to a limited extent, with incremental compilers. They also fail if a change propagation violates static semantic constraints in the target language. Then a further change propagation would be required in the target language, which an incremental compiler of the source language is not capable of. In addition, compilers cannot provide the services that the process engine could use to implement user activities or a-posteriori tool integration, and compilers cannot notify the process engine of the occurrence of process events. Finally, a significant number of languages used in software processes are graphical. For these languages it is not possible to construct a compiler with current state-of-the-art in compiler construction.

A PSDE will, nevertheless, have to include compilers and debuggers for the programming languages used. In particular, we will require compilers for the generation of executable code, because we have not required this from tools. Instead we require tools that are used for editing, analysing and browsing programming language documents to be able to interface with compilers. They will, therefore, have to dump programming language documents into files of the operating system's file-system in such a way that they are accessible for a compiler of the respective programming language.

Chapter 3

Requirements on Databases for PSDEs

It is commonly accepted that documents in software development environments must be stored in a *database* of some sort [GL85, LS88, Ber87, PPT88, KFP88, BOSV89, HW91]. Databases are created, accessed, updated and administered using the functionality provided by a *database system*. In this thesis we consider *database systems for software engineering* (DBSEs) as a basis for tool construction. The tool requirements delineated in the last chapter now determine a tool builder's requirements on a DBSE. We, therefore, consider the impact of tool requirements on the DBSE that is used for tool construction. While doing so, we refine and complete earlier published requirement sets such as [Ber87, DEH$^+$91, ESW92, ESW93].

3.1 Persistent Document Representation

3.1.1 Document Representation

The common internal representation for documents manipulated by syntax-directed tools is an *abstract syntax tree* of some form [RT81, DGKLM84, HN86]. Nodes in the abstract syntax tree often have additional attributes whose values represent semantic information such as references to a string table, symbol tables or type information. Operations that tools perform in the structure-oriented mode of editing can easily be implemented as operations on this abstract syntax tree. Template insertion, for instance, is implemented as subtree replacement. After free textual input, the abstract syntax tree can be established with techniques well-known from compiler construction [ASU86].

Static semantic checking of a document that is represented as an abstract syntax tree can be done by *attribute evaluations* along parent/child paths in the document's attributed abstract syntax tree [Knu68]. The evaluation paths are computed at tool construction time based on attribute dependencies. If inter-document consistency checks between different documents are implemented by attribute evaluations, all inter-document consistency constraints must be checked at an artificial root node which has sub-trees for each document. With respect to concurrent tool execution many concurrency control conflicts arise at these root nodes and decrease efficiency. Therefore, techniques based on the introduction of additional, non-syntactic paths for more direct attribute propagation have been developed [JF82, Nag85, Hoo87]. They generalise the concept of abstract syntax trees to *abstract syntax graphs*. Such non-syntactic paths implement *semantic relationships* which connect syntactically disjoint parts of possibly different documents even of different types. They can be used for consistency checking, change-propagation when the document is changed and even for implementing static semantic analysis and browsing facilities. To handle these semantic relationships in a consistent way, the obvious strategy is to view the set of documents making up a project as a single *project-wide abstract syntax graph*.

We note that this generalisation to a single project-wide graph does not necessarily undermine the concept of a document as a distinguishable representation component. If we distinguish between *aggregation edges* in the graph, which implement syntactic relationships, and *reference edges*, which arise from semantic relationships, then a document of the project is a subgraph whose node-set is the closure of nodes reachable by aggregation edges from a document node (i.e., a node not itself reachable in this way), together with all edges internal to the set[1]. The edges not included in this subgraph are then necessarily the inter-document relationships inherent in the project. When we use the term abstract syntax tree of a document in the following, it shall denote all nodes reachable from a document node together with all aggregation edges internal to the set. Nodes that cannot have outgoing aggregation edges are called *terminal nodes*, for their origin lies in terminal symbols of the underlying grammar. Those nodes that may have outgoing aggregation edges shall be called *non-terminal nodes* accordingly.

As an example, consider Figure 3.1. It outlines how the different abstract syntax trees representing documents of Figure 2.1 on Page 5 are integrated to a project-wide abstract syntax graph. Nodes of the abstract syntax graph are drawn as rectangles. The annotation given in a rectangle identifies the type of respective node. Edges are depicted as arrows. Aggregation edges are drawn as solid arrows. Intra-document reference edges are depicted with dashed arrows and dotted arrows denote inter-document reference edges. Inter-document reference edges implement semantic relationships between increments of possibly different document types. Attributes of terminal nodes are depicted as quoted annotations close to the rectangles that represent the node. They represent lexical values identified with the respective terminal symbols. Additional attributes are required for the graphical documents in order to store coordinates or even lists of coordinates for the graphical layout of increments. These are for reasons of brevity omitted.

The refinement of entities defined in the entity relationship diagram in terms of modules of the architecture is reflected by inter-document reference edges labelled `ToArch`. Likewise, the refinement of modules of the architecture definition in terms of module interface documents is stored by means of reference edges labelled with `ToDesign`. Intra-type reference edges labelled `ToDecl` represent the use/declare relationship between type increments of a module interface document. The parameter type of function `CreateWindow` with the attribute `STRING`, for instance, has an outgoing reference edge to the node where it is declared, that is to the `TypeImport` node with attribute `STRING`. This node represents an import that is itself connected via an inter-document reference edge labelled `ToExport` to another node contained in the subgraph of module `BasicTypes` where the type is exported.

[1]What we call a subgraph here, is comparable to the notion of a *composite entity* in PACT VMCS (c.f. [Tho89]).

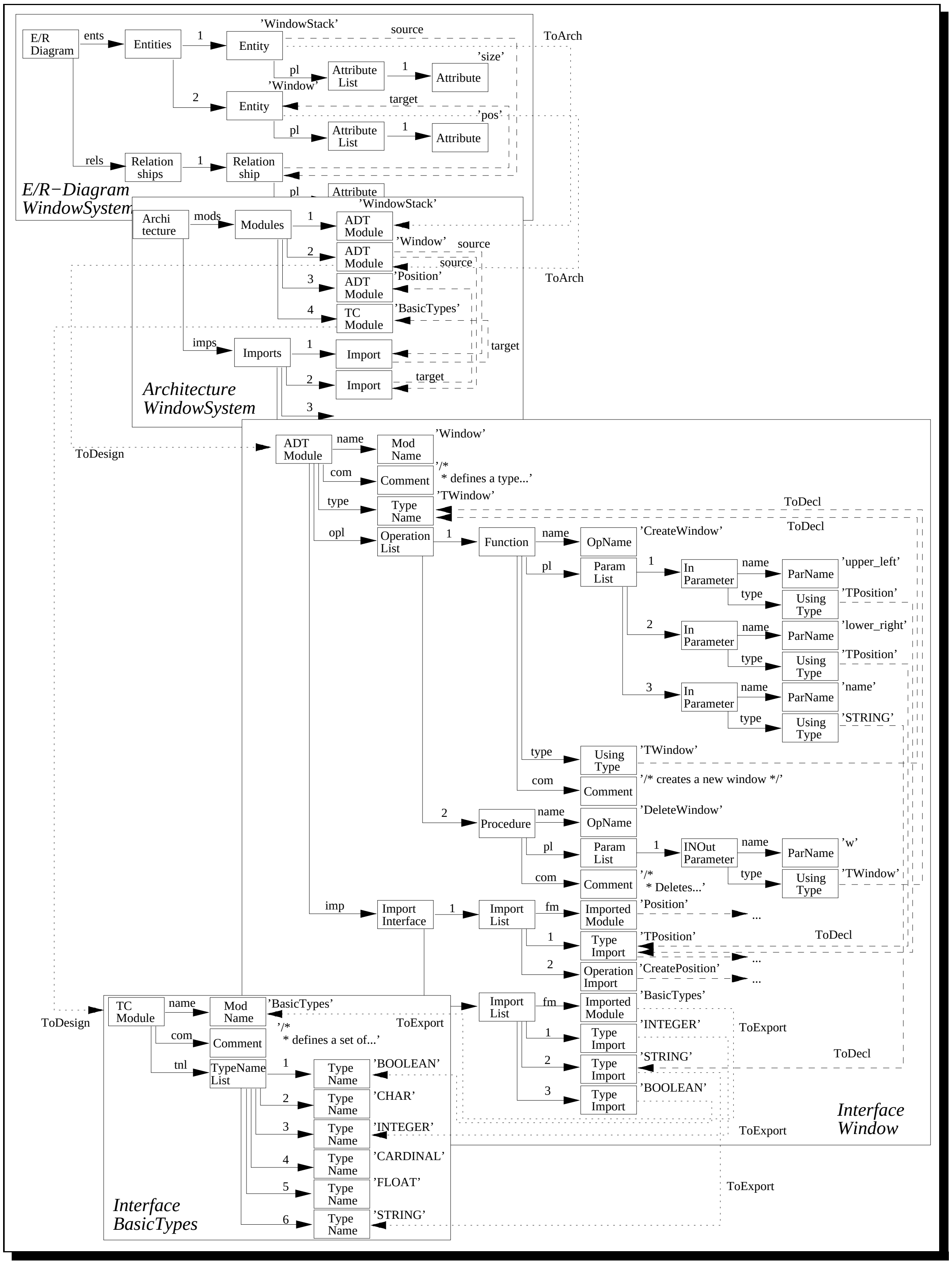


Figure 3.1: A Project-wide Abstract Syntax Graph

3.1.2 Persistent Representation

Due to the requirements of persistence and integrity, a persistent representation of each document under manipulation must be updated while user-commands are executed. Typically such an execution affects only a very small portion of the document concerned, if any. Given that the representation under manipulation is the project-wide abstract syntax graph, however, the update can easily become inefficient if, firstly, a complex-transformation between the graph and its persistent representation is required and, secondly, the persistent representation is such that large parts of it have to be rewritten each time, although it is not modified. This would for instance be the case, if we had chosen to store the whole graph or subgraphs in sequential operating system files that are updated at the end of each command execution.

Such inefficiency can be avoided if the persistent representation takes the form of an abstract syntax graph itself, with components and update operations that are one-to-one with those required by the tools concerned. The DBSE must, therefore, support definition, access and incremental update of a graph structure of nodes, attributes and edges with associated labelling information. The DBSE must avoid imposing limits on the overall size of the graphs it can handle. To preserve the integrity of the abstract syntax graph, the DBSE must support *atomic transactions* that enable a sequence of update operations to be grouped together so that they are either performed completely or not at all. To facilitate failure recovery to the state of the last completed command execution, completed DBSE transactions must be *durable*.

3.2 Data Definition and Data Manipulation Language

The kinds of nodes and edges required to represent a project, and the attribute information associated with each, cannot be determined by the DBSE itself. They should be defined by tool builders and then be controlled by the DBSE in order to have different tools sharing a well-defined project-graph. The overall structure of the project's syntax graph should, therefore, be defined in terms of the data definition language of the DBSE and established and controlled by the DBSE's *conceptual schema.*

As a minimum, we require that the *data definition language* (DDL) can express the different *node types* that occur within the graph, that it can express which *edge types* may start from node types and to which node types they may lead, and that it can express which attributes are attached to node types. Such basic requirements are common to any graph storage. To allow for concise schema definitions, the DDL should allow properties common to more than one node type to be specified only once in, for instance, an abstract node type declaration. All node types sharing this property should then be declared in such a way that they *inherit* the property from the abstract type. As an example, consider nodes of types `Function` and `Procedure` as used in Figure 3.1. They have common properties such as outgoing aggregation edges to nodes of types `OpName`, `ParamList` and `Comment`. It would be appropriate to define these properties only once in some abstract node type `Operation`. Then `Function` and `Procedure` node types should inherit the common properties from `Operation`.

In practice, the data definition language should be tailored towards syntax graphs that the DBSE is used to store. Common structures in syntax graphs are multi-valued aggregations or references such as lists, sets and dictionaries of nodes. The data definition language should, therefore, offer the means to express these *multi-valued edges* as conveniently as possible.

They often not only contain nodes of one type, but of a number of different types. As examples, consider data flow diagrams that consist of a set of processes, terminators, stores and flows [dM78]. The DDL should, therefore, facilitate the definition of this kind of heterogeneous structures. Finally, structures in abstract syntax graphs may be nested. As examples consider data flow diagrams where a process can be refined by another data flow diagram or procedure declarations in Modula-2 that can contain nested procedure declarations. To anticipate these structures, the DDL of the DBSE must be able to express recursive structures.

As argued previously, changes to the internal syntax graph should become incrementally persistent. Therefore, edit operations performed by tools on documents have to be implemented in terms of operations modifying the internal syntax graph. These operations should be established as part of the DBSE schema mainly for two reasons:

Encapsulation: The structure definition of the project-graph should be encapsulated with operations which preserve the graph's integrity. They then provide a well-defined interface for accessing and modifying the graph. In order to enforce usage of this interface, the operations must become part of the DBSE schema.

Performance: Executing graph accessing and modifying operations within the DBSE is more efficient than executing similar operations within tools. In the latter case a significant number of nodes and edges need to be transferred from the DBSE to tools via some network communication facility, which is rather expensive in time.

To establish graph-modifying operations as part of the DBSE schema, the *data manipulation language* DML must be powerful enough to express them. This means in particular, that the DBSE's DML must be capable of expressing the creation and deletion of nodes and edges as well as the assignment of attribute values. Moreover, the DML must be computationally complete, as alternatives and iterations are needed in graph-modifying operations for graph traversal purposes.

The DBSE should provide predefined operators to access and modify multi-valued aggregations or references such as lists, sets and dictionaries. These operators should facilitate enumeration of all nodes of these structures. Given the identity, a property or a position, there should be operators to search for nodes contained in such structures. Finally, there should be operators to update these structures through the insertion or deletion of a node.

Abstract syntax graph definitions for single document types have already become rather complex. If we consider C++ class definitions, for instance, about 80 node types have to be defined with their aggregation and reference edges because the subset of the C++ grammar for class definitions has as many non-terminal and terminal symbols. In order to cope with the complexity of the schema definition for the whole PSDE, which may contain several document types, the DBSE's DDL must offer structuring mechanisms for schemas. Then the overall conceptual schema of a PSDE can be defined in fairly independent component schemas – one for each tool. As these schema components must rely on definitions of other schema components in order to implement inter-document consistency constraints, importing and exporting schema definitions between components must be supported by the DDL.

3.3 Views

In PSDEs, there are often instances of different document types that provide different *views* of the same aspect of a software system. Most information contained in documents of early phases (such as architectural or module interface design) is included in documents produced in later phases (such as the module implementation). Hence, documents of different types often contain redundant information. The module interface design in Figure 2.1 on Page 5, for instance, defines signatures of operations. The C implementation contains each of these operations with the same names, and parameter lists matching the signature of the design. If design and implementation of the module are considered as distinct documents, this leads to a high number of inter-document consistency constraints. In the document representation, this leads to redundant nodes in many abstract syntax graphs that must be related by inter-document reference edges.

Eliminating this duplication by sharing the aggregation subtrees concerned has the following advantages:

- storage of the schema and corresponding data requires less space,
- inter-document consistency preservation especially across document boundaries, is automatically achieved, and
- the conceptual schema is simplified.

Such sharing cannot be contemplated, of course, if automatic inter-document consistency preservation between documents is inappropriate.

If subtree sharing is to be used, tools must use the same conceptual schema. In order to maintain appropriate separation of tool concerns and so allow separate tool development and maintenance, the DBSE must provide a view mechanism like that offered in many relational database systems.

The view mechanism must allow for view definitions consisting of *virtual node and edge types*. A virtual node or edge type declaration is based on a (real) node or edge type declaration given in a conceptual schema. The set of all virtual node and edge type declarations of a view define structure and behaviour of a class of *virtual abstract syntax graphs*.

Instances of these virtual node and edge types are virtual as well. They are called *virtual nodes* and *virtual edges*, respectively. They do not persist in the database, but are derived by the view mechanism according to the virtual node and edge type type definitions from nodes and edges stored in the database. These stored nodes and edges have been instantiated from real node and edge types defined in the conceptual schema. Therefore, we call them *real nodes* and *real edges* respectively. Accordingly, the abstract syntax graph they form is called *real abstract syntax graph*.

As an example consider Figure 3.2. It depicts an excerpt of the abstract syntax graph from Figure 3.1 with two virtual abstract syntax graphs defined on top of it. The two virtual syntax graphs represent excerpts of a module implementation graph and a module interface graph. Dashed lines indicate the relationship between virtual nodes and the real nodes they were derived from.

The particular functionality required from a view mechanism for a DBSE is concerned with how virtual node and edge types are defined based on real node and edge types. Not all node

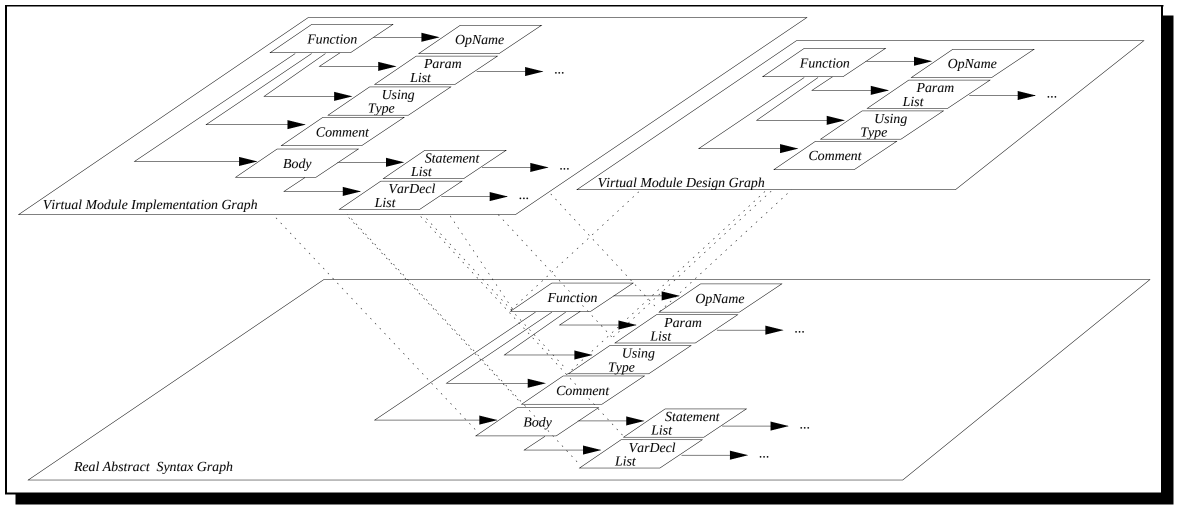

Function
OpName
Param List
Using Type
Comment
Body
Statement List
VarDecl List
Virtual Module Implementation Graph
Virtual Module Design Graph
Real Abstract Syntax Graph

Figure 3.2: Two Views of an Abstract Syntax Graph

and edge types defined in a conceptual schema need to be accessed in a view defined on top of the schema. In the above example, it is not necessary for a view defining a virtual module interface graph to include variable declaration lists and statement lists. Therefore, the view mechanism must support the hiding of node and edge types defined in the conceptual schema from one or the other view.

Even if a node type must be visible in a view, it may be appropriate to hide parts of its declaration. Attributes, operations or edges starting from a node may only need to be visible in some views while they are hidden in others. In the example above, function nodes have to be visible in both views. As the body of a function need not be seen in the interface view, the edge leading from a function node to a body should be hidden from the module interface view. Moreover, it may be reasonable to build the interface and implementation tool in a such way that changes to the interface are made in the module interface document only and inhibited in the implementation. Therefore, operations for changing the signature of a function may be hidden in the implementation view while they are available in the design view.

Some operations may be specific to only one view. Then it would be inappropriate to define them in the conceptual schema, for they would not be shared by different views and have to be hidden in all views but one. We, therefore, require that operations for accessing or modifying virtual nodes or edges can be added to a view. Operations defined in a conceptual schema can then also be redefined in a view. Then the operation from the conceptual schema would be hidden first. After that its redefinition would be added to the view.

Views are used by tools like editors that have to modify virtual abstract syntax graphs. Therefore, views must be updatable, i.e. updates that tools perform on virtual nodes and edges must transparently migrate into the database the view is built on. To make this updatability possible we cannot have virtual node or edge types added to a view, but not defined in the conceptual schema. If the node or edge type were defined on the relevant view only, which would be attractive for the same reason as adding operations to a view would be, the view could not be updated because the DBSE does not know of any real node or edge types upon which to store, for example, new virtual nodes or edges.

3.4 Schema Updates

In Subsection 2.2.1, we noted that existence and handling of inter-document consistency constraints depends on the kind of software process used. It has been widely recognised, that software processes cannot be fully anticipated, but may have to be changed "on the fly" [MS91, BFG93a].

If such a change results in the introduction of a new document type, the schema of the PSDE needs to be *updated* with subgraph definitions for this new document type. The DBSE must, therefore, enable new types of nodes and edges that are to be included in the project-wide ASG to be defined. Existing nodes and edges must not be affected by this kind of schema update. Consider as an example the introduction of a new document type `ReviewReport`. This new type may be introduced due to a process change requiring that each module interface definition be reviewed before implementation can start. If this report consists of sections for each reviewed module and subsections for each operation defined in a module, new node types for the review report, its sections and subsections will have to be defined together with the aggregation edges connecting them.

The introduction of a new document type may also require changes to existing document types. If a new document type is introduced, inter-document consistency constraints of existing document types may also have to be changed. Thus, in the structure definition of existing subgraphs, additional reference edge types to the subgraph of the newly introduced document type have to be added. In the above example, for instance, new inter-document reference edges between nodes in a module interface graph and sections or subsections in a review report have to be defined.

If document types are changed in this way, there is a need to update already existing documents of these types in such a way that they conform to the new definition. This database update cannot be done automatically by the DBSE, but needs to be specified by the PSDE builder. For newly introduced node and edge types, nodes and edges have to be created according to the current state of the database. Then new inter-document reference edges may have to be created between existing nodes and new nodes. In the above example, a review section or subsection node must be created for each module and each operation. Then these new nodes must be related by inter-document reference edges to module and operation nodes in the module interface subgraphs.

Once specified, the DBSE must execute these kinds of database updates. This could be done at one point in time with an eager strategy that updates all existing documents of the changed type according to the update specification. This, however, requires a PSDE shutdown which is not possible generally. Therefore, the DBSE should also offer a lazy update strategy in which the DBSE executes the document update whenever a document with an out-dated structure is accessed. As a matter of fact, the DBSE then has to maintain a history of schema updates which will slow down the overall PSDE performance.

3.5 Version Management

Given the overall representation of a project as defined in Section 3.1, the DBSE must support management of versions of those subgraphs that represent versionable documents. This requires that the DBSE offers a concept for defining subgraphs. Subgraphs may be defined statically in the schema or dynamically at run-time. For a static definition, the DBSE's DDL must provide language primitives for distinguishing aggregation edges from reference edges. A dynamic definition requires a predefined type that can be used for implementing subgraphs. This type must offer operations for including and removing nodes and edges to or from the subgraph. Static definition is more efficient in terms of space and time since for any existing project-wide graph no further actions are required to identify the subgraphs that implement documents. Dynamic definition of subgraphs is more flexible, since arbitrary components of the project-wide abstract syntax graph can be identified as a subgraph.

The particular functionality required for version management of subgraphs is that, during creation of a root node of a subgraph, an initial version or *root version* must be created as well. Versions must have a name called *version label* for accessing them. Therefore, the DBSE must offer facilities for defining version labels. The DBSE must then support derivation of a new version of a subgraph from a particular version of the subgraph. It must maintain the version history between different versions of a subgraph, i.e. the DBSE must keep track of the predecessor/successor relationship between different subgraph versions. It must facilitate navigation along predecessor/successor relationships. Moreover, the DBSE must support a tool session in selecting a *current version* of a subgraph so that all its nodes and edges are seen in the state of that version. In addition, the system must support the definition of default versions of subgraphs which are persistently stored and used to determine accesses to nodes and edges when no current version has been explicitly selected.

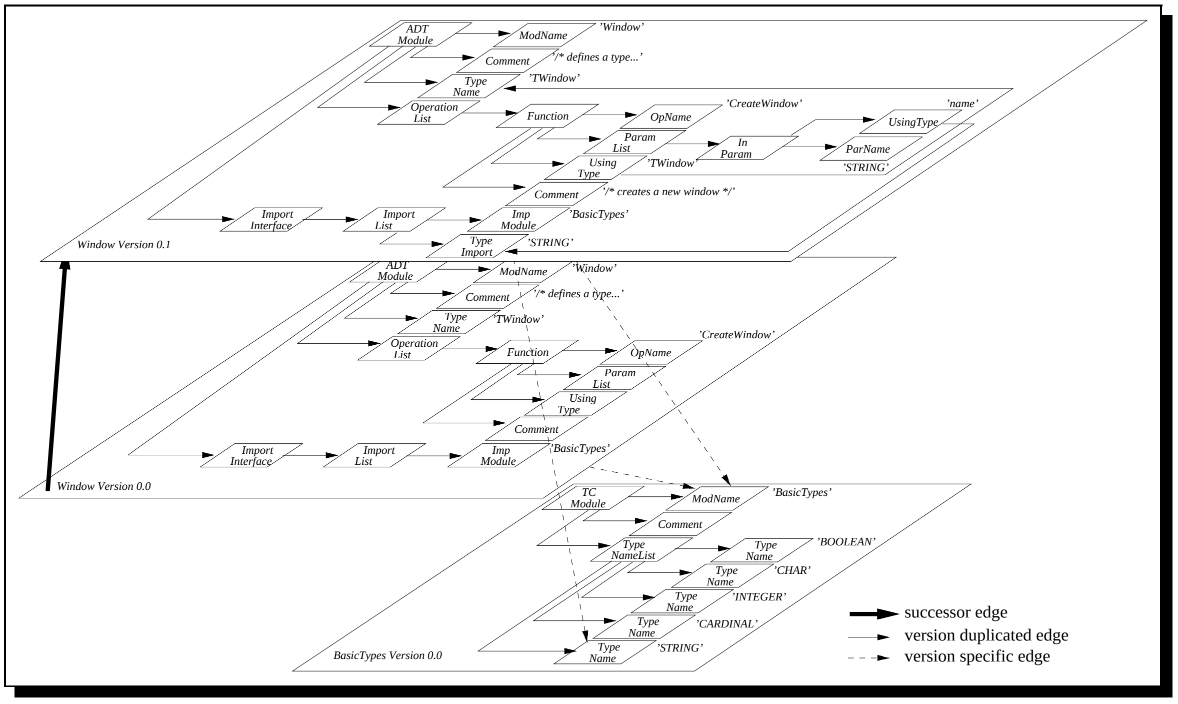


Figure 3.3: Versions of Subgraphs of the Project-Wide Abstract Syntax Graph

Figure 3.3 depicts earlier subgraph versions of the module interface graphs displayed in Figure 3.1. The two subgraphs that represent module `Window` and module `BasicTypes` are under version control. Currently two versions of `Window` exist. They are labelled `Version 0.0` and `Version 0.1`. There is a *successor edge* between these two subgraph versions indicating that `Version 0.1` is a successor of `Version 0.0`.

After two subgraph versions have been derived from the same predecessor, they may have to be merged. We note that this merging cannot be done completely by the DBSE, but requires additional actions by a tool. In case of conflicting changes in the two subgraph versions where the same nodes have been modified, the DBSE cannot decide which version of the node to take. Instead, this has to be determined by the tool user. The basic mechanism a tool builder requires from a DBSE is that it can merge two subgraph versions in the version derivation graph and provides a primitive to find the difference in terms of nodes and edges between different versions of the same subgraph.

We also note that the DBSE need not offer the means for freezing versions. During configuration management, changes will be required to subgraph versions, although the corresponding document is considered frozen. These changes might have to create or delete reference edges that keep track of a new semantic relationship between document versions. Changes of attribute values might be required during inter-version consistency checks. These changes are impossible if the subgraph is frozen. As a consequence, the DBSE must not impose the same behaviour on subgraph versions that we had required for document versions, but it must enable changes to be made to subgraphs that have successor versions. In fact, the mechanism that freezes documents cannot be implemented in the database, but must be implemented within tools.

Within versioned subgraphs the DBSE must resolve between fully lazy and fully eager *duplication strategies* for nodes and aggregation edges of the subgraph concerned. Fully lazy duplication gives maximum sharing of components, and hence minimum storage usage, but complicates the update process during user edits. Fully eager duplication avoids all such complications, but implies maximum storage usage.

The DBSE must also decide between alternative strategies for handling both intra- and inter-document relationships (or reference edges). Intra-document reference edges will normally be treated (like aggregation edges) as *version-duplicated* (i.e., a new edge is automatically created for each version created). Inter-document reference edges may be seen as *version-specific* (and hence not duplicated when new versions are created). Version-specific edges then represent relations between document versions within a configuration.

In the example given in Figure 3.3, aggregation and intra-document reference edges are considered as version-duplicated. Therefore, the edges that connect using types with the nodes where the types are declared are drawn as a solid shape. Inter-document reference edges are considered as version-specific, since they form particular configurations of subgraph versions.

3.6 Transactions

In Subsection 2.3.2 we required integrity preservation and immediate persistence for command execution in tools. To implement this, we require the means from the DBSE to group a set of operations that access or modify the abstract syntax graph to one unit. We call these units *transactions* in the following. A transaction is started by a tool before it begins to execute a command. It, therefore, issues a start transaction request to the DBSE. Then the tool performs the operations necessary to execute the command. When finished it completes the transaction by issuing a commit request to the DBSE. If it detects an intolerable error, the tool may also explicitly request a transaction abort. This undoes the effect of the current transaction and recovers each modified node and edge to the state it had when the transaction was started.

In general, we require transactions to have ACID properties as suggested in [Gra78]. Due to the atomicity property, transactions are either performed completely or not at all. Due to the durability property, the effect of a completed transaction in every case persists in the database. The consistency[2] preservation property ensures that after completion of a transaction, the abstract syntax graph is in an integer state and tools can continue using it. Finally, the isolation property ensures that the effect of a transaction is independent of other concurrent transactions. Hence the isolation property of transactions contributes to the multi-user support required in Subsection 2.2.3. To achieve isolation of transactions, the DBSE has to apply a concurrency control protocol. The most common protocol is two-phase locking [BHG87]. During this protocol nodes and edges are locked by the DBSE as soon as they are accessed by tools, i.e. locking should be transparent to tools. During commit or abort all locks of the transaction are released at once.

If a node or edge is accessed without being modified, the DBSE only has to lock it in read mode. If it is modified, the DBSE has to lock it in write mode. Read locks are *compatible* with each other, i.e. the DBSE can grant arbitrarily many read locks for a node or edge. Unlike read-locks, write locks are neither compatible with read locks nor with write locks. This implies that if a transaction obtains any lock on a node or edge, another concurrent transaction cannot acquire a write lock on the same node or edge.

This strict concurrency control protocol is appropriate in syntax-directed tools for the following reasons. Command execution in tools only lasts for a very short period of time, say less than some hundred milliseconds. During this time, only a few nodes are modified. In addition, it is unusual for two or more users to edit the same document concurrently. Therefore, the only node accesses that could possibly result in concurrency control conflicts are node accesses along inter-document reference edges. They effectively cause a concurrency control conflict only if another tool concurrently accesses the remote node in an incompatible mode. In these rare cases the tool can await completion of the concurrent transaction before it gets the lock granted. Users will hardly ever notice these short delays.

In even rarer cases, it can happen that concurrent command execution results in a *deadlock*, because two-phase locking ensures serialisability, but is not deadlock-free. Consider that a transaction executing one command has locked a set of nodes and edges while another transaction has locked another set of nodes and edges. If both transactions now try to lock a node or edge the other transaction has already locked in an incompatible mode, a deadlock will occur. We require the DBSE to detect these deadlocks and inform the tools about them. The tools can then ask their users to abort the command execution and retry it later. If one user aborts, the deadlock will be resolved without losing significant effort. The tool will recover to the state it was in before the aborted transaction started.

In many cases a PSDE builder knows that concurrency control conflicts cannot occur. If we consider an editor for a programming language like Modula-2, for instance, and assume that only one user is responsible for one module, then concurrency control conflicts cannot occur during edit operations on statements, comments, parameter names and local variable declarations. If two-phase locking is used for these operations, the DBSE will use unnecessary effort checking for concurrency control conflicts. If the DBSE offers a weaker transaction concept which we call *activity* in the following, the transaction throughput can be increased significantly. These activities need not guarantee isolation of the group of operations they

[2]This notion of consistency is at a lower level and must not be confused with static semantics or inter-document consistency. For the DBSE any syntax graph is consistent, even though it might represent a completely inconsistent document, as long as the graph conforms to the schema definition.

execute, but only ensure atomicity and durability. These two properties are needed to preserve the integrity of the project-wide abstract syntax graph against any kind of hardware or software failures. During a tool session, both transactions and activities may have to be used in an arbitrary sequence. As an example, consider that the user of the Modula-2 editor wants to change a procedure name after he or she has changed a comment. Changing the procedure name must be performed as a transaction, since other concurrent users may, at the same time, create an import relationship in one of their modules importing the changed operation.

We note that we do not require any *advanced transactions* such as nested transactions [Mos85], design transactions [KSUW85, Kat84], group transactions [Kel89], CAD transactions [BKK85] or split/join transactions [KHPW90] from the DBSE. All these advanced transactions assume patterns of cooperation between users of a PSDE, which do not apply in general. Therefore, these advanced transactions cannot be built into the DBSE, but should rather be defined explicitly by the process model and be implemented by the process engine. The DBSE and the tools only have to offer the basic mechanisms to enable the process engine to do so. As argued in [Wol94], ACID transactions, together with the means for versioning documents, are sufficient for this purpose.

Note also that version management with lazy node duplication interferes with concurrency control. Assume that a node is shared by two versions `V1` and `V2`. If one transaction modifies the node in version `V1`, while some other transaction accesses the node in version `V2`, a concurrency control conflict arises. This conflict is not serious and should not cause one of the transactions to be blocked or even aborted. Instead, the shared object is split into two versions and the two concurrent transactions continue to be executed. This, of course, requires concurrency control management to be aware of the versioning strategy. Thus, the lazy duplication strategy for version control must be implemented inside the DBSE together with the basic concurrency control. It is impossible to implement it on top of a DBSE that does not support versions of subgraphs.

3.7 Performance

The performance that a tool builder requires from a DBSE is determined by the performance a user requires from a tool. To discuss this thoroughly, we have to consider the command execution cycle of tools. It is depicted as a state transition diagram in Figure 3.4.

The diagram displays states of a tool as circles, and transitions between them as arrows. Transitions in this diagram reflect the operations the tool executes. After start-up, the tool is in the state where no increment is selected. When the user moves his or her pointing device to an increment and pushes a button in order to select the increment, the tool has to respond by high-lighting the respective part of the document at the user interface. A response time acceptable for this would be less than half a second. This, however, is non-critical from the DBSE performance point of view, since the abstract syntax graph stored in the DBSE need not be accessed at all in order to perform this operation.

To perform command selection from a pop-up menu, the database is needed only to validate the preconditions of menu entries during menu computation. Therefore, only some navigation along edges and accesses to nodes are needed that again should be performed in less than half a second.

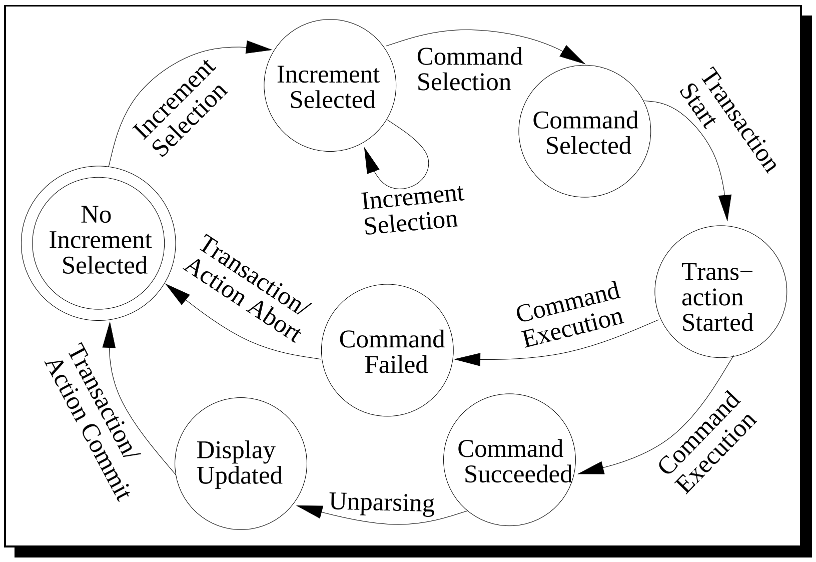


Figure 3.4: Command Execution Cycle of Syntax-Directed Tools

After a user has selected a particular command, a DBSE action or transaction respectively must be started depending on whether the successive operations could cause a concurrency conflict or not. After that, operations defined in the schema for accessing and modifying the abstract syntax graph are executed. The number of edges that need to be traversed or created and the number of nodes that must be accessed or created range from a few, when inserting a template, to some thousand, when parsing a large text after free textual input. A command execution may fail or succeed. If it fails, the transaction/action must be aborted. If it succeeds, all nodes of the abstract syntax graph, which were changed during the transaction/action execution must be *unparsed*, i.e. redisplayed at the user interface, and their presentations must be associated with the node identifiers. Again these may be a few, up to some thousand. Then the database must perform a transaction/action commit. When operating tools in the structure-oriented mode, users are hardly willing to accept a gap of more than one second between transaction/action start and commit or abort. The acceptable response time for *parsing* an increment after free textual input may increase with the size of the input text.

3.8 Distribution

Most PSDEs run on workstations connected via a local area network in which each user works at one workstation. In order to benefit from the computational power of modern workstations and not to produce a bottleneck of the PSDE on a centralised host, tools of a PSDE should be executed on the user's workstation. Then, however, a need for *distribution* of documents or at least *distributed access* to documents and distributed tool communication arises.

When a DBSE is used to store the project-wide abstract syntax graph in a database, a database monitor has to implement the concurrency control protocol. Therefore, tools cannot directly access nodes and edges, but have to communicate their access requests to the database monitor and await the monitor handling their requests. Both tools and monitor require a significant number of computations to fulfil their duties. To achieve a PSDE performance acceptable for a number of concurrent users, execution of database monitors and tool processes should be distributed in order to balance the load over several machines.

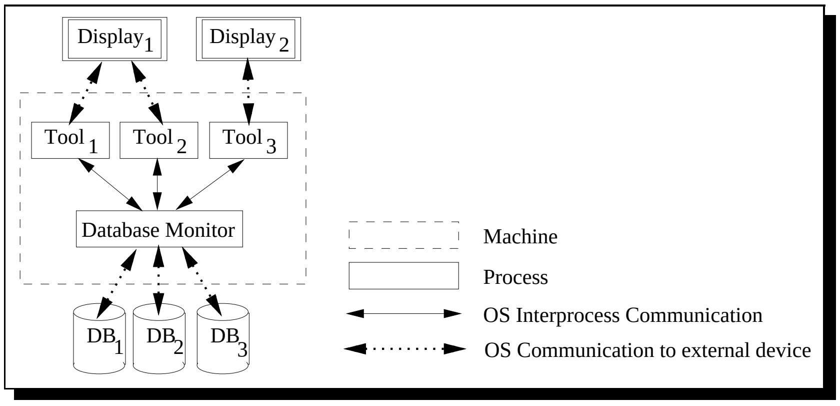


Figure 3.5: Centralised Architecture

Figure 3.5 depicts a model where this is not the case. Database monitor and tool processes are executed on the same host. The monitor and tools use facilities provided by the operating system, such as shared memory, message queues or semaphore sets for their inter-process communication. The advantage of this model is that inter-process communication facilities provided by the operating system perform fast. The severe disadvantage is that all tools must be executed on the same host. In fact, they have to share the host's resources such as virtual memory or CPU-time. The performance of each tool will decrease as the number of concurrent tools increases. Hence the host will become a performance bottleneck as soon as a certain number of concurrent tools is reached.

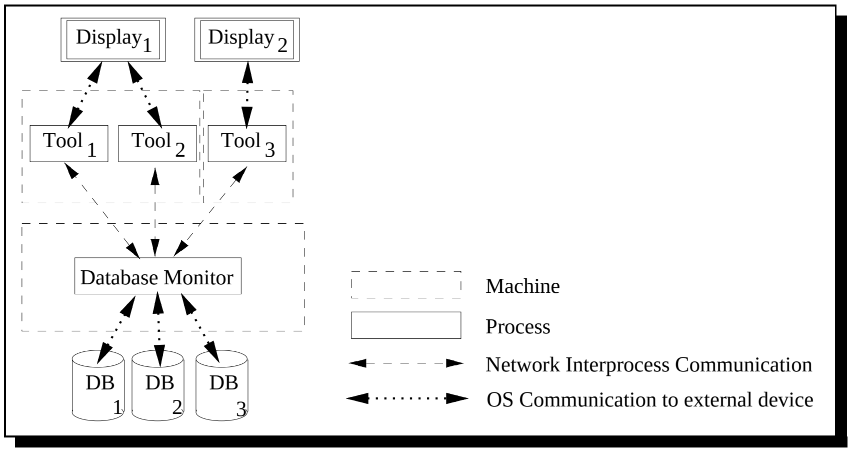


Figure 3.6: Client/Server Architecture

To avoid this situation, the obvious strategy is to use a *client/server architecture* and have tools running on possibly different client hosts and the database monitor on a server host. This model is depicted in Figure 3.6. As a consequence, communication between database monitor and tools can no longer be implemented by operating system primitives, but network communication protocols such as sockets or remote procedure calls must be used. The advantage of this is that the load directly caused by tool execution is removed from the server host.

In Section 3.2 we required the DBSE's schema to define all syntax graph accesses and modifications. There are two options on where to execute these operations. The first option is a

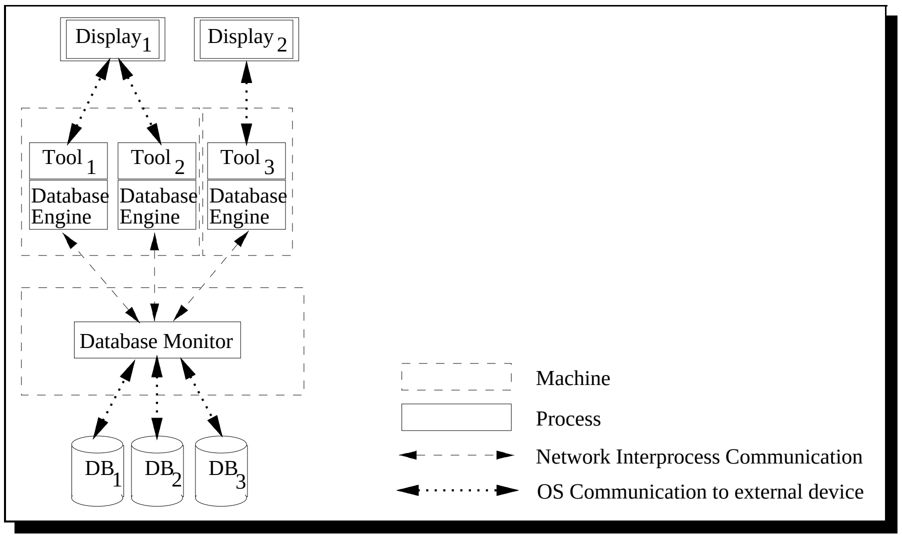


Figure 3.7: Client-based Client/Server Architecture

server-based client/server architecture, where the database monitor would execute access and modification operations, receive operation parameters from tools and communicate operation results back to tools. This could still result in a performance bottleneck on the server host, for a great deal of the functionality of tools is implemented by graph access and modification operations. The other option which remedies this problem is sketched in Figure 3.7. In this *client-based client/server architecture* the part of the database system (called *database engine* in the following) that executes operations defined in the schema is linked with the tool. The engine is, therefore, executed in the same operating system process as the tool. Thus communication of the tool with the database engine is achieved by procedure calls that perform fairly efficiently. The duties that remain with the database monitor are concurrency control and elementary operations on raw data called *pages* that it has to communicate via some network communication protocol to processes running the database engine. Hence the server hosting the database monitor is further relieved from load compared with server-based client/server architectures.

Database monitors access pages either directly on raw disk devices or indirectly via the operating system's file-system. If the latter is the case, the host running the database monitor may be further relieved from load by storing raw data on disks that are connected to other hosts. This is sketched in Figure 3.8. Then the database monitor must use network file-system facilities. The server running the database monitor becomes a client of some other file servers. We, therefore, call this architecture *multi-level client/server architecture*.

In the previous model, any access a tool performs to a node or edge of the project-wide abstract syntax graph must be processed by the database monitor. Therefore, even multi-level client/server architectures cannot be used in arbitrary large projects. The *distributed database architecture* sketched in Figure 3.9 no longer assumes only one monitor process and allows for multiple database monitors. Each of these monitors controls a set of local databases. Tools are still served by one monitor. Accesses of a tool to a remote database must be transferred from the local monitor to the respective remote monitor and be handled there.

In practice, hybrid architectures that combine different distribution paradigms may be used. Database monitors of distributed database systems, for instance, can store their raw data on

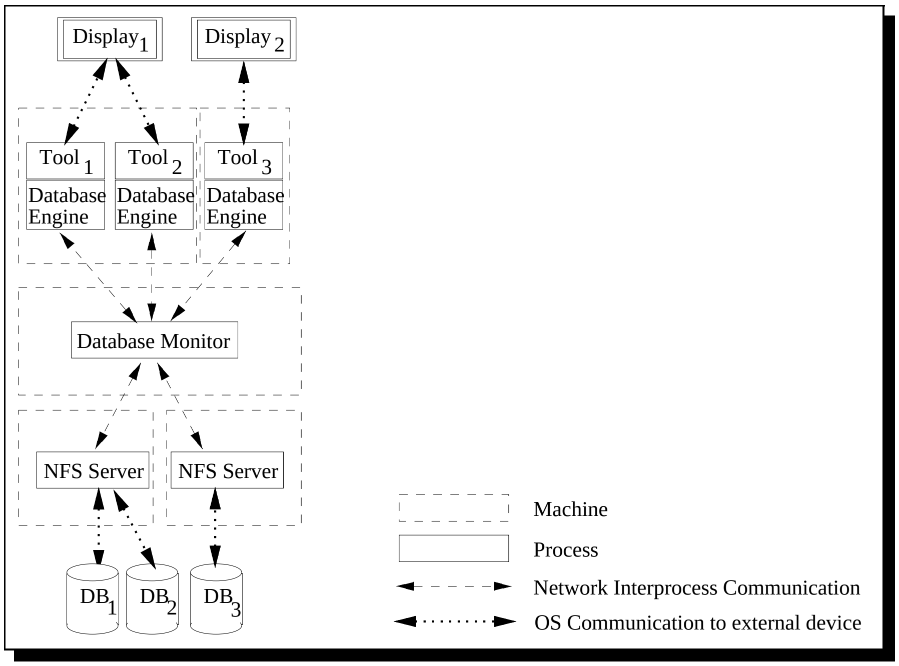


Figure 3.8: Multi-level Client/Server Architecture

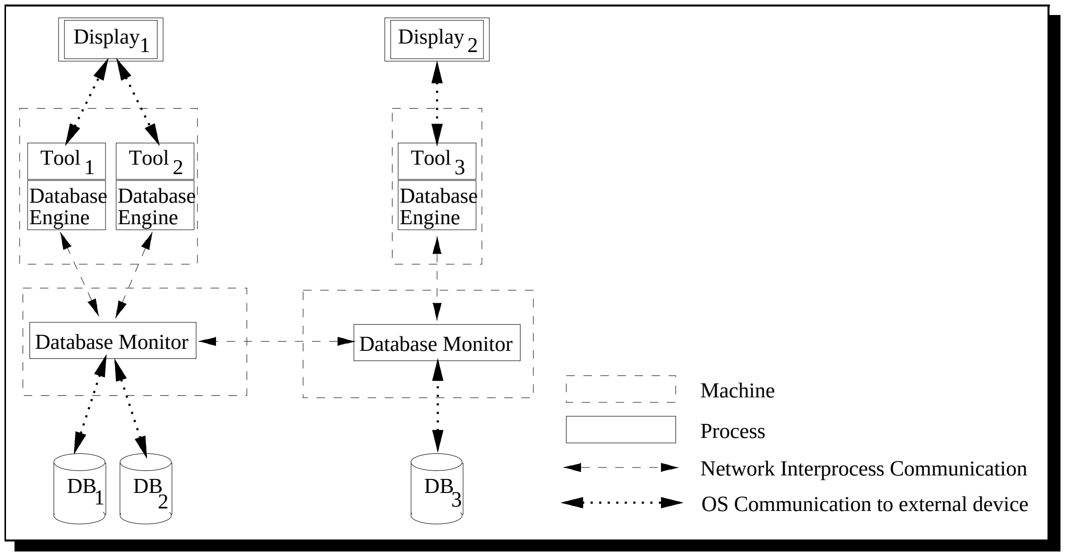


Figure 3.9: Distributed Database Architecture

other file servers. Similarly, the database engine in distributed database systems may reside with the database monitor or be linked with tools.

In short, we require from the DBSE at least distribution support based on a client-based client/server architecture or a multi-level client/server architecture. For large projects, acceptable performance will not be achieved without a distributed database architecture. The disadvantage of using a distributed database architecture is that it will incur additional administration overheads as several database monitors must be controlled and data distribution be administered. Moreover, the two-phase locking protocol will no longer be sufficient for concurrency control. Instead a two-phase commit protocol will be required.

3.9 Administration

Data stored in the DBSE is probably the most important resource of a software house and must, therefore, be securely protected against hardware failures such as disk crashes. There are several options to achieve data security. The obvious solution is to dump project-wide ASGs on backup media, like tapes or optical disks. Therefore, the DBSE must offer *backup* facilities. As the size of a project database may be too large to be completely backed-up daily, the DBSE must allow for incremental backups. For daily incremental backups, the PSDE administrator should not have to shut down the database since software production must not be disturbed by these administration procedures. In the event of a disk crash, the PSDE administrator then needs a facility for restoring the contents of a database from the backup-media. The second option to achieve security against hardware failures is replicated storage. Therefore, the DBSE must offer facilities to the PSDE administrator for defining one or more replicas of a master database. It must then transparently perform any operation the PSDE executes on the master database on all replicas as well. In the event of a disk crash a replica may be taken as the new master database and a new replica can be created. Replication may additionally slow down DBSE performance, but has the advantage that in the event of a crash no data is lost at all and PSDE operation can continue without too long an interruption. In the first approach, DBSE performance would be decreased only slightly during backup. It is not as safe as replication, as all changes made after the last backup will be lost. Moreover, it could result in a significant PSDE stoppage: To restore a database from one full backup and a set of incremental backups the full backup must be restored first and then each incremental backup must be added successively.

Additional administration effort is required for a DBSE with a distributed database architecture. The administrator has to decide which machine should host tool processes for which users. The administrator must then designate machines that host database monitors. Next it must be determined which monitors shall serve which tool processes, after which the databases must be set up in a way that communication between database monitors is minimised. During PSDE operation, network traffic between database monitors must be monitored in order to find out if there are any subgraphs located in the wrong database. If these are found, the PSDE administrator must transfer subgraphs to different databases.

3.10 Interfaces

The DBSE is used by different kinds of users, namely tool builders, the PSDE administrator and tool users, in different ways. The PSDE administrator needs a user interface which allows him or her to browse through an abstract syntax graph from an administrative point of view. A *backup tool* is required to perform incremental or full backups.

In order to enable a tool builder to install, view and test a schema, the DBSE must offer a set of tools. To install the schema, the DBSE must provide a *compiler* for the DDL and DML. As the schema of a PSDE will become rather complex the compiler should work incrementally. To review the current state of the schema, a tool builder needs a *schema browser*. We require operations that access and modify the project-wide abstract syntax graph to be part of the schema. After these operations have been established, the tool builder will have to test and debug the operations. Therefore, the DBSE must provide some *schema debugger* that enables execution of operations, inspection of the state of the abstract syntax graph with a graphical *object browser*, computation of execution traces, inspection of the call stack and handling of breakpoints.

We cannot assume that tools can be completely implemented within the database schema. The user interface required from tools, for instance, requires the use of powerful user interface management systems. Although most database systems provide some UIMS support, for instance to define forms, the functionality offered must be considered as too weak for the construction of syntax-directed tools. For this reason, a tool architecture must have a subsystem which arranges for the presentation of documents and menus. This subsystem must be built on top of some UIMS and be written in a standard programming language.

Then, however, the need arises to access the schema's data structures and operations from a programming language. This is needed firstly to present documents, which are stored as subgraphs of the project-wide abstract syntax graph within the DBSE, at the user interface. To compute the document representation, the presentation subsystem must be able to navigate through the abstract syntax graph. Secondly operations defined in the schema need to be executed as soon as the user has chosen to execute a command. To actually execute a command, operations defined in the DBSE schema must be performed. The DBSE must, therefore, provide a *programming interface* to achieve this binding with one or more programming languages.

Moreover, the presentation subsystem must arrange for an association of increments at the user interface with nodes in the abstract syntax graph. Only then can it manage to invoke the operations defined in the schema for the currently selected increment. This association can best be achieved if the DBSE offers *unique node identifiers* in the programming language, representing nodes of the project-wide abstract syntax graph. They can then be used to associate nodes with increment presentations.

3.11 Summary

We require the following functionality from a database system in order to use it as a DBSE. The system should support the definition of structure and operations for project-wide abstract syntax graphs in a set of schemas. It should offer a view mechanism to allow different kinds of tools to share subgraphs without losing the ability to develop tools separately. As software processes evolve over time, DBSE schemas must be changed, even though project-wide abstract syntax graphs may already have been instantiated. Therefore, the DBSE should update existing abstract syntax graphs according to the schema change as well. To keep track of the development history of documents the DBSE should be able to maintain different versions of subgraphs of the project-wide abstract syntax graph. To preserve the integrity of project-wide abstract syntax graphs and allow for concurrent access of multiple users, the DBSE must offer ACID transactions. The performance of the DBSE should be so fast that tool commands can be executed in less than 1000 milliseconds. The DBSE should offer the means for distribution with a client/server architecture in order to allow for a number of concurrent users without giving up the above performance requirements. Beyond a client/server architecture, databases may have to be distributed in order to support even larger numbers of concurrent users. For purposes of data security and performance tuning, the DBSE should offer functionality to support a PSDE administrator. Finally, the DBSE should offer a set of different user interfaces to a PSDE builder and a PSDE administrator as well as a programming interface.

Chapter 4

Selecting a Database System

In this chapter, we shall review a number of classes of database systems, namely relational DBSs (RDBS), structurally object-oriented DBSs (SODBS) and object DBSs (ODBS), with reference to the requirements delineated in the previous chapter. The aim of this chapter is thus to select the most appropriate class and then the most appropriate system of this class as the further basis for tool construction.

Assessment with respect to functional requirements on database systems can be done analytically using the requirements presented in the previous chapter. Therefore, we shall present the distinguishing features of database systems of the above classes and discuss how well they meet the functional requirements. Evaluation of database system performance is more difficult, for it requires systematic, practical experimentation [DEL92]. This is why we start with an excursion to the problem of database performance evaluation in Section 4.1. We present our methodology for developing application-specific benchmarks for performance evaluation purposes and then develop the Merlin Benchmark using this methodology. The following sections assess the different classes of database systems. To assess their performances we implement the Merlin Benchmark with archetypical representatives of the different classes and discuss the impact of the benchmark results on the performance of tools. In Section 4.2, we assess relational database systems. In Section 4.3, the suitability of structurally object-oriented database systems is discussed and in Section 4.4 we explore object database systems.

4.1 Performance Evaluation of Database Systems

The general strategy to evaluate the performance of a system a-posteriori, i.e. after the system has been constructed, is to define and execute *benchmarks*[1]. A benchmark defines a synthetic load that approximates the load of a set of real applications. Benchmarks can be classified into *atomic benchmarks*, which perform a single operation, or *complex benchmarks*, which include a set of distinguishable operations. Atomic benchmarks are, for instance, the Whetstone Benchmark [CW76] or the Dhrystone Benchmark [Wei84] for measuring operating system performance. Their result is given either by the execution time for their operation or the reciprocal value, i.e. how often the operation can be executed within a given amount of time. Examples of complex benchmarks are the Wisconsin [DeW91] Benchmark for relational

[1]Performance modelling and simulation, i.e. a-priori performance evaluation of a system that is under construction is outside the scope of this thesis since we do not want to build a database system.

databases or the Sun Database Benchmark [CS92]. Their result is a vector of execution times rather than a single value. Comparison of their results is not as simple as it is for atomic benchmarks since they do not necessarily reveal an order between the systems. On the other hand, complex benchmarks allow performance comparison of systems with respect to different criteria. As we want to compare database systems with respect to several concerns, we only consider complex benchmarks for our evaluation problem.

Measuring execution times is based on operating system primitives and can be given in different terms. For the area of database system performance evaluation, the most important execution time is the *elapsed real-time* of an operation. Only this time will indicate how long a user of a tool will have to wait for a database system to complete an operation. We are not interested in the *CPU time*, which is the time the CPU spent on the execution of an operation, or in the *system time*, which is the time the system spent waiting on external devices. Execution times in this thesis, therefore, indicate the real-time that elapses during an operation execution.

Database systems of the different classes differ significantly with respect to the provided functionality, the data model and the programming interface. Due to the lack of established standards in structurally object-oriented and object database systems, even systems of the same class differ significantly. This heterogeneity prevents the definition of a benchmark as a uniform piece of source code like the Whetstone, Dhrystone, Sun or Wisconsin Benchmarks. The benchmark definition, therefore, has to be defined on a more abstract level than source code. We, therefore, call these benchmarks *abstract benchmarks*. It will basically be a conceptual schema plus a number of operations based on the entities defined in the schema. Finally, one or more initial database states must be determined. The benchmark then has to be implemented for each different database system whose performance is to be evaluated.

4.1.1 Existing Abstract Database Benchmarks

Three abstract database benchmarks have been defined. The first one is the so called *Simple Benchmark* [DHS$^+$91, DEL92]. The second one is the *Hypermodel Benchmark* defined in [ABM$^+$90]. The recently suggested *OO7 Benchmark* [CDN93] has gained substantial acceptance for comparison of object database systems. We outline the definitions of each of these benchmarks in the next subsection to be able to explain why they cannot be used for evaluating database performance of software engineering applications.

To simplify the comparison, we use a common notation for describing the conceptual schemas of the three different benchmarks. We select the entity relationship notation suggested by [BPR88] since it can express inheritance and ordered relationships which are used in the benchmark. In this notation, a rectangle represents an entity[2]. A solid arrow between entities represents an aggregation relationship. Its semantics are that no object can exist without being related in an aggregation to an existing object, i.e. the aggregation relationship models the part-of/belongs-to relationship. Dotted lines model reference relationships. At the end of arrows or lines, black circles represent a many-end of a relationship, and white circles represent a one-end. A circle placed on a line declares the relationship to be ordered. Attributes of entities are described within the rectangle, whereas attributes of relationships are shown in a circle connected to the resp. arrow. A triangle on a line between entities indicates an inheritance relationship meaning that a sub-entity inherits all attributes from its super-entity and can participate in the same relationships as the super-entity.

[2]In the following, *entity* and *type* are used synonymously, whereas *object* denotes an instance of an entity.

4.1.1.1 The Simple Benchmark

The Simple Benchmark was defined in order to measure the performance of elementary OMS operations. The conceptual schema for this benchmark (as well as for the others in this thesis) is shown as an extended Entity-Relationship Model (EER-model) in Figure 4.1.

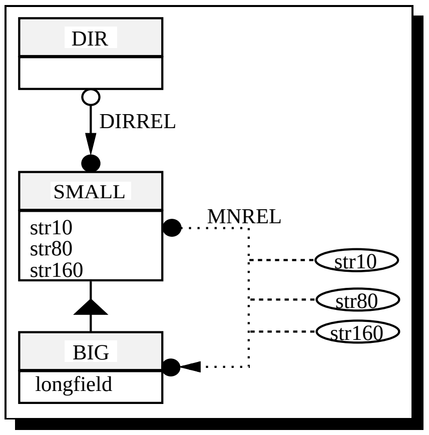


Figure 4.1: EER-Diagram for the Simple Benchmark

Objects of type `DIR` model a composite object, which is composed of a number of other objects. `DIRREL` models the composition relationship. Component objects can have different sizes. Small objects represent, for instance, syntax graph nodes that have three small attributes. Big objects have an additional long-field attribute that can be used to store long comments, for instance, or source code or even compiled object code. Apart from the composition relationship, the benchmark considers reference relationships and defines a relationship `MNREL` that models references between small and big objects. `MNREL`, in addition defines three small attributes.

The operations of the Simple Benchmark include creation and deletion of small and big objects, as well as creation and deletion of relationships between them. Furthermore operations on attributes of entities and relationships such as storing and retrieving strings of lengths 10, 80, and 160 bytes or long-fields of the lengths 10 and 128 KBytes are defined.

The benchmark requires that the operations access and modify a non-empty database. This avoids the possibility that objects accessed by the operations will reside only in database caches (i.e. in main memory). A realistic size of an initial database, created before performance measurements start, guarantees that operations have to access secondary storage (as usually happens in real applications). The Simple Benchmark requires the initial database to contain 3,000 objects of type `SMALL` and 400 objects of type `BIG`.

4.1.1.2 The Hypermodel Benchmark

The Hypermodel Benchmark differs from the Simple Benchmark in using more complex data structures and operations. The benchmark is a development for comparing databases with respect to hypertext applications.

The conceptual schema of the Hypermodel Benchmark is shown in Figure 4.2. It includes three different entities, namely `Node`, `TextNode`, and `FormNode`. `TextNode` and `FormNode` are subtypes of `Node`. `Nodes` represent sections of a hypertext, which are further structured.

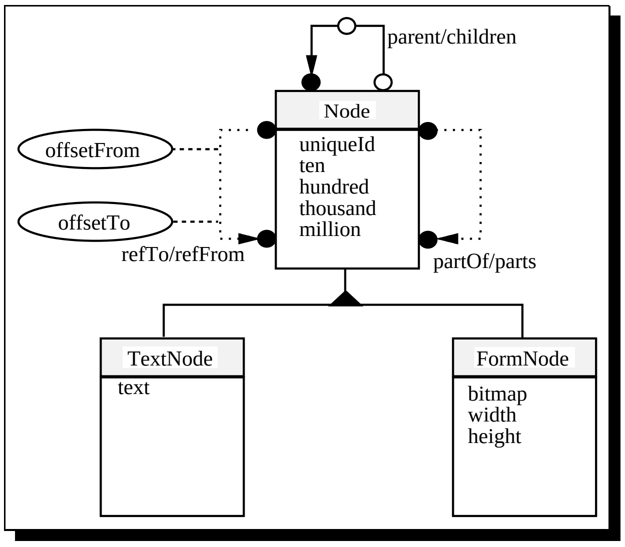


Figure 4.2: Conceptual Schema of the Hypermodel Benchmark

`TextNodes` represent an unstructured text and `FormNodes` represent a bitmap. Three relationships are defined, namely the `parent/children` relationship, the `partOf/parts` relationship and the `refTo/refFrom` relationship. The `parent/children` relationship is of cardinality 1:n, ordered and defines the aggregation structure between nodes. The m:n `partOf/parts` relationship models the section/subsection structure of a hypertext and the m:n relationship `refTo/refFrom` models arbitrary hypertext links. Each `Node` has a unique identifier and four attributes (`ten, hundred, thousand,` and `million`) for storing randomly selected numbers of a particular range. A `TextNode` contains an additional `text` attribute and a `FormNode` has three attributes. `Width` and `height` store the dimensions of a picture and a long-field attribute `bitmap` stores the picture itself. The `refTo/refFrom` relationship contains two attributes, `offsetFrom` and `offsetTo`, that store relative coordinates of hypertext links within `text` attributes.

The operations of the Hypermodel Benchmark include mainly retrieval operations such as queries for attributes with particular names or attribute values in particular ranges, lookups for node sets connected by the above mentioned relationship in normal or reverse order, and finally, operations performing a sequential scan and a transitive closure traversal following different relationships. The only update operations substitute words in the `text` attribute of a text-node and inverts a sub-rectangle within the bitmap attribute of a randomly selected `FormNode`. A detailed description of these operations is of no concern for this thesis.

The initial database contains a balanced tree of nodes and `father/children` relationships. The size of the initial database depends of the height of this tree. Each inner node is of type `Node` and has exactly five children. Each leaf node is either of type `FormNode` or `TextNode`. The `partOf/parts` relationship is created for each node by selecting one inner node of level k and relating it to five random nodes at level k+1. The `refTo/refFrom` relationship is created for each node to another random node. Nodes are numbered and the numbers are stored in the `uniqueId` attribute. `Ten, hundred, thousand,` and `million` are initialised by random numbers selected from the corresponding interval. Each attribute of objects of type `TextNode` is initialised with a text containing up to 100 words, each word having up to ten characters. A formnode consists of a random square bitmap with an edge length of up to 400 pixels.

4.1.1.3 The OO7 Benchmark

The OO7 Benchmark is intended as a yardstick for comparing the performances of ODBSs when they are used in complex engineering applications such as CAD, CAM or CASE. Therefore, the benchmark is more complex than the Simple or the Hypermodel Benchmark.

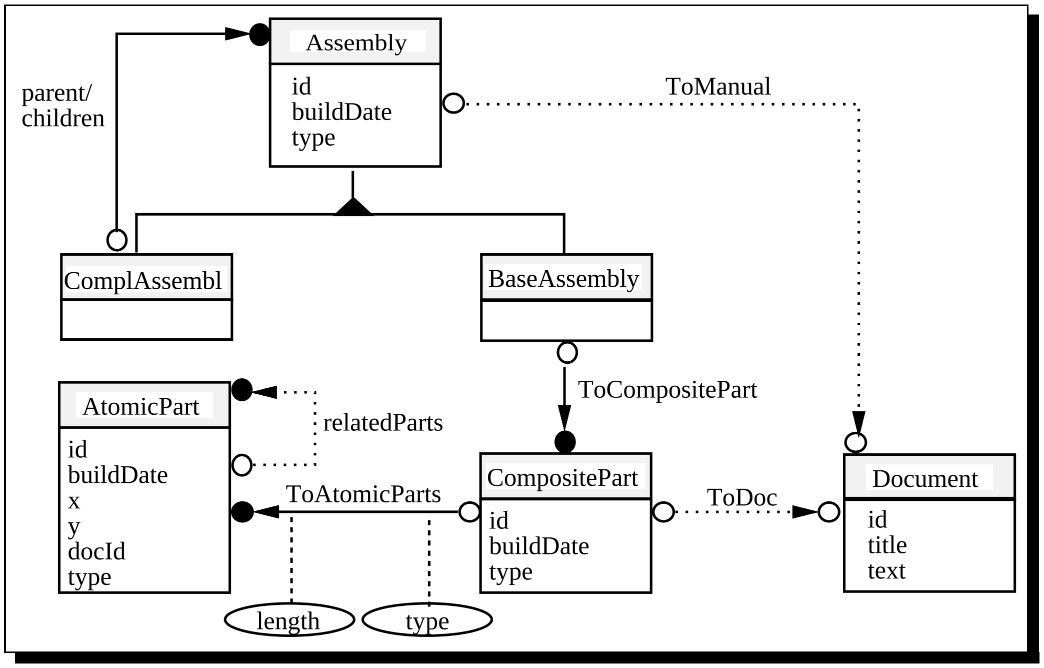


Figure 4.3: Conceptual Schema of the OO7 Benchmark

The conceptual schema of the OO7 Benchmark, displayed in Figure 4.3, includes six entities. All the entities have an attribute `id` modelling a unique object identifier. Entity `CompositeParts` models design primitives of the application area such as a register cell in chip design or a procedure in a programming language. A composite part has a `buildDate` attribute storing information about object creation time and a string attribute for storing `type` information of composite parts. Each composite part has a reference relationship to one documentation object, i.e. an instance of entity `Document`. Documents have a string attribute for storing a `title` and a long field attribute for storing `text`. A composite part is composed of a set of atomic parts. These parts model statements in procedures or gates in register cells. They have a number of small attributes that store graphical coordinates, type information, creation time and the like. In addition to the aggregation relationship, atomic parts can have reference relationships with each other. The set of all composite parts model a library of which complex designs or programs can be composed. The entity `Assembly` with its sub-entities `ComplAssembl` and `BaseAssembly`, as well as the aggregation relationships between them, determine this composition. Complex assemblies are composed by the `parent/children` relationship from a set of other assemblies, i.e. either complex or base assemblies. A base assembly, in turn, is composed of a set of composite parts.

The benchmark defines traversal and query operations. The traversal operations navigate through the composition hierarchy. Some of them update attributes during traversal. Updates that modify the composition hierarchy are not included in any of the operations. The query operations retrieve the set of all atomic parts or subsets of those that fulfil particular conditions on the `buildDate` attribute.

The OO7 Benchmark defines three initial database states: small, medium and large. All of them contain balanced ternary trees of assembly parts of height seven, i.e. 1093 assembly parts. The databases differ in that a small database contains 10,000 atomic parts, whereas medium and large databases contain 100,000 atomic parts. For a large database, ten assembly trees are established, whereas small and medium databases only contain one tree.

4.1.2 Application Specific Benchmarks

The suitability of a benchmark for a particular performance evaluation task depends on the degree to which the load characteristics of the benchmark match the characteristics of a future application. In the context of this thesis the benchmark's load on a database has to simulate the load characteristics imposed by syntax-directed tools in order to find those database systems that execute tool commands within the time frames required in Section 3.7. The load of a database application is heavily dependent on the data structures used in that application. From the discussion in the last chapter, we know how data manipulated by syntax-directed tools is structured. The benchmark must, therefore, address the following issues:

Entities and attributes: The number of different entities and their attributes in the benchmark schema must simulate the number and structure of node types that occur in project-wide abstract syntax graphs. Consider, for example, syntax-directed tools for structured analysis, modular design and implementation in any programming language. The graph schema will define a large variety of node types. Node types are rather heterogeneous with respect to number, type and size of their attributes. Some node types will have no attributes at all. Others will have small attributes for graphical coordinates (like nodes for processes in data-flow diagrams) or lexical values of identifiers. Others again will have rather large attributes (like mini specifications in data-flow diagrams or comments in programming language syntax graphs). Arbitrary combinations will exist. Unfortunately, the overall number of types, as well as the number, size and type of attributes defined for node types influence the time required for retrieving and creating objects. Hence the heterogeneity of types outlined will have to be reflected in a benchmark to address the performance requirements of syntax-directed PSDE tools.

Relationship types: Different relationships representing different kinds of edges have to be reflected in the benchmark schema. Relationships with aggregation semantics should be included to represent syntactic edges. The depth of nesting in these aggregations should meet the average height of syntax trees. Reference relationships must be included to represent non-syntactic edges. There should be single and multi-valued relationships to measure the different performance of operations on single and multi-valued edges. Ordered multi-valued relationships should be included to measure the database performance with list structures that often occur in syntax-directed tool schemas. Unlike unordered relationships, the database system will in addition have to retain an order, which will obviously require time.

Complex update operations: The operations of the benchmark should reflect the load on the database caused by tool commands. Only then can we draw conclusions from benchmark results on the performance of future tool commands. In particular, operations should not only include traversals and attribute updates, but also creation and deletion of objects of various types in order to reflect editing commands that effectively change the structure of syntax graphs. The commands offered by a tool are usually implemented by a number of elementary database operations. Unfortunately, simply summing up the execution time of these elementary operations gives wrong results. In fact, complex tool commands can sometimes be implemented much more efficiently by exploiting a particular feature of the database system under investigation than by just taking a particular order of predefined simple benchmark operations. Some database systems, for example, provide a special type called dictionary and a corresponding efficient member function. If this type and especially the member function had been implemented by a number of simple benchmark operations, this would result in a much higher execution time than if using the predefined member function.

Initial database states: The initial database states defined for a benchmark should reflect realistic situations of the application. This includes the number of objects in the database as well as the number and cardinality of different relationships between objects. If the number of objects chosen is too small, the database system might keep all objects in a system cache and perform far faster than with a realistic scenario. If the cardinality of relationships is chosen wrongly, searching in lists or sets might be more efficient than later in the real application.

In assessing the existing benchmarks against these requirements their inappropriateness becomes evident.

Entities and attributes: The main problem of the Simple and Hypermodel Benchmarks is that their conceptual schemas are too simple to meet schemas of PSDE tools. Both the Simple Benchmark and the Hypermodel Benchmark include just three different entities and, therefore, do not reflect the heterogeneity in number of entities nor in number, size and types of entity attributes that occur in project-wide syntax graphs of a PSDE.

Relationship types: The requirements for relationships are not reflected by any of the existing benchmarks. Neither the Simple, nor the OO7 Benchmark includes any ordered relationships. Therefore, these benchmarks cannot detect deficiencies of, for example, relational database systems that do not support ordered relationships. The Simple Benchmark schemes does not reflect the depth of nested aggregations that occur in syntax graphs. Neither Simple nor Hypermodel Benchmark adequately address reference relationships. In the case of the Simple Benchmark, the reference relationship `MNREL` is instantiated by the initial database and by benchmark operations as if it were a 1:1 relationship, i.e. the benchmark connects one object of type `BIG` with one of type `SMALL`. In the case of the Hypermodel Benchmark, the `partOf/parts` relationship links a node with five other nodes and the `refTo/refFrom` relationship relates a node with exactly one other node. In a PSDE, however, a reference relationship is usually of cardinality 1:n where n tends to become rather large, namely up to a few hundred. As an example, consider a basic type identifier in a large software system. It will be used in a large number of modules by a large number of operations as parameter or result type. All objects representing the use of this identifier must be linked by a 1:n relationship to the object representing the declaration.

Complex update operations: Simple Benchmark operations create, change and delete objects, but just like the conceptual schema itself, they suffer from oversimplification. They in no way reflect complex commands that occur in syntax-directed tools. As argued above, the performance results of these simple operations cannot be used for analytically deriving execution times of more complex operations. The Hypermodel and OO7 Benchmark do not create any new objects or relationships. Since almost any editing command will create new nodes and edges the performance figures obtained with these benchmarks will be more or less useless for our purposes.

Initial database states: The static and simple definition of the initial database for the Simple Benchmark does not at all reflect situations that appear in PSDEs. The same is true for the Hypermodel and the OO7 Benchmark. Even though they define more than one initial state, all states in both benchmarks define a completely balanced tree, which is a very unusual situation in PSDEs.

Besides clarifying the deficiencies of existing benchmarks, the above examples are supposed to indicate that even different PSDEs could have very different performance requirements, i.e. it is impossible to define a general benchmark for evaluating database systems for PSDEs let alone for arbitrary database applications. In more detail, the tools and types of documents in a PSDE determine the entities and their relationships, which vary significantly depending on the particular tools included in the environment. As mentioned, storage requirements for a structured analysis diagram are very different from the requirements for the storage of source code. In addition, inter-document consistency constraints are very important for the definition of a benchmark. In a PSDE following a transformational approach a much less fine-grained data model is needed than in a PSDE which facilitates intertwined document development. This, in turn, results in a much lower number of relationships for the transformational case. Furthermore, the initial database state depends on a particular application, i.e. tools, document types, and even the scale of projects being performed with the PSDE.

Our approach, therefore, is not to extend the benchmarks mentioned to meet (some) additional requirements from PSDEs, nor to develop another general benchmark for the area of PSDEs, but we suggest developing *application specific benchmarks*. In our case, we have to develop a benchmark that measures the database performance of syntax-directed tools which meet the requirements discussed in Chapter 2. We, therefore, propose an organised, carefully designed process to define an appropriate benchmark. This process includes the steps

1. definition of the benchmark based on the requirements of a particular application and
2. benchmark implementation on top of the database systems under investigation.

The methodology for developing application specific benchmarks is described in the next section. While we describe the methodology, we illustrate its use for developing the *Merlin Benchmark*. The Merlin Benchmark will allow performance measurement of database systems for syntax-directed tools.

4.1.3 Developing Application-specific Benchmarks

Our approach towards benchmark definition is to start from an existing application whose load characteristics cover those of future applications. For the scope of this thesis, this means that we have to look for an environment that includes graphical as well as textual syntax-directed tools. The complexity of the languages of the environment must be representative of that of a large number of languages. Complexity of a language here is measured in terms of number of different increments, which determines the number of syntax graph node types. Moreover, the environment should check and preserve static semantics and inter-document consistency in the same way as we expect this from tools.

The Groupie [ES94] environment seems to meet these requirements for the Merlin Benchmark reasonably well because

1. it supports the editing of graphical architecture documents that are, from a structural point of view, very similar to other graphical documents, such as data-flow diagrams, entity relationship diagrams or Petri-Nets,
2. it supports the definition of textual module interface definitions that are very similar to Modula-2 module definitions, Ada package definitions, C header files or C++ class definitions, and
3. the two tools are highly integrated in order to check and even automatically preserve static semantics and inter-document consistency, as far as possible.

If the benchmark properly simulates the database load of Groupie tool commands, we will be able to forecast database performance for a large number of graphical and textual tools. In addition, we will be able to forecast the impact of inter-document consistency preservation on database performance. Nevertheless, there are tools, such as editors for implementing modules, packages or classes, which have more demanding load characteristics. This is because their syntax-graphs are even deeper and it will take longer to traverse them, for instance during unparsing. The structures that occur inside these graphs, however, are covered by the Groupie tools as well. We can, therefore, be sure that the benchmark implementation will reveal the fastest database system. It remains to be seen whether the fastest system still performs fast enough when building tools for programming languages on top of it. Using our process suggested for benchmark definition one could define a benchmark for these implementation tools. This, however, is beyond the scope of this thesis.

The overall objective of the definition process for an abstract database benchmark now is to derive a conceptual database schema, the corresponding update and retrieval operations and a set of initial database states.

4.1.3.1 The Conceptual Schema

The derivation of the conceptual database schema starts from taking the syntax definition of each document whose development is supported by a particular PSDE. The syntax definitions are usually given as, or at least can be transformed into, a tree grammar representation defining the abstract syntax of each document. This is what we need as the basis for our benchmark definition.

Taking our Groupie example, the abstract syntax of two document types is defined as shown in Figure 4.4. The first document type allows modules and their import-relationship to be defined. The second type is dedicated to the detailed definition of export- and import-interfaces of modules. In particular, it allows the declaration of types, procedure heads, and function heads exported by a module and refines the import-relationships from other modules by allowing the definition of imported objects for each relationship.

Based on the transformation rules given in Table 4.1, it is a straight-forward exercise to derive a conceptual schema in terms of an EER model, from a tree grammar. Applying the transformation rules of Table 4.1 to the tree grammar excerpt in Figure 4.4 results in the EER model in Figure 4.5.

```
Fixed arity operators:

arch        -> ARCH_ID MOD_LIST
module      -> MOD_ID  COMMENT  IMP_LIST

Atomic operators:

arch_id     -> IDENT
mod_id      -> IDENT
comment     -> STRING

List operators:

module_list -> MODULE ...
import_list -> IMPORT ...

Phyla:

ARCH_ID     :: arch_id
MOD_LIST    :: module_list
MODULE      :: module
MOD_ID      :: mod_id
IMP_LIST    :: import_list
IMPORT      :: mod_id
COMMENT     :: comment
```

```
Fixed arity operators:

module      -> MOD_ID   COMMENT  EXPORT_PART  IMPORT_PART
export      -> TYP_ID   OP_LIST
func        -> OP_ID    PAR_LIST TYP_ID  COMMENT
proc        -> OP_ID    PAR_LIST COMMENT
cbv_par     -> PAR_ID   TYP_ID
cbr_par     -> PAR_ID   TYP_ID
import_part -> IMP_LIST
import      -> MOD_ID   IMP_OBJ_LIST

Atomic operators:

mod_id      -> IDENT
typ_id      -> IDENT
op_id       -> IDENT
par_id      -> IDENT
comment     -> STRING

List operators:

op_list     -> OP     ...
import_list -> IMPORT ...
par_list    -> PAR    ...
imp_obj_list-> IMP_ID ...

Phyla:

MOD_ID      :: mod_id
COMMENT     :: comment
EXPORT_PART :: export
IMPORT_PART :: import_part
OP_ID       :: op_id
TYP_ID      :: typ_id
PAR_ID      :: par_id
OP          :: func proc
PAR         :: cbv_par cbr_par
IMP_LIST    :: import_list
IMPORT      :: import
IMP_OBJ_LIST:: imp_obj_list
IMP_ID      :: typ_id op_id
```

Figure 4.4: Abstract Syntax for Merlin Benchmark

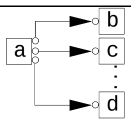
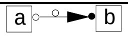
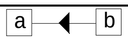
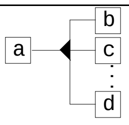
For tree grammar component of type	Substitute $\boxed{a}$ with
Fixed arity operator `a -> B C .. D`	
List operator `a -> B ...`	
Atomic operator `a::=B`	
Phyla `a::B C ... D`	

Table 4.1: Transformation of Tree Grammar into EER Models

The next step is to extend the schema with reference relationships, which model reference edges. Not including such relationships in a schema would result in significant performance increase of all update operations based on the schema definition. The additional relationships to be introduced are all of cardinality one-to-many.

As an example, the constraints defined for the Groupie documents that must be materialised in additional relationships are that

1. the interfaces of modules that occur in an architecture diagram are specified in the specification document and vice versa,
2. each import interface relationship in the architecture is specified in detail in the specification language and vice versa,
3. names of modules, types, functions and procedures are unique within an architecture

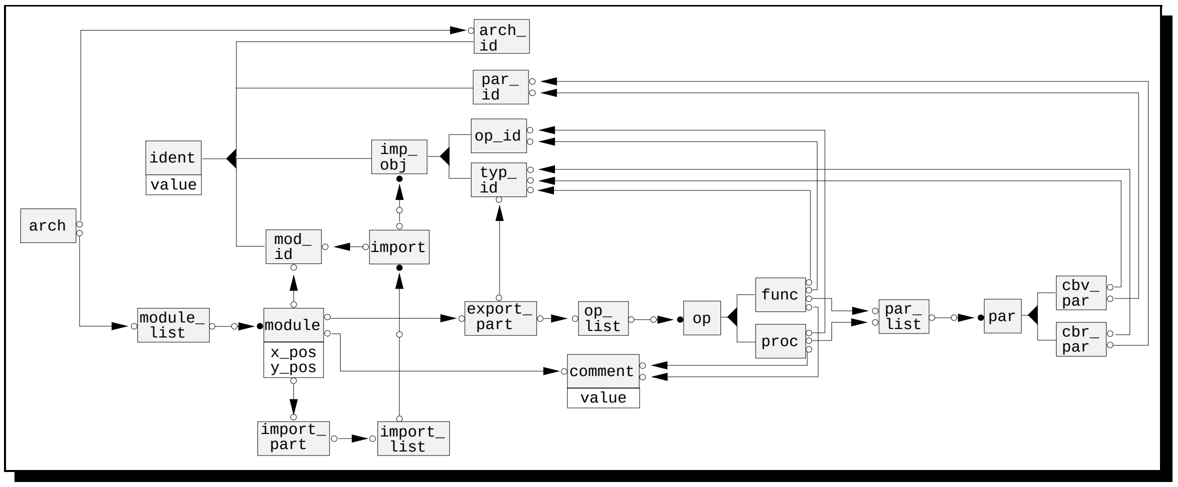


Figure 4.5: EER Model Deduced from Grammar

diagram and all related specification documents,

4. modules that participate in an import-relationships have to exist,
5. objects that are imported by an import-relationship are exported elsewhere,
6. cyclic import relationships are forbidden,
7. types used in parameters of operations and result types of functions are declared, i.e. they are either exported by the module in which they are used, or imported from elsewhere.

The result of adding relationships, according to those constraints, to the EER model of Figure 4.5 is given in Figure 4.6. The first additional relationship type, which we call *dictionary*, is drawn using dashed lines. A dictionary is a 1:n relationship that provides for efficient associative access to the multiple relationship elements. It will be used to implement scoping rules efficiently. One dictionary relationship defines all identifiers which are declared in an architecture. Another dictionary relationship defines all the identifiers which are exported by a module, i.e. that may be used as imported objects. The last dictionary relationship defines all type identifiers that may be used in a module. To avoid change propagations, reference relationships, which substitute aggregation relationships, allow for object sharing. In particular, the types used in parameters of operations, or result types of functions, are no longer viewed as copies of types defined in export interfaces, but as references to them. Furthermore, the copies of identifiers of imported modules and objects in import lists are transformed into references to the respective identifiers. This not only enables the omission of time consuming change propagations, but also enables quick checks to be made as to whether an exported type or operation is actually used.

Implementing this schema would require as much effort as building a schema for the two Groupie tools, which is far too much. Benchmarks should be simple to implement. The next and major step is, therefore, a simplification of the schema defined so far. This simplification results in a benchmark schema which will be simpler to implement while still addressing application specific load requirements.

The simplification is defined by a number of rules. The application of these rules removes all entities and relationships which do not influence the performance of benchmark operations. The entities remaining in the schema then represent the worst case situation. The rules for schema simplification are that

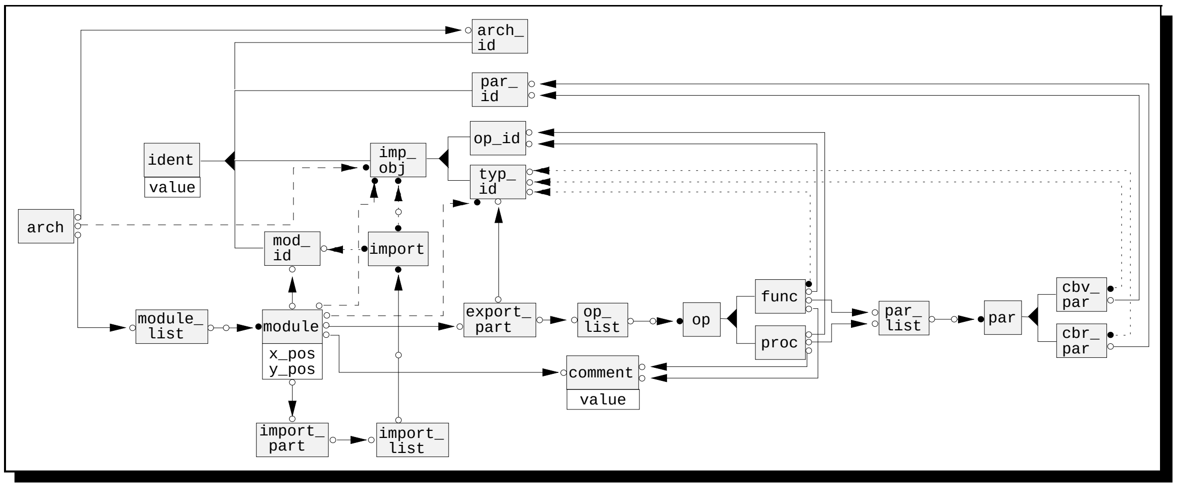


Figure 4.6: EER Model Enhanced with Context Sensitive Relationships

1. relationships, which start from or end in all sub-entities of an inheritance relationship, are replaced by one relationship which starts from or ends in the super-entity,
2. entities, which do not participate in any relationship except as a target of an aggregation relationship, are transformed into attributes of the entities, where the aggregation relationship starts,
3. sub-entities of an inheritance relation, which participate in the same relationships and carry the same attributes as another entity of that inheritance relation, are removed. The remaining entity is then considered to be a representative of the removed entities. As a consequence, execution times of benchmark operations that access this entity should be interpreted as upper bounds rather than as exact values of operations that would have accessed objects of the removed entity,
4. an inheritance relationship with only one sub-entity is removed with its sub-entity. The super-entity subsumes all relationships the sub-entity participated in, as well as all the subentity's attributes,
5. an entity that neither participates in a context-sensitive or inheritance relationship nor carries attributes and which is the source of only one aggregation relationship, is removed. The aggregation relationship that started from the entity now starts from each entity that had an aggregation relationship with the removed entity.

The order for the application of these simplification rules to the EER model is as follows. Rules 1-4 may be applied repeatedly in any order. The application of Rules 1-4 stops when none of the rules can be applied any longer. As a result these rules might produce obsolete entities. These obsolete entities are then removed by Rule 5. After that Rules 1-4 could again be applied. They must, however, not be applied again in order to avoid oversimplification of the EER model.

Using these simplifications we are able to simplify the EER model shown in Figure 4.6. The result of this process is depicted in Figure 4.7. We applied Rule 1, 3, and 4 to `cbv_par` and `cbr_par` with the effect of removing these entities and transferring their objective to `par`. Entity `par` is now considered as a representative of `cbv_par` and `cbr_par`. Then we were able to apply Rule 2 to `par_id` transforming it into an attribute of entity `par`. After that, we applied Rule 3 to `func` and `proc` with the effect of removing the entity `proc`. That enabled us to apply

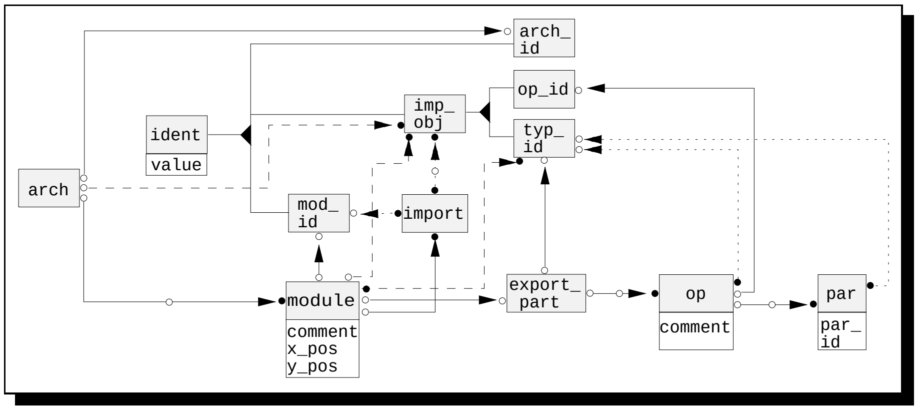


Figure 4.7: EER Diagram of Simplified Database Base Schema

Rule 4 to `func`, with the effect of replacing entity `func` by its super-entity `op`. As `comment` is merely the target of aggregation relationships, we could apply Rule 2 to it, thus transforming `comment` into attributes of `module` and `op`. The same rule was applied to entity `value` connected to `ident`, transforming this entity into an attribute of `ident`. Finally, the entities `module_list`, `import_part`, `import_list`, `op_list`, `par_list` were removed according to Rule 5.

4.1.3.2 The Initial Database States

The next step in defining a benchmark is the definition of an initial database. In order to determine a realistic structure and a realistic number of objects, we perform an analysis of existing documents. We assume that these documents exist. Tool builders have usually gained preliminary experience with the document types while they were producing documents during case studies. We use these documents to analyse the number of objects that participate in 1:n relationships of the benchmark schema. Moreover, we have to obtain average sizes for the attributes defined in the schema. The required analysis of existing documents can be performed by a parser generated by `lex` and `yacc` for instance. In contrast to the Simple, Hypermodel and OO7 Benchmarks, this approach leads to initial databases that have a structure similar to that of real documents.

Metric component	Value
Number of exported types	1
Number of exported operations	17
Number of imported modules	4
Number of identifiers	132
Number of comments	18
Number of imported objects/import relationship	6
Number of parameters per operation	3
Length of identifiers [bytes]	12
Length of comments [bytes]	256

Table 4.2: Metric for a Module in the Initial Database

The structure of the initial database for the Merlin Benchmark is based on the analysis results of approximately 5,000 lines of specification produced when specifying Groupie itself. Table 4.2 defines the number of components of a module contained in the initial database according to the schema defined in Figure 4.7. To vary the size of the initial database we increase the number of modules by increasing the number of levels in the architecture as follows. The import-relation between modules leads to a directed acyclic graph of modules. We divide the modules into n levels ($n \geq 3$). Each level L_i contains 2^i modules ($i \in \{0, \ldots, n-1\}$). Except for the top-most level where a module imports from both modules at level 1, a module in level L_j imports from four random modules of level L_{j+1} ($j \in \{1, \ldots, n-2\}$).

4.1.3.3 The Benchmark Operations

The final step in defining a benchmark is the definition of the operations. It follows the guidelines discussed for operation definition introduced on Page 42. We, therefore, include operations for creation of all entities that are defined in the benchmark schema, various traversals through the syntax graph in order to measure performance of unparsing operations, changes to attribute values and operations that delete objects. The main point in the detailed operation definition is to define parameter values for the operations in a way that they respect the structure of the initial database. Thus, the definition of those values is also based on the above mentioned analysis results.

A problem that we have to address is the variance in execution times of complex benchmark operations. The execution times vary because they depend on the states of external devices, such as the position of disk heads, the hardware bus, caches of disk controllers, of the operating system and of the database system itself. As these states are beyond our control we must execute benchmark operations repeatedly and compute mean values of the operation execution times to obtain reliable results. Repeated operation executions, of course, must be done from exactly the same logical database state[3]. For two reasons it is not appropriate to create the database anew before measuring each operation. First, creating a database of a considerable size is rather time-consuming (in the order of magnitude of hours) whereas benchmark operations will execute within a few hundred milliseconds. Secondly, the physical database organisation will vary and this might even increase the variance of operation execution times. Instead, we suggest to organise time measurement of benchmark operations within *benchmark runs*. A benchmark run is a sequence of benchmark operations that leaves the database in exactly the same logical state that it had before the run began. This means that we first have to perform create operations, then perform traversals and attribute changes (or vice versa) and then have to delete all created objects again. We can then repeat the execution of benchmark runs and have achieved that a benchmark operation is in each run performed in the same logical database state.

According to the above considerations, the operations of the Merlin Benchmark are clustered into four groups. Operations of the first group create increments of the two document types like modules, types, operations, parameters and comments. The second group is dedicated to measuring the impact of changing attribute values. The third group simulates tool operations that perform traversals through the database. The last group deletes all previously created objects. A Merlin Benchmark run, therefore, consists of the following operation sequence.

OpenOMS: Open the access to the OMS, i.e. perform all necessary operations such as authorisation or reservation of buffers to access data in the OMS.

[3]We will not be able to influence the physical database state, i.e. the way how objects are stored on disk pages, since it is controlled by the database system and this is in fact one of the reasons why multiple operation executions are required.

CreModul: Create n new modules in the architecture. Set their names to `Module_No_j+2`^{i}, ($j \in \{0, \ldots, n \Leftrightarrow 1\}, i =$ levels in the initial database). Set the graphical coordinates to $(50 \cdot j, 100)$.

CrModTyp: For each created module set the export type identifier to `Type_No_j+2`^{i}, ($j \in \{0, \ldots, n \Leftrightarrow 1\}, i =$ levels in the initial database).

CrModCom: For each module created by **CreModul** expand the module comment increment to an arbitrary string which is 256 bytes long.

CrModImp: For each created module create four import relationships to arbitrarily given modules of the bottom-most level of the initial database.

CrImpObj: For each created import relationship of each created module increment expand the list of imported objects to the exported type and the $1^{st}, 3^{rd}, 7^{th}, 11^{th}$, and 14^{th} operation of the respective module.

CrExpOpe: Create a list of 17 operation increments in each created module, update their name to `Mod`j`_Func_`k with j being the number of the module as determined by the **CreModul** operation and $k \in \{1, \ldots, 17\}$ and expand their type to the type which is exported by the module.

CrOpePar: Expand each operation parameter increment in each created module to a parameter list with three call by reference parameters. Expand the parameter names to arbitrary unique identifiers (12 bytes) and the parameter types to the types which are imported by the first three import lists of the module.

CrOpeCom: For each operation expand the operation comment increment to an arbitrary string of length 256 bytes.

UnparMod: Create textual representations of the created modules in a string of the host programming language.

UnparArc: Create a textual representation of the architecture in a string of the host programming language.

AnaUsage: For each module of the bottom most level of the initial database, return the names of modules to which it exports objects. Return the names as a list of the host programming language.

ClosTrav: Return the names of those modules that are reachable from the module at level 1. Return the names as a list of the host programming language.

ChModNam: Change the module name of each module to another unique name of the same length and update the name in the import parts of those modules which use the module.

ChModTyp: Change the exported type of each module to another name of the same length and update the name in the import parts and parameter lists of those modules which use the module.

ChModCom: Change the contents of the module comment of each created module to another arbitrary string of the same (256 bytes) length.

ChOpeNam: Change the names of the exported operations of each module to another name of the same length and update the name in each import list which imports the operation.

ChParNam: Change the names of each operation parameter to a new value of the same length.

DlOpeCom: Delete every operation comment in all modules that has been created during the benchmark.

DlOpePar: Delete each operation parameter in each module that has been created during the benchmark.

DlOperat: Delete all exported operations in all modules that have been created during the benchmark.

DlImpObj: Delete all imported objects in all modules that have been created during the benchmark.

DlImpRel: Delete all import relations in all modules that have been created during the benchmark.

DlModule: Delete all modules that have been created during the benchmark.

CloseOMS: Perform all operations that are necessary to start the benchmark again.

4.1.4 Summary

In this section we have discussed database performance evaluation using abstract database benchmarks. We have argued, why the three general purpose benchmarks Simple, Hypermodel and OO7 Benchmark are inappropriate for our as well as many other database performance evaluation tasks. We, therefore, consider database performance evaluation as a problem that cannot be solved in general. To solve it, it is required to exploit knowledge of a particular application domain. We have suggested a method to acquire this knowledge in terms of application-specific load characteristics and use it for the definition of application-specific benchmarks. The load characteristics are elicited based on existing schemas and data. A conceptual benchmark schema is defined based on a representative schema of the application domain. Then existing data is analysed in order to determine realistic cardinalities for relationships in an initial database. In a third step, benchmark operations are defined that create objects of the entities contained in the benchmark schema, change attributes, traverse along relationships and finally delete the objects again. Following the suggested method, the Merlin Benchmark has been defined in order to measure database performance for syntax-directed tools. We have implemented the benchmark for representative database systems of different classes. In the next sections, we discuss the Merlin Benchmark implementations and the results of its execution along with the assessment of the respective classes using the functional requirements presented in the previous chapter.

4.2 Relational Database Systems

The relational data model was introduced by Codd [Cod70] and has gained substantial recognition especially in business applications. Meanwhile a number of database systems implementing the relational data model have evolved as highly reliable products. These systems are called relational database systems (RDBSs). RDBSs are available for almost any hardware and operating system platform. The most important RDBSs are Ingres [SWKH76], Sybase, Oracle, Informix and DB2. A PSDE built on top of one of these systems would, therefore, be built on a firm base and would (at least from the DBSE point of view) be highly portable.

It has often been claimed that RDBSs are inappropriate for storing complex structures, such as abstract syntax graphs (c.f. [Mai89] for instance). The major concern of these claims is RDBS performance. They refer to the work of Linton, who has implemented storage of abstract syntax trees for a programming environment in Ingres [Lin84] 15 years ago. Meanwhile, a number of advances have been made in the implementation of RDBSs as well as the performance of underlying operating system and hardware platforms. Moreover, an algorithm has been proposed that facilitates an efficient materialisation of the transitive closure of graphs [ABJ89]. A major result of this section is the observation that current RDBS implementations perform efficient enough for storage of abstract syntax graphs if these advances are exploited. As main disadvantages of RDBSs we rather identify the lack of appropriate version management primitives, non-updatable views and a schema definition language that does not support the encapsulation of graph structure definitions.

Definition and Manipulation of Syntax Graphs in RDBSs

The relational data model is based on sets of tuples of attributes. In the first normal form[4], attributes are of atomic types only. These sets are called *relations*. RDBSs represent relations as *tables* in which each column represents an attribute type, each row represents a tuple and each column of a row represents an attribute. A relational database consists of a set of named tables. Usually, a table has a column where *unique key attributes* are stored. These keys are then used to identify tuples. In order to implement one-to-many relationships between tuples, a column for secondary key attributes is defined where unique keys of related tuples are stored. Many-to-many relationships are implemented in separate tables that have columns for the unique keys of the related tuples.

The data definition language (DDL) is standardised for all RDBSs. It is used for defining the structure of tables. The `CREATE TABLE` directive of the relational DDL has the following form:

```
CREATE TABLE <name> (<att_name> <att_type> [<att_options>],...)
```

It creates a new table with the given name and as many columns as attributes are declared in the create directive. An attribute declaration includes a name and an atomic type. The name can be considered as the column title and the type defines the kind of data that can be stored in the column.

If abstract syntax graphs have to be stored in an RDBS (c.f. [Lin84]), a table has to be created for each node type. Tuples in such tables implement nodes of the respective type. Each table has a column with unique key attributes that are used as node identifiers. Node attributes are implemented by additional columns in the respective table. This is only possible for node attributes that are not structured any further. For structured attributes, such as error lists or dictionaries, additional tables and references between them are required. Edges are also implemented by additional columns of tables. If an edge exists between two nodes, the node identifier of the source or target node is stored in a column of the tuple implementing the target or source node respectively. Note that it is not necessary to store node identifiers in both tables to implement an edge. Navigation in both directions will be made possible by the associative nature of relational query languages. To implement multi-valued edges that lead from nodes of type T_1 to an arbitrary number of nodes of type T_2, the edge should be implemented by an additional column in the table implementing T_2. Then this table may contain multiple tuples with the same value v in this column, meaning that these tuples implement nodes that

[4]Non First Normal Form (NF^2) databases [SP82] where attributes may be *tuples* or even sets are not supported by any of the above mentioned systems.

are all connected to the node identified as v. Implementing it in the table representing T_1 would violate the *unique key constraint.* This constraint requires that tuples differ in the unique keys. It would be violated because the table would contain tuples that only differ in the column representing the multi-valued edge. If a multi-valued edge between T_1 and T_2 is ordered, an additional column is required in the table for T_2 in order to store node positions. Heterogeneous edges, i.e. edges that lead to different types of node cannot be implemented in a straight-forward manner because one column cannot reference node identifiers from arbitrary tables. To implement heterogeneous edges, we have to store different types of nodes in one table and store type information in an additional column.

Module						
ID	TYPE	name	com	type	opl	imp
100	ADT	1	1	2	10	55
101	TC	6	4	...	NULL	NULL

ModName	
ID	value
1	Window
6	BasicTypes

ImportInterface
ID
55

OperationList
ID
10

Operation							
ID	TYPE	name	pl	type	com	OPL	POS
1	FUNC	3	70	4	2	10	1
2	PROC	5	71	NULL	3	10	2

OpName	
ID	value
3	CreateWindow
5	DeleteWindow

ParameterList
ID
70
71

Parameter					
ID	TYPE	name	type	PL	POS
11	IN	...	...	70	1
12	IN	...	...	70	2
13	IN	...	...	70	3

Comment	
ID	value
1	/* defines a type ...
2	/* creates a new window */
3	/* Deletes ...
4	/* defines a set of ...

UsingType		
ID	value	DefinedIn
2	TWindow	...
4	TWindow	...

ImportList				
ID	fm	imp	IMPINT	POS
1	7	...	55	1
2	9	...	55	2

ImpModule		
ID	value	ImpFrom
7	Position	...
9	BasicTypes	6

Figure 4.8: Tables Implementing an Abstract Syntax Graph

As an example, consider how the abstract syntax graph displayed in Figure 3.1 on Page 21 is implemented in a set of tables. Figure 4.8 depicts twelve tables that only partly implement the graph. In that figure, the headline of a table is the table name. The next line shows the attribute names, i.e. the column headlines. In order to save space the attribute types are not given explicitly. The reader can easily deduce them from the values stored in the respective column. Unique key attributes are underlined. If an attribute value represents an edge to a node not included in the figure, "..." are inserted instead.

The table `Module` implements node types `ADTModule` and `TCModule` since these two are used in a heterogeneous aggregation. The attribute `TYPE` is used to distinguish them. Attribute `name` refers to the node identifiers of Table `ModName` which implements the edge `name` between the respective nodes. Attribute `com` refers to node identifiers of Table `Comment` which implements edges labelled `com` between the respective nodes. Table `ModName` provides an attribute `value` for storing lexical values of identifiers that are module names. Table `Operation` implements heterogeneous operation lists so that they can consist of procedures and functions. Therefore, column `TYPE` stores the information as to whether a tuple in this table implements a procedure or a function node. The association between tuples of this table and tuples in Table `OperationList` is achieved using the attribute `OPL`. It implements that these two tuples implement nodes that are reachable, via a multi-valued edge, from the operation list node, implemented by the tuple in Table `OperationList`. To implement the order between these operations, the column `POS` is used. When traversing through the multi-valued edge, one node must be visited before another node, if the `POS` attribute of the first node is smaller than the `POS` attribute of the other node. Multi-valued reference edges are implemented similarly. As an example consider the edges that connect use and declaration of types, or import and export of names. To implement the latter edge, Table `ImpModule` has a column `ImpFrom` that contains references to identifiers of Table `ModName`.

We note that schemas for syntax graphs that are constructed in this way are in *third normal form* [Dat86], because node attributes are the only non-key attributes in these tables and they are non-transitively dependent on the node identifier.

If a schema of a PSDE is defined in this way, we will be faced with the problem of not being able to restrict edges to lead only to nodes of particular types. This is because the only attribute types that can be used for a column are the predefined types. Hence, integrity between tables must be controlled by the tool builder without any further support from the RDBS. Another drawback of the relational DDL is that the schema will be poorly structured. Considering PSDEs that support a number of different languages, the underlying abstract syntax graph could easily contain some hundred different node types. Hence, the schema defined in this way would contain as many different tables that are not structured any further. Moreover, the DDL does not provide any dedicated concepts for implementing heterogeneous and ordered multi-valued edges. They must be hand-coded on top of the relational data model. Implementing heterogeneous multi-valued edges in the way we suggested, will lead to meaningless columns of tuples (such as the `opl` attribute for the second tuple in Table `Module`). Furthermore, we have to hand-code ordered multi-valued edges with additional position attributes. If a new node is inserted into a list, the position attributes of all the tuples that occur after the tuple which implements the new node, must be adjusted. This requires effort during tool construction and decreases performance.

Data access and manipulation in the relational data model are based on the relational algebra [Dat86]. The relational algebra operators are implemented in data manipulation languages such as the Standardised Query Language (SQL) which evolved from System/R's query language SEQUEL [ABC$^+$76]. As SQL is supported by any of the above systems and its expressive power is similar to other languages, we only consider SQL in this thesis and ignore other relational query languages. SQL consists of four commands: `SELECT`, `INSERT`, `UPDATE` and `DELETE`. The most general form of a `SELECT` statement is:

```
        SELECT att1, att2, ..., attn FROM table1, ..., tablem WHERE <condition>
```

This statement first joins the given tables, then selects all those tuples that fulfil the given condition and finally projects the table along the given columns named by the given attributes[5]. It thus provides a DML language construct for the relational algebra operators selection, projection and join. In more specialised forms, the `SELECT` statements only execute one or two operators. If only one table name is defined, the statement will not cause a join of tables. If no condition is given, the selection will be omitted. If a "*" is defined instead of explicit attribute names, no projection will be executed. The `INSERT` command implements the union operator and the `DELETE` command implements the difference operator of the relational algebra. The `UPDATE` command provides a means to assign new attribute values to attributes in tuples or even to attributes in sets of tuples. It is thus a short-cut for deleting and inserting these tuples.

[5]In practice, the query optimiser of RDBSs often chooses a different ordering, but this is transparent to the tool builder.

In order to implement abstract syntax graph operations in SQL, a tool builder can only use these primitives for table accesses and manipulation. To replace a subgraph g_1 with g_2 (which is a very common pattern for operations on abstract syntax graphs), a sequence of SQL `DELETE` statements has to be used to, firstly, delete all tuples that implement nodes of g_1 from a set of tables, which implement node types occurring in g_1. Then a sequence of SQL `INSERT` statements is needed, which inserts tuples for each node of g_2 into the respective tables. Finally, the attributes of all tuples that referred to node identifiers of deleted nodes must be adjusted with an SQL `UPDATE` statement in order to preserve referential integrity. To implement an unparsing operation of a subgraph, a sequence of SQL `SELECT` statements is needed: one for each table implementing a node type contained in the subgraph.

Operations that access and modify the abstract syntax graph cannot be defined within the schema as required, but must be implemented on top of it, since SQL is not computationally complete. The results of queries must be evaluated in a programming language. Therefore, RDBSs offer pre-compilers that allow for SQL statements embedded in a host programming language. In fact, the abstract syntax graph structures defined in the schema are not encapsulated, but may be queried and updated in an arbitrary way. In addition, implementation of inter-document consistency checks require that tools access tables defined for other tools. Therefore, a table defined in the schema cannot be associated with a particular tool but may be used within multiple tools. To make things even worse, it cannot be decided which tools use a schema component in case the component is changed.

Views

RDBSs offer a view mechanism in order to define virtual tables based on tables already defined. These virtual tables are not physically stored in the database, but computed, based on existing tables. The view mechanism is based on the SQL `SELECT` statement. Its general form is:

```
CREATE VIEW name AS (SELECT ... FROM ... WHERE ...);
```

It creates a virtual table and fills it with the result of the query defined by the SELECT statement. As soon as any of the tables accessed by the query are updated, the view is transparently and incrementally re-evaluated as well.

To a certain extent this view mechanism can be exploited in order to have different views on a conceptual abstract syntax graph as required in Section 3.3. A tool builder can define a view as a query that projects particular attributes in order to hide edges starting from a node or attributes defined for a node. To hide operation lists and type names from the table `Module`, for instance, for using the table within the Groupie architecture tool, the following view could be used:

```
CREATE VIEW Arch_Module AS
        (SELECT ID, TYPE, name, imp FROM Module);
```

This statement projects table `Module` along `ID`, `TYPE`, `name` and `imp` and thus hides columns `opl` and `type`. The virtual table `Arch_Module` is then visible in the schema as if it were a real table.

The use of an RDBS view facility is severely limited due to the view-update problem [BS82, GPZ88]. View updates can only be performed if they do not modify that part of a database that is complementary to the view. For software development tools, this problem is worse than for commercial applications since updates in tools occur much more frequently. As an example of the view-update problem in the domain of PSDE tools consider the following scenario for the above view. An integrity constraint for a conceptual schema may require that whenever a node of type `Module` is created, the edges to the type name and operation list must be created as well. Creation of such a module cannot be initiated through view `Arch_Module`, because it would also require an update in the complement of the view. As updates affect other tables, except in very rare cases, views of tables implementing syntax graphs are in general not updatable. This confines the use of views to tools such as browsers or static analysers that do not change the underlying tables.

Schema Updates

RDBS schemas can be updated by inserting new columns into table definitions using the `ALTER TABLE` statement or by deleting tables with the `DROP TABLE` statement. The contents of tables whose structure has been updated are preserved as far as possible. In the case of a newly created column, the rest of the table is unaffected and the new attributes in that column are initialised.

These schema updates are sufficient to implement the changes required in Subsection 3.4. Adding an edge or attribute to a node type can be implemented by creating a new column in the table implementing the node type. Deleting a node type can be implemented by deleting the respective table. Deleting an edge or attribute from a node type can be implemented by creating a new table without the column implementing the edge or attribute and filling it with a projection that drops the respective column.

The integrity of existing tables, however, can neither be automatically guaranteed by the RDBS, nor can a tool builder define a strategy, which the RDBS can use to re-establish integrity. Instead, the tool builder has to reestablish integrity after the schema update manually. Therefore, newly created columns have to be filled with reasonable values. A further problem in RDBSs, which is due to the separation of data structure definition in the schema and operation definition in a programming language with embedded query facility, is the consistency between these two subsystems. It has to be checked and reestablished manually whenever a schema is changed. For instance the case could arise where a deleted table or column is still used in an embedded query and the RDBS cannot draw a tool builder's attention to that. Therefore, all tools using the schema have to be recompiled after a schema change.

Versions

In Section 3.5 we required support from the DBSE to version subgraphs of the project-wide abstract syntax graph. The relational data model only defines the concepts set, tuple and attribute. A concept that could be used to define subgraphs is not supported by relational DDLs. Therefore, subgraphs cannot be defined statically in the schema, but have to be defined operationally. This operational definition cannot be achieved by using only a relational query language because computation of a subgraph requires the closure of those nodes reachable via particular edges from the root node of the subgraph to be computed. Computation of this closure requires iterations or recursions as nesting may occur. In fact, these iterations or recursions cannot be expressed in a relational DML since only tables, views or results of other queries can be queried, but a query cannot use its own result. Identification of subgraphs, therefore, can only be achieved in a host-programming language using embedded DML statements.

As a consequence, RDBSs do not offer any of the basic versioning operations required for subgraphs in Section 3.5 either. These would also have to be implemented within a host-programming language. A lazy duplication strategy for nodes cannot be implemented at all since it has to be integrated with the RDBS concurrency control manager as it was discussed in Section 3.6.

Transactions

RDBSs support ACID transactions in the way we required in Subsection 3.6. In particular, an RDBS transaction can group a set of SQL statements that implement an abstract syntax graph operation. The effect of these statements on a database will persist if and only if the transaction is completed with a commit. In the case of an explicitly issued abort command or in the case of an implicit abort due to a hardware or software failure, the database is restored to its state before the transaction started. RDBSs ensure isolation of transactions. Therefore, the integrity of an abstract graph cannot be violated by concurrent updates. This allows the implementation of tools that work in parallel on the same abstract syntax graph.

The commercially available RDBSs only support these conventional transactions. To our knowledge there are no RDBSs that support a concept that could be used to implement activities.

Performance

Unlike the assessment of functional requirements that relies on properties common to all RDBSs, the execution of benchmarks requires the use of a particular RDBS implementation. We have taken Oracle because it is one of the most popular and well-tuned systems. The benchmark results, however, will be influenced by particular design and implementation decisions taken for the development of Oracle and have to be interpreted carefully. We shall therefore, pay attention to those results that derive from deficiencies of the relational data model and query language, and take the benchmark results to draw a general picture on RDBS performance with syntax graphs. The implementation of the Merlin Benchmark for Oracle is described in detail in [Sch94]. We only outline the main results here.

All performance figures for the Merlin Benchmark implementations on top of the various database systems have been obtained on the same hardware and software configuration (c.f. Table 4.3) so that the results can be compared. For the RDBS benchmark we used the Oracle Version 6.0.36.5.1.

Machine	Sun SPARCstation IPX
Operating System	SunOS 4.2 Release 4.1.3
Main Memory	40 MB
Disk	WRENV IV 94181-385h with 320 MB
Disk controller	Emulex MD 21

Table 4.3: Description of the Hard- and Software Configuration

Figure 4.9 depicts the execution times for update operations of the Merlin Benchmark. In this implementation, the conceptual benchmark schema was implemented in the way suggested by

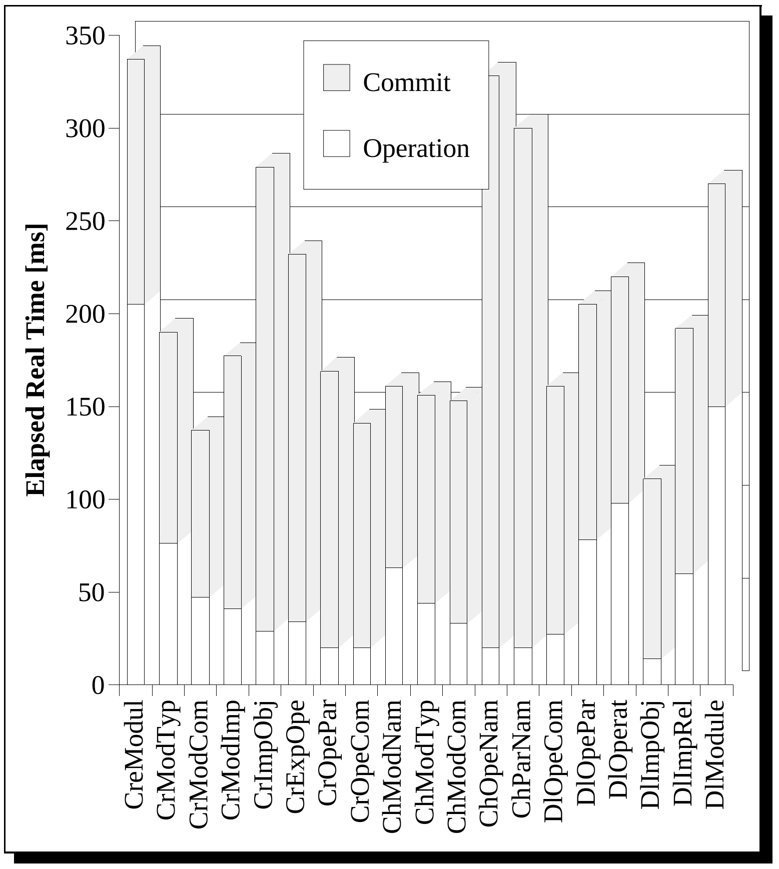


Figure 4.9: Oracle Update Operations in a Small Database

Figure 4.8 on Page 54. The benchmark was executed five times on an initial database with three levels, i.e. a small database containing only $\sum_{i=0}^{2} 2^i = 2^3 \Leftrightarrow 1 = 7$ modules. During a benchmark execution, a module was added, fully expanded, changed, traversed and finally deleted as defined by the Merlin Benchmark run (c.f. Pages 50–52).

The charts are divided into two parts. The white part represents the average time for executing a benchmark operation divided by the number of increments the operation is applied to. For `CrExpOpe`, for instance, the operation time was divided by 17 since `CrExpOpe` creates 17 new operation templates. The grey part depicts the amount of time used for the successive transaction commit.

As Figure 4.9 suggests, those parts of the structure-oriented command execution in a syntax-directed tool that updates a syntax graph, can be executed in Oracle in less than 200 milliseconds. Committing the effect of such a command can be done in less than 300 milliseconds. The two transitions `CommandExecution` and `TransactionCommit` of Figure 3.4 on Page 31 can be performed together in less than 350 milliseconds.

Figure 4.10 depicts the impact of database size on update operations. The figures displayed there have been obtained from benchmark executions on a large initial database containing 8 levels, i.e 255 modules. As Figure 4.10 suggests, the execution times for performing the two transitions `CommandExecution` and `TransactionCommit` do not increase significantly, i.e they remain below 450 milliseconds. Now consider the performance of the first operation, i.e. `CreModul`. It requires about 200 milliseconds and is by far the slowest. This is because the

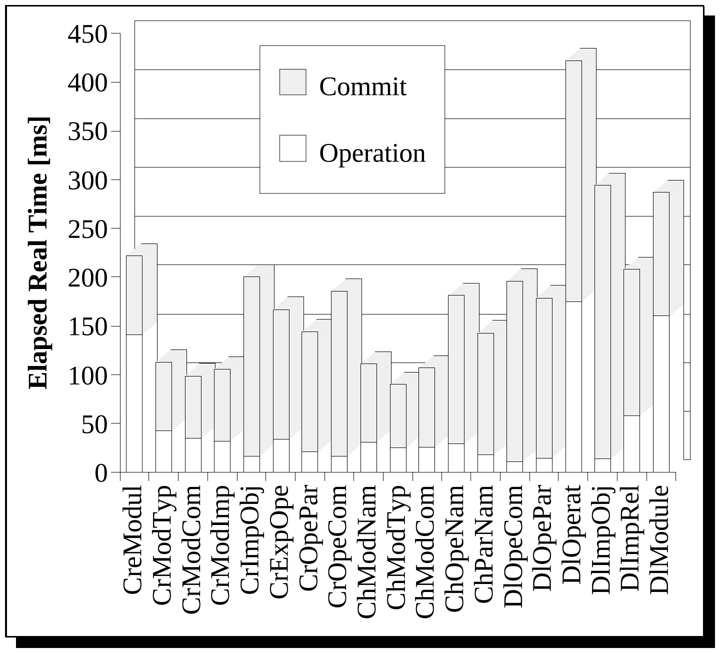


Figure 4.10: Oracle Update Operations in a Large Database

operation was executed immediately after database login where all system caches were empty. Hence, the database monitor had to transfer a number of pages into the caches. Later on, their availability in the cache improved the performance of successive operations significantly. The performance figures of operations `DlOperat`, `DlOpePar`, and `DlImpRel`, which delete elements from the beginning of lists are also interesting. These operations require significantly more time than other update operations. This is because deleting an element from a list requires all other elements in the list to be updated in order to adjust the position attribute. We thus found support for our concern regarding ordered multi-valued edges in the relational model.

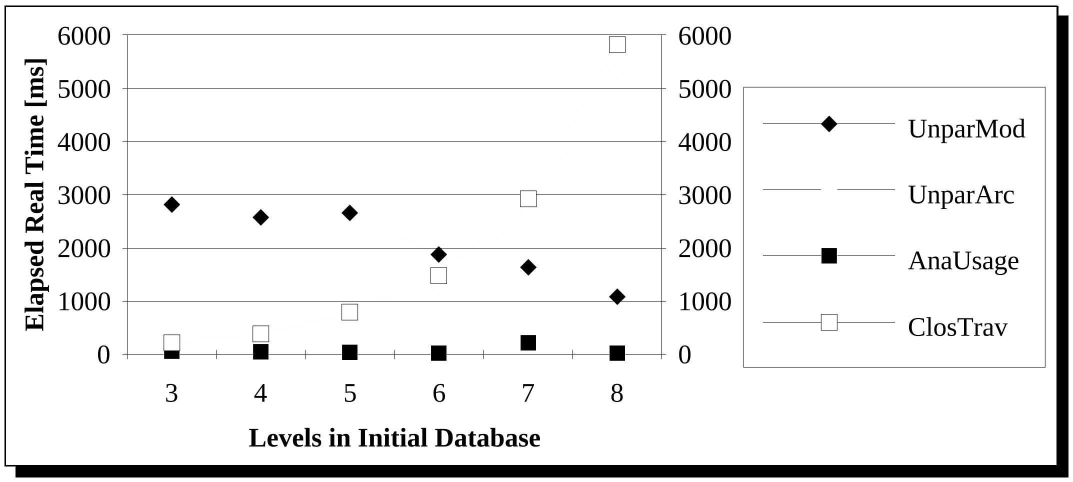


Figure 4.11: Dependency of Traversal Operations on Database Size

To see whether Oracle meets the performance requirement of less than a second for a complete command execution, the transition `Unparsing` of Figure 3.4 also has to be considered. This transition is simulated by the two traversal operations `UnparArc` and `UnparMod` whose execution times are depicted in Figure 4.11.

As suggested by this figure, the performance of these two traversal operations is significantly slower than the performance of update operations. The performance of `UnparMod` varies between 2,600 milliseconds and 1,300 milliseconds. It is not completely clear to us why the performance improves for larger databases. It may be due to a change in the query optimisation strategy when large databases are queried. The performance of `UnparArc`, however, increases linearly with the number of modules contained in the initial database. This is what we had expected because the operation has to traverse through all the modules stored in the database. The performance of this operation for a large database is almost `6,000` milliseconds. The performance of Oracle with these traversal operations is intolerably slow. If unparsing alone takes such a long time, execution of structure-oriented editing commands will hardly ever be completed within a second. The reason for this weak performance is not specific to Oracle, but will also be found in all other relational databases. It is because traversing the syntax graph requires a number of select and join operations, and a join, in particular, is rather time consuming.

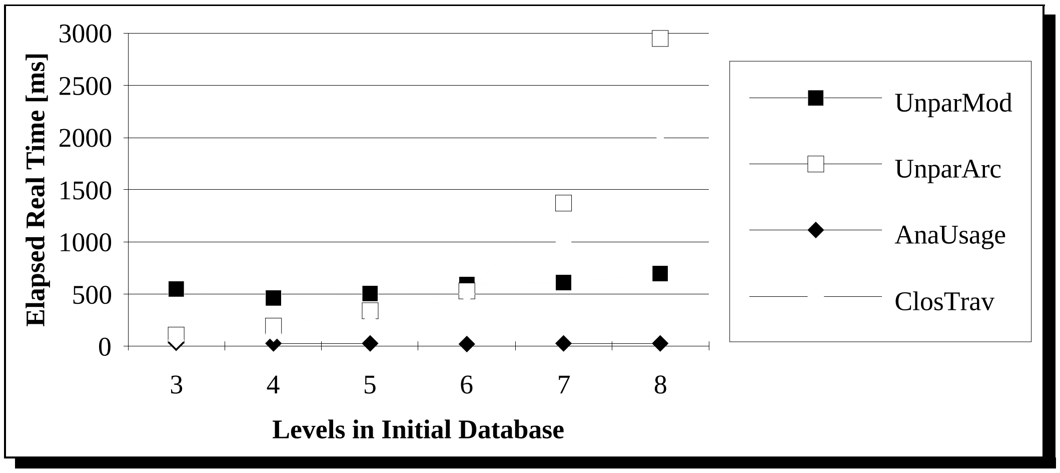


Figure 4.12: Traversal Operations on a Denormalised Schema

[Sch94], therefore, implements the Merlin Benchmark schema using a second strategy. It is based on materialising the transitive closure of syntactic father/child relation. The schema is defined as suggested in [ABJ89]. Therefore, all entities that implement non-terminal nodes are stored within a single table. Post-order and index number attributes are used to represent the father/child relationship. All navigations to non-terminal nodes can then be implemented by a single select operation. A single join operator is required for traversing to terminal nodes. The schema, however, is denormalised to first normal form because post-order and index numbers are non-key attributes that do not depend on the primary key, but on other post-order and index numbers. Using this strategy, [Sch94] was able to reduce the time required for unparsing by a factor of two as displayed in Figure 4.12. For an interface definition, the sum of the times required for operation execution, unparsing and transaction commit is then below a second. For architecture documents it remains below a second as long as fewer than 64 modules with fewer than 255 import relations have to be accessed at a time.

This approach, however, has a number of serious implications. The first is that schema denormalisation causes update anomalies (c.f. [Dat86], Pages 386–388) and update operations perform about 50 % slower. The reason is that post-order and index numbers of other tuples might have to be recomputed when a new node is inserted. Secondly, the schema is completely unstructured and **very** hard to understand and maintain. Finally, the table for tuples representing non-terminal nodes contains a lot of attributes that are meaningless. These attributes waste a significant amount of disk space. Figure 4.13 displays a comparison of the two approaches with respect to disk usage. Storing the initial databases in a denormalised form requires more than four times the space needed to store them in third normal form.

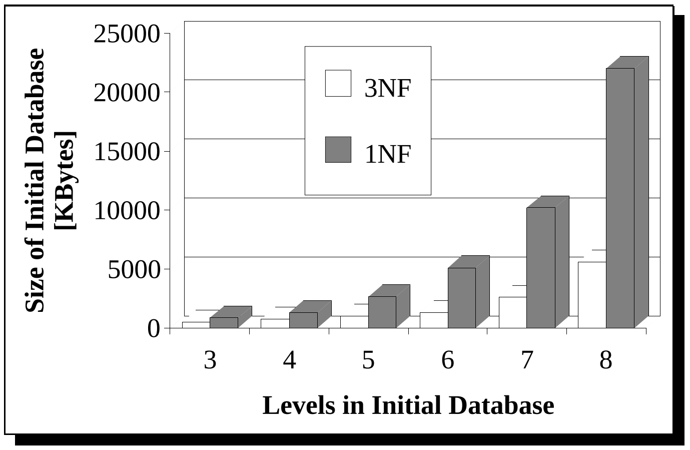


Figure 4.13: Disk Space Used by Oracle

Distributed Databases

All commercially available RDBS products support a client/server architecture. In these architectures, queries are executed on the server. The architectures are thus server-oriented. The servers use operating system files for storing their pages. These files could be stored on remote file servers. The architecture would then be a multi-level server-oriented client/server architecture. Syntax graphs managed by an RDBS could be accessed and modified from different workstations.

During the last 15 years a lot of research has gone into the management of distributed databases. The main issues that have been addressed are distributed database design [CPW87], distributed query processing [SY82], distributed concurrency control [BG81] and deadlock management [Obe82].

Few of these theoretical results, however, have emerged as commercially available RDBS products. One reason for this is that many of the suggested algorithms are NP-complete and the heuristics suggested to cope with this complexity make particular assumptions about the characteristics of data that do not hold in general. Only recently, did Oracle Inc., for instance, release version 7.0 of its RDBS product that is, in a limited way, capable of managing distributed databases. This product release can transparently access and update tables managed by remote monitors.

We, however, required distribution of subgraphs. To have distributed subgraphs managed by different monitors, partitions of tables must be distributed. This is because nodes of the same type may occur within distributed subgraphs. Then different partitions of the same table have to be managed by different monitors. This cannot be achieved by storing these partitions in separate tables since then distribution would not be transparent to tools. Tools would have to have knowledge about the actual subgraph distribution over different tables because any embedded SQL statement must declare the name of the table it should apply to. Moving a subgraph from one monitor to another would then require changing the corresponding tool, for tuples would now be stored in logically different tables.

Administration

All above mentioned RDBS products support a PSDE administrator in maintaining databases. They provide facilities for dumping databases to backup media and recovering a database from a backup. Some of them even maintain a redo-log which they use for bringing the database back to the state of the last completed transaction. This, of course, only works if the log was not damaged during the disk crash. It is not clear to us whether there are RDBSs that support replicated databases. Oracle, however, does not.

Interfaces to RDBSs

RDBSs offer ad-hoc query facilities which can be used by a tool builder in order to browse through the schema, as well as through the tables that implement abstract syntax graphs. Despite the conceptual gap between graphs and tables, the facilities provided can be considered as sufficient for tool construction.

The programming interface to RDBSs is implemented by *pre-compilers* that translate embedded SQL statements into a host programming language which invokes procedures from a library provided with the RDBS. Embedded SQL is an ANSI standard [Dat89]. Moreover, RDBSs provide a low programming interface (e.g. the Oracle call interface). This interface can be used to directly access the contents of tables.

Data transfer between an RDBS and programming languages is based on *host variables*. A host variable is a variable of the host programming language. Its type is either atomic and, therefore, compatible with an atomic type of the RDBS or a complex type declared within the RDBS library. Then it implements tuples or tables. Host variables may then be used within embedded SQL statements. Oracle's pre-compiler, however, does not check for type compatibility between host variables and their use within embedded SQL statements. This is inherently unsafe and may easily lead to run-time type errors.

The result of a relational query is, in general, a set of tuples. RDBSs provide the concept of *cursors* to iterate through a set of tuples. A cursor is opened while declaring the query in embedded SQL. After query execution, the cursor can be used to fetch one result tuple after another into host variables which may then be examined further. Unfortunately, RDBSs do not support the declaration of nested cursors. Nested cursors are required to traverse through nested graph structures. To implement this kind of traversal, the cursor iteration must be completed while intermediate results are stored elsewhere. Then the cursor must be declared again using the stored information.

Summary

RDBSs only partly meet the requirements imposed by PSDE tools on a DBSE. The schema definition language can only express the structure of syntax graphs, but not the way they are accessed and modified. In addition, tables are not an appropriate formalism for declaring the structure of attributed graphs, since edge types cannot be defined and ordered, multi-valued edges cannot be expressed. Relational views cannot be used in syntax-directed editors due to the view update problem. Schema updates are possible, but the tool builder performing such an update is in charge of re-establishing integrity of data in changed tables manually, i.e. without any help from the RDBS. Version management is not supported at all by RDBS. All RDBSs meet our requirement to support ACID transactions. Activities, however, are not supported. The performance required by PSDE tools can be achieved by RDBSs only if the schema is denormalised. As an immediate consequence the schema is completely unstructured and disk space is wasted. RDBSs support distribution using a server-based client/server architecture. The distributed database facilities cannot be exploited due to the fact that tables have to be partitioned. Database administration is well supported. Finally, RDBSs provide programming interfaces to various programming languages. Their implementation, however, must be questioned from a software engineering point of view, for they do not ensure type safety during query execution.

4.3 Structurally Object-Oriented Database Systems

To overcome the deficiencies of RDBSs, a high number of non-standard database systems have been developed. The distinguishing feature of this class of systems is that they enable types of objects and different kinds of relationships among them to be defined. We, therefore, call them, in accordance with [Dit86], structurally object-oriented database systems (SODBS).

Unlike RDBSs, SODBSs are rather heterogeneous. They have no common data model, do not have a common query language and also their programming language bindings differ significantly. We, therefore, cannot review this class of systems as a whole, but have to consider archetypical representatives instead.

Among SODBSs there are systems that focus on efficient management of graphs consisting of attributed nodes and edges. GRAS [BL85], PGraphite [WWFT88], Cactis [HK88] and Adage [GRDM90] are representatives of this subclass. All these systems are more or less stable results of research projects. We have chosen the GRAS system since it is available in the public domain (`ftp.informatik.rwth-aachen.de:/pub/packages/GRAS`) and is still being improved and maintained. This review will be presented in Subsection 4.3.1.

A second class supports data models based on extended entity relationship approaches. Representatives of this class include Damokles [DGL86], DASDBS [SW87], PCTE/OMS [GMT87], PCTE+ [ECM89], ECMA-PCTE, CAIS [AJPO88] and PCIS [Spe92]. From this class of systems, there is currently only a PCTE/OMS implementation available as a commercial product, which we can use for practical experimentation. We have, therefore, selected this one as a representative of this class of systems. PCTE/OMS will be reviewed in Subsection 4.3.2.

4.3.1 GRAS

GRAS was developed during the last decade [BL85, LS88, KSW92] as a database system dedicated to the development of SDEs with syntax-directed tools. In particular, it served as a basis for constructing the IPSEN environment [Nag85].

Data Definition and Manipulation Language

The GRAS data model is based on graphs. A GRAS database is called a *graph pool*. It consists of multiple named graphs. Each graph, in turn, consists of a number of attributed nodes and edges. This data model has been explicitly defined for storing abstract syntax graphs. The nodes have three predefined attributes. The first one is used for storing unique node identifiers determined by GRAS upon node creation. The second attribute is used as a node label which can be used during queries (for a discussion of queries in GRAS, we refer to the discussion of the programming interface on Page 70). A tool builder may freely decide on the use of the third attribute. GRAS considers it is as an arbitrary long sequence of bytes. Edges may lead from a source node to a target node and have an attribute that stores an edge label. Multi-valued edges of any kind, i.e. edges with multiple source or target nodes, are not supported by GRAS. Neither are edges that have source and target nodes in different graphs supported. When we discuss the multi-user capabilities of GRAS, it will become clear that this is a serious drawback.

GRAS does not have a DDL or a DML. Consequently GRAS databases do not have any schema to store type information. Graph access and manipulation operations can only be defined in a host programming language to which GRAS provides an interface (c.f. Page 70). Due to the lack of a schema, however, GRAS cannot check for type compatibility of those operations with graph structures that are actually stored in a graph pool. As an example, GRAS has no means to prevent a tool from creating an edge between two nodes which does not conform to the structure of an abstract syntax graph. If the same or another tool then traverses this edge and expects a target node of a different type, a run-time error cannot be averted. Another example is that attribute information is stored without any type information. If for instance a tuple is to be stored in a node attribute, it must be converted into a sequence of bytes. If it is read, the inverse conversion must be applied to reconstruct the tuple. If, for whatever reason, this is done in the wrong way, subtle errors will occur.

Views

GRAS does not encompass view definition capabilities.

Schema Updates

As there is no schema in GRAS, there is also no need for schema updates. Accommodating a change in the structure of a graph, however, is much more complicated than in RDBSs. To implement a change in a graph's structure, all tools accessing and modifying the graph must be changed. Unlike RDBSs, where a pre-compiler checks to some extent for conformity of the changed tools to the changed schema, changes of tools that use GRAS must be done without any support from GRAS. To convert existing graphs to a new structure that is defined by the changed tools, a separate program must be written.

Versions

The approach to version management taken in GRAS is based on the notion of deltas [Wes89a]. A delta is a sequence of graph storage operations that, if applied to one version V_i of a graph, yields another version V_j of the same graph. GRAS maintains forward and backward deltas in forward and backward logs in order to support arbitrary navigations within a graph's version history.

There are two drawbacks to this approach. Firstly, it is bound to the notion of a graph. The GRAS database system determines the granularity for versioning to be a complete graph, i.e. a "*typical engineering document*" [KSW92]. If smaller granules, such as procedure definitions or sections of a technical documentation, must be versioned, they have to be stored within single graphs. Then, however, our concern regarding missing support for inter-graph edges is immediately reinforced. Secondly, two versions of a graph cannot coexist, because to obtain another version a backward or forward delta must be applied to a version. This, in turn, destroys the former version. Therefore, GRAS cannot support different developers working concurrently on different versions of a graph, which was one of our main rationales for requiring versions.

Transactions

Graphs can only be accessed if they have been opened before. A parameter of the operation to open a graph is an *access mode*, which can be `read` or `write`. The open operation then locks the graph in the given mode. Read locks are compatible with each other, but write locks are neither compatible to read nor to write locks. If a graph to be opened has been locked in an incompatible mode already, execution of the open operation is delayed or even rejected. In order to achieve reasonable concurrent development, those subgraphs of a project-wide abstract syntax graph whose nodes represent documents and, therefore, are accessed together should be stored within separate GRAS graphs. Again, this reinforces our concern about missing support for inter-graph edges, since then inter-document reference edges cannot be implemented using GRAS edges. In addition, the concurrency control mechanism is too restrictive, since the locking of graphs, rather than the locking of nodes unnecessarily, rejects a considerably high number of possible access schedules. While one tool modifies a particular part of a subgraph, another tool may well access a different part of the same subgraph. These schedules are inhibited by the locking protocol of GRAS.

GRAS supports the notion of graph transactions. A set of operations accessing or modifying **one** graph can be clustered to a transaction. These transactions may even be nested. The effect of a transaction becomes persistent as soon as the outermost transaction has been successfully completed with a commit operation. These transactions ensure atomicity and durability of operations on one graph. However, GRAS does not support the grouping of a set of operations that access and modify nodes and edges in different graphs, for example to perform an inter-document consistency check.

Performance

For the implementation of the Merlin Benchmark with GRAS, we decided to store those subgraphs that implement architecture, or module interface documents respectively, within separate GRAS graphs in order to allow the concurrent development of documents. Due to the absence of inter-graph edges, we implemented reference edges between nodes of two graphs with attributes. The node identifier of the target node of an outgoing inter-graph edge is stored with the graph name in an attribute of the source node. Vice versa, the graph name and node identifiers of incoming edges are stored as attributes of the target node. To create or delete these edges, however, the target graph must be opened, the respective attribute modified and the graph closed afterwards.

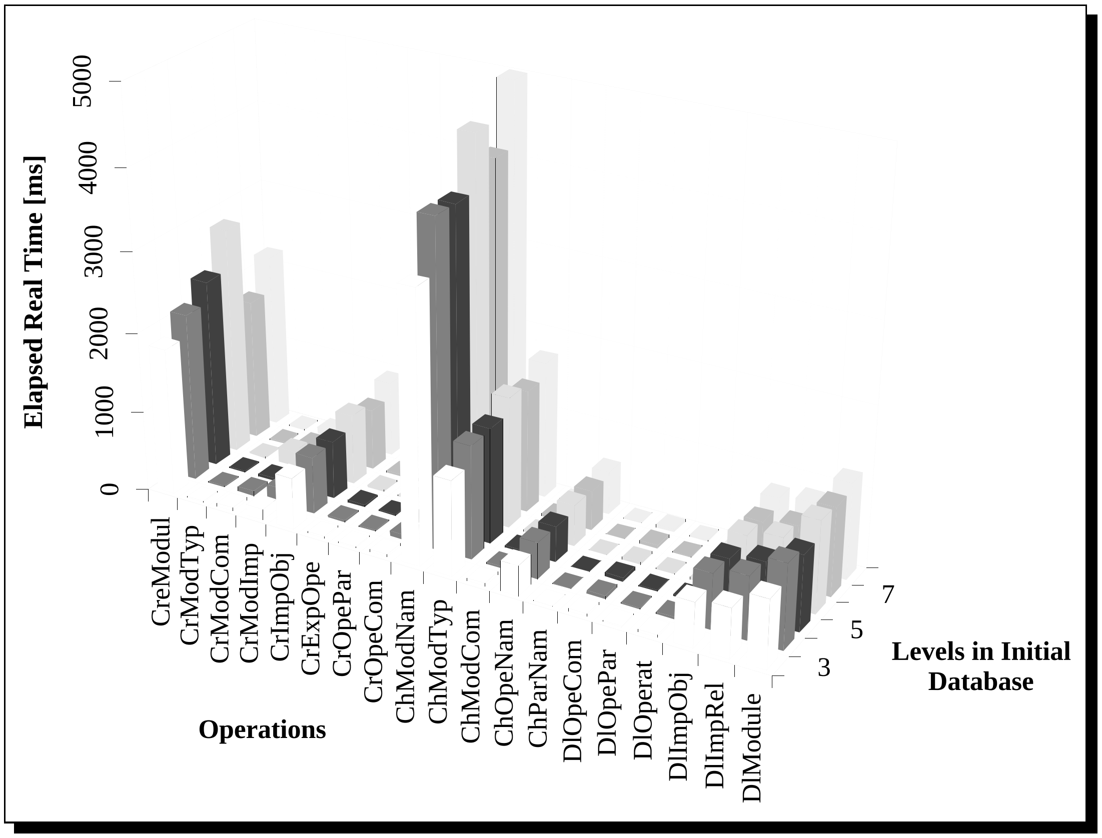


Figure 4.14: Update Operations in GRAS

Figure 4.14 depicts the dependency of execution times of update operations on the database size. It is worthwhile noting, that the performance of these operations only slightly decreases when the database grows. As these operations have to access and modify multiple graphs, we could not use the transaction mechanism provided. Thus a bar only represents the times GRAS required for executing the benchmark operations. A number of operations perform very fast, i.e. in less than 50 milliseconds. These are the very operations that only modify nodes and edges in graphs that have been opened already. The other operations either create, open, delete or close a graph and perform significantly slower. `CreModule` creates a new graph for a module interface document. It establishes the graph name as the name of the module in order to allow for an association with the module node in the architecture graph. `CrImpObj` creates an import and, therefore, must create an inter-graph edge to the graph where the respective type or operation is exported. `ChModNam` has to perform a number of time-consuming operations. Not only must it rename the graph, but also perform a change propagation to those nodes in import lists of other graphs that represent an import relationship to the node. It has, therefore, to open and close a set of other graphs. For the same reason, `ChModTyp`, `DlImpObj`, `DlImpRel` and `DlModule` have to open and close other graphs. In addition `DlModule` has to delete the graph from the graph pool. The overall performance of these operations, must be considered far too slow.

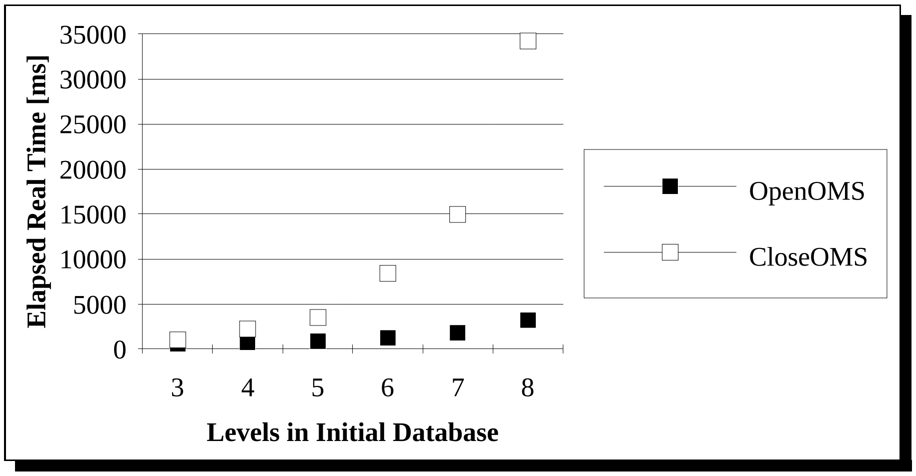


Figure 4.15: Performance of Open and Close Operations of Graphs

It has become clear that the performance of operations modifying several graphs is determined by the performance of opening and closing graphs. Figure 4.15 investigates these operations in more detail. During the benchmark operation `OpenOMS`, the only GRAS operation is to open the architecture graph. In `CloseOMS` nothing else is done but closing this graph. As shown in Figure 4.15, the performance of opening and closing a graph depends on the database size, i.e. the number of nodes and edges in the architecture graph. For a three-level database there are 162 nodes and 315 edges in this graph. For an eight-level database about 5,200 nodes and 12,500 edges have to be maintained in the graph. Although the performance when opening the graph increases while the database grows it remains below three seconds. Closing a large graph, however, takes longer than half a minute. The concurrency control protocol imposes the need to open and close graphs frequently. As they may well be as large as this architecture graph, the performance of closing a large graph is unacceptably slow.

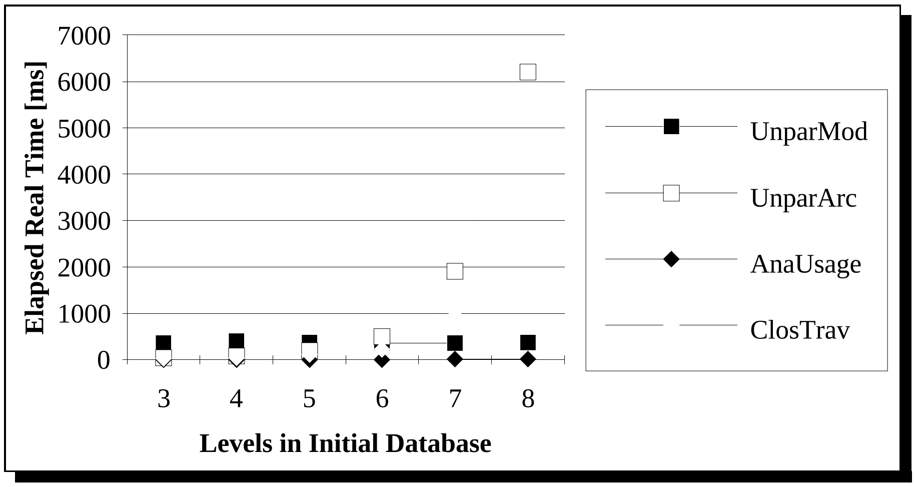


Figure 4.16: Traversal Operations in GRAS

Figure 4.16 displays our performance measurements for traversal operations. The operation that unparses a module interface is independent of the database size and performs in about 350 milliseconds. This is very fast. The execution time for unparsing an architecture document, however, increases polynomially with the number of modules in the architecture. If the architecture contains less than 90 modules, the elapsed real-time remains below a second. For an architecture containing 256 modules, unparsing is as slow as in Oracle with a schema in third normal form.

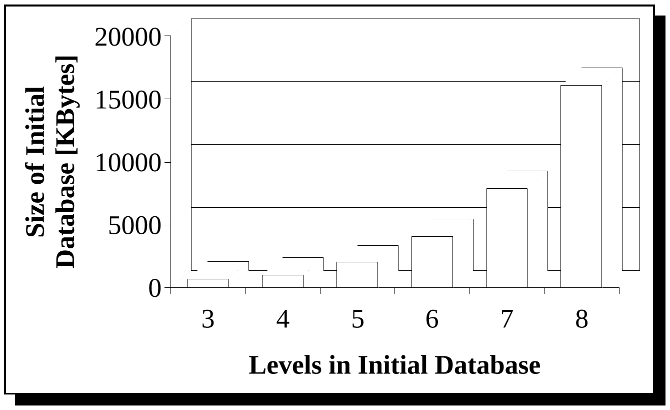


Figure 4.17: Disk Space Used by GRAS

Figure 4.17 elaborates on the disk space utilisation of GRAS. The disk space increases linearly with the number of modules in the architecture. The ratio is about 63 KBytes per module. This is about three times as much as the space required when using Oracle with a schema in third normal form and about 50 % less than Oracle with a denormalised schema.

In short, the performance requirement of less than a second for command execution in the structure-oriented mode of editing is only partially met by GRAS. If the tools did not have to open and close graphs, the command as well as the successive unparsing of a medium-sized document could be done in less than 400 milliseconds. If only one other graph was to be opened, the tool command would not be executed within a second. Together with the lock protocol, this restricts the use of GRAS to single user environments.

Distribution

GRAS supports distribution of graphs with a client/server architecture. Multiple clients may use multiple servers for any access or modification of a graph. A dedicated control server keeps track of the association between opened graphs and graph server processes in such a way that any request of a client is routed to the server in charge of accessing the respective graph. The client/server architecture implementation is implemented by remote procedure calls (RPCs): each creation, deletion of a node or edge, each read/write operation of an attribute and each edge traversal is sent by a client side communication interface via RPC to the server side communication interface and is then handled on the server. Thus the architecture can be considered as a server-based client/server architecture.

We doubt that this architecture is efficient enough to be used for the implementation of distributed tools in a PSDE. Unfortunately, we cannot prove our doubts using the Merlin Benchmark since the implementation of the client/server GRAS version has not yet been completed. Our main concern is that the execution of a single RPC on currently available workstations and network facilities takes about 4 milliseconds. Considering that unparsing a document (about 500 nodes) takes at least 500 graph operations this causes an RPC overhead of 2 seconds.

Administration

GRAS does not provide any explicit support for administration, but relies on services provided by the host operating system. This is unfortunate, since a PSDE administrator must, therefore, know how GRAS maps graph pools to operating system files. To a certain extent this knowledge must be rather detailed, as wrongly made modifications to some of these files can easily corrupt the whole database. Moving these files to another directory of the file-system, for instance, corrupts the database.

Interfaces

A graphical graph browsing facility, which allows a tool developer to investigate an existing graph, is available. As GRAS does not have a query language, it does not offer a user interface for ad-hoc queries.

GRAS offers programming interfaces for Modula-2 and C. They contain operations for the following purposes:

- graph pool operations,
- graph operations,
- partial match query operations (PMQs).

The programming interfaces offer operations to create or delete graph pools and to create or delete graphs within a graph pool. Moreover, they provide operations to open and close a graph identified by a unique user-defined graph name. To access and manipulate these graphs, the programming interfaces provide a number of further operations:

- Creation and deletion of labelled nodes,
- Creation and deletion of labelled edges,
- Assignment and retrieval of attributes to/from nodes.

During creation of a node, GRAS assigns a unique object identifier to a node, which may, in turn, be used for efficiently accessing the node. In addition to unique object identifiers, applications can associate unique external names to nodes. These names enable an associative search to be made for nodes in a graph. GRAS provides the means to perform *partial match queries* in order to implement graph traversals. A partial match query is specified by two components: a node identifier and an edge label. Using partial match queries, a tool can retrieve a single node or a set of nodes that are connected to or from the node by an edge with the given label.

Summary

GRAS does not support the definition of structures of syntax graphs in a schema. In fact, tools can access and modify graphs stored in GRAS however they like. To overcome this, a dedicated graph specification language called PROGRESS [Sch91a] has been defined that is capable of expressing structure and operations on syntax graphs. A generation facility has been constructed that maps PROGRESS specifications to procedures for accessing and modifying GRAS graphs. The use of GRAS is then partly transparent to a tool builder. PROGRESS will be discussed with other tool specification languages in Section 6.6. Different views or updates of a schema are not supported. Versioning of graphs is supported based on delta storage techniques. Concurrency control is based on locking graphs, which we consider as the wrong granularity. GRAS performs very fast on medium-sized graphs that need not be opened or closed, i.e. when GRAS is used in a single-environment. A client/server architecture for GRAS has been proposed, but not yet been implemented. We doubt that the proposed architecture can achieve the performance required due to the way RPCs are used. GRAS does not support a PSDE administrator with dedicated administration tools. Programming language bindings are provided for Modula-2 and C. In short, GRAS seems to be suited to tools in a single-user environment, but not for construction of tools for process-centred software development environments.

4.3.2 PCTE/OMS

The *portable common tool environment* (PCTE) was developed within an ESPRIT-I[6] project as a framework to support SEE construction. The framework has been enriched with *PCTE added common tools* (PACT) [Tho89] in a further ESPRIT-I project. PCTE consists of components for process execution, input/output, inter-process communication, emulation of UNIX system calls, user interface management and an *object management system* (OMS) [GMT87]. All components have standardised interfaces so that SEEs that solely rely on PCTE services are easily portable between different hardware and operating system platforms. While most of the provided services (e.g. the user interface or inter-process communication) have not been accepted since more powerful standards have become available (e.g. X11 with OSF/Motif or OSF/DCE), PCTE's OMS has gained substantial recognition.

Data Definition and Manipulation Language

Schema definition in PCTE/OMS is based on an extended binary entity relationship model. The model is reasonably well suited for defining the structure of abstract syntax graphs. Node types can be defined as entities and attributes can be attached to them. Edge types can be defined as links. Links can have composition, reference or stability semantics. Inheritance is used to define common properties of node types. All entities have to inherit from a predefined entity `Object`.

A drawback of PCTE's data model is that only a restricted number of predefined types can be used for declaring attributes. It is, therefore, not possible to define types for complex attributes such as error sets or symbol tables. These types have to be implemented using entities and relationships.

In order to implement ordered multi-valued edges, key attributes can be assigned to links of cardinality many. Unfortunately these key attributes cannot be updated. They can only be determined upon link creation. In order to insert an element into a list at a position p, all links leading to elements after p have to be deleted and created with a new (higher) key attribute.

The operations available for abstract syntax graphs cannot be defined within the schema, but must be implemented in a host programming language using one of the standardised programming language bindings. This has a number of serious drawbacks. The programming

[6]ESPRIT is the European Strategic Programme for Research in Information Technology

interfaces offer only a fixed number of operations for an infinite set of schemas. Hence the operations provided by the interface must be generic. This genericity is achieved by passing strings that denote schema definitions as arguments. These strings must then be interpreted at run-time with the consequences that, firstly, no consistency checks against the schema can be made at compile-time and, secondly, performance is decreased due to the required argument interpretation. This is only bearable if the time required for interpretation is small compared to the time spent handling fetched data. This is only the case if coarse-grained objects are managed and again this is an indication that PCTE/OMS has not been built for management of fine-grained objects. A further drawback of these programming interfaces is that information about the schema is widely spread into tools without using a well-defined interface. Hence the data abstraction principle is broken[7].

The overall schema definition can be structured into *schema definition sets* (SDSs). Import clauses make the relationships between different SDSs explicit and contribute to the comprehensibility of schema definitions.

Views

Different SDSs can be used to establish the *working schema* of a tool. The working schema is computed by the superposition of all entity and link definitions provided by the different SDSs. As only those objects, attributes and links whose definitions have been included in the tool's working schema are accessible by a tool, the working schema can be used as a view mechanism for tools.

Schema Updates

A PSDE builder may at any time change definitions in the SDS. Existing objects, attributes and links are then changed accordingly.

Versions

The *version and configuration management common service* (VMCS) developed in the PACT project is capable of managing versions of *composite entities*. A composite entity has a root object, which determines the composite entity as follows. The root object and the set of all objects reachable via composition links belong to the composite entity. All reference links between objects belonging to the composite entity also belong to the composite entity. Therefore, composite entities could be used to implement subgraphs of a project-wide syntax graph and, using VMCS, these subgraphs could be versioned. The operations provided in order to version composite entities are the creation of a revision (the predecessor version is frozen), creation of a snapshot (the successor version is frozen), deletion of a version and traversals through the version history graph starting from an arbitrary object in the composite entity. Freezing a version is implemented by giving the predecessor/successor links the stability property, which means that the object they lead to cannot be updated. As we have argued in Section 3.5, this assumption is undesirable. It will lead to the problem of attributes of increments in a frozen

[7][Tho93] admits this. He argues that in PCTE "Tool Dependency on Schema Properties is Undesirable".

version being unable to be changed during configuration management, for instance to mark an increment as erroneous.

A disadvantage from the performance point of view is that VMCS maintains predecessor and successor links between all objects that belong to a composite entity. This is unnecessary for versions in abstract syntax graphs as predecessor/successor relationships are only required between subgraphs, but not between abstract syntax graph nodes. This is again an indication that PCTE/OMS is intended to be used with coarse-grained objects. Moreover, in the implementations available, utilisation of physical disk space is inefficient, since an eager object duplication strategy is used: All entities of the composite entity are physically duplicated as soon as a new revision or snapshot is derived, regardless of whether they really differ. To change this, the version mechanism has to be integrated with PCTE's transaction management in the underlying OMS. It can no longer be added as a service on top of the OMS.

Transactions

PCTE/OMS supports a concurrency control mechanism with three kinds of activities: *unprotected activities*, *protected activities* and *nested transactions*. Unprotected activities are neither atomic nor do they operate in isolation from other concurrent activities. Protected activities are not atomic, but the concurrency control mechanism guarantees isolation from other concurrent activities. Transactions are both atomic and isolated. For protected activities and transactions, locking is done implicitly. Nevertheless, an application may explicitly lock resources. Therefore, even activities, which have been started as unprotected may lock particular objects and then the concurrency control mechanism ensures that no concurrent activities access these objects in an incompatible mode.

This mechanism does not meet our requirement of atomic, but non-isolated activities. Atomicity may not be an important requirement in activities that access a small number of coarse-grained objects. In the case of a failure, the user knows which objects he or she was currently working on and he or she can reestablish integrity manually. When a high number of fine-grained objects, such as a project-wide abstract syntax graph, are accessed during an activity, however, the user is no longer aware of which objects have been affected by a failure and he or she cannot re-establish integrity manually. Therefore, we consider atomicity to be too important to be given up and consequently the unprotected and protected activities offered by PCTE/OMS cannot be used.

Performance

The Merlin Benchmark has been implemented using version 12.4.1 of the Emeraude implementation of PCTE/OMS 1.5 [NW94]. The data model of PCTE leaves different alternatives for implementing the benchmark schema. It has been defined within one SDS. Entities of the benchmark schema defined on Page 49 have been declared as types in that SDS, aggregation relationships have been defined using composition links and reference relationships have been implemented using PCTE reference links. Reference relationships were also used for implementing the dictionary relationships, since the PCTE data model does not include dictionaries. Benchmark operations were implemented using the C programming language binding. They were executed as PCTE transactions in order to ensure their atomicity.

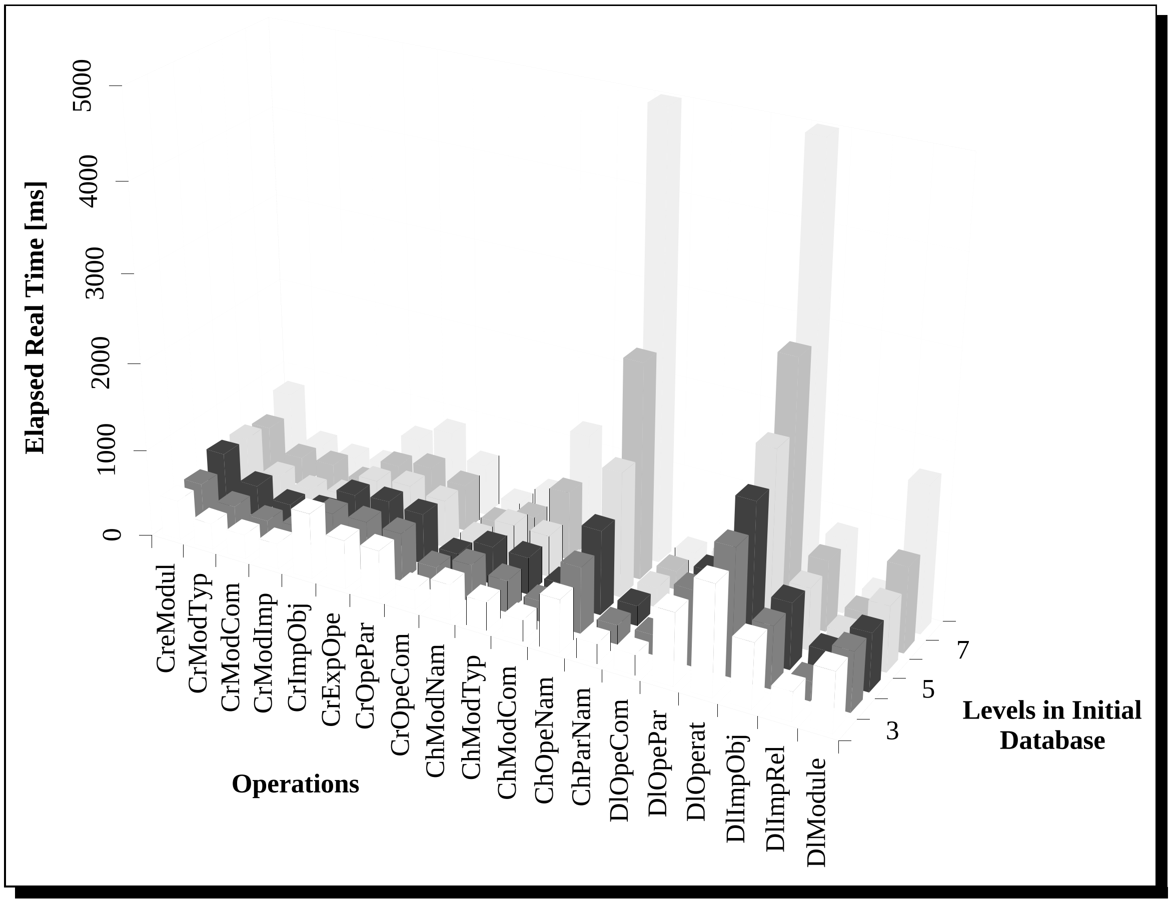


Figure 4.18: Update Operations in PCTE/OMS

Figure 4.18 depicts the dependency of execution times of update operations (including the time spent during the successive commit) on the database size. For some operations the performance significantly deteriorates when the database grows. It deteriorates for those operations that create, update or delete identifiers whose scope is the overall architecture. Due to the lack of the dictionary relationship, these operations had to perform a linear search over all identifiers in the scope, in order to check for uniqueness. Execution times of these static semantic checks thus grow linearly with the number of modules in the database. The times required by `ChOpNam` and `DlOperat` increase considerably as well. The reason for this is that a high number of objects have to be visited to check the uniqueness of operation names.

Figure 4.19 displays the relationship between times required for benchmark operations on an eight-level database including the successive transaction commit. It thus refines the last row of Figure 4.18. It shows that ten of the operations with their successive commits require more than 500 milliseconds. In the case of `ChOpName` and `DeleteOp`, the time even exceeds 5 seconds, which must be regarded as intolerably slow. These figures suggest that PCTE's performance of commit operations for transactions with a high number of write accesses is far too slow.

Figure 4.20 depicts how the times required for traversal operations depend on the database size. Unparsing a module constantly requires about 800 milliseconds. Unparsing an architecture linearly increases up to 5,800 milliseconds for an eight-level database. Unparsing an architecture is faster than a second for architectures smaller than 40 modules.

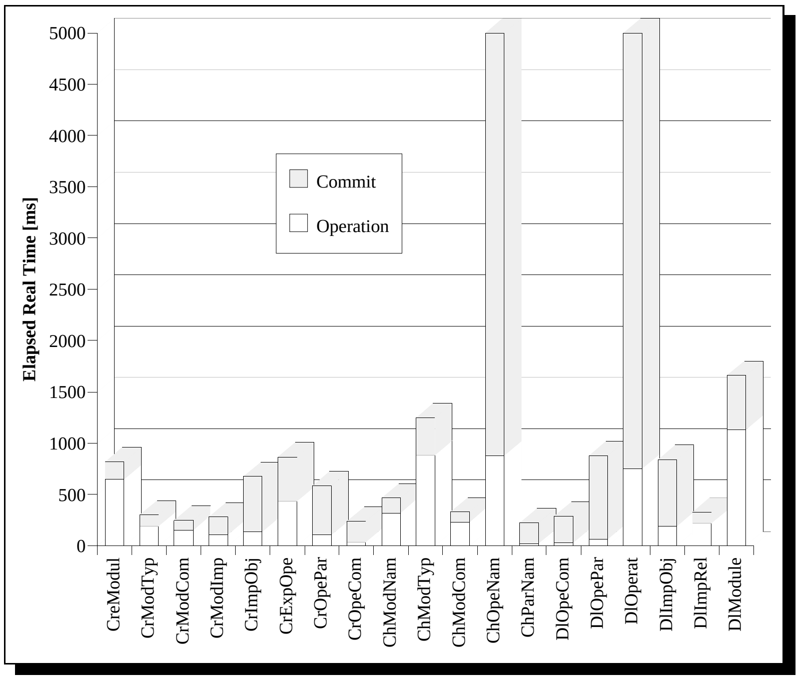


Figure 4.19: PCTE Update Operations and Commit in a Large Database

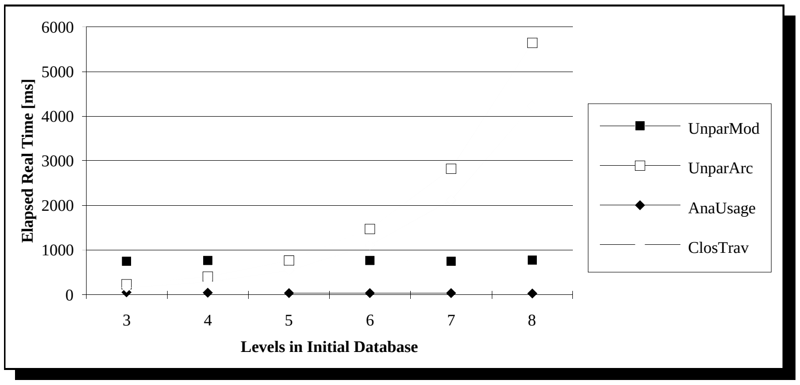


Figure 4.20: Dependency of Traversal Operations on Database Size

Finally, Figure 4.21 elaborates on the disk space utilisation of PCTE/OMS. The PCTE volume used for the benchmark had a fixed size of 25 MBytes. The real space utilisation in the volume has been obtained with a `vol scan` command. The disk space increases linearly with the number of modules in the architecture. The ratio is about 28 KBytes per module. Surprisingly, the utilisation is better than in Oracle with a denormalised schema and also better than in GRAS.

In short, the performance of PCTE/OMS must be considered too slow if it is used for syntax graphs of large size. Only simple operations, such as `CrModImp`, `CreOpCom`, `ChModCom`, `ChParNam` or `DelOpCom` where a small number of objects are modified, will meet the performance requirement of less than a second for an operation, commit and successive unparsing.

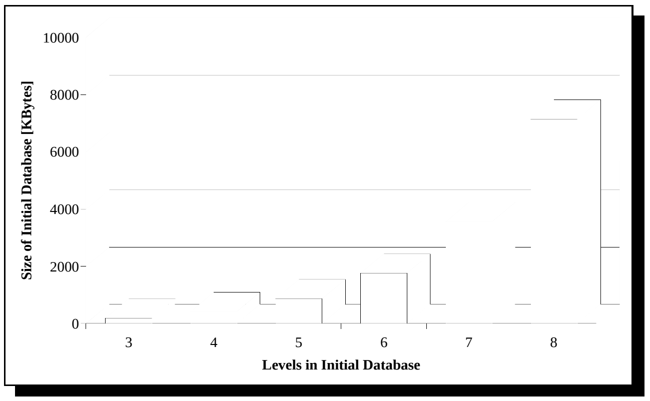


Figure 4.21: Disk Space Used by PCTE

Distribution

PCTE/OMS has a distributed database architecture. Objects managed by PCTE/OMS physically reside on *volumes*. The concept of volumes does not impose any constraints on links. Unlike GRAS, where edges are confined within graphs, links in PCTE may span between arbitrary volumes. For each volume a particular host is selected as the *volume server*. Other hosts may *mount* volumes and, by doing that, provide tools running on that host with transparent distributed access to objects physically stored on remote volumes. Objects that are rarely updated but frequently read may be held as replicated objects on several volumes in order to decrease network traffic or to limit the impact of unreachable volume servers. PCTE/OMS then manages updates to these replicated objects transparently.

Administration

PCTE/OMS provides the PSDE administrator with a number of facilities. A backup tool can be used by the administrator to write composite objects or even complete volumes incrementally onto backup media. A few tools, particularly for starting hosts, mounting or un-mounting volumes and managing data replication, are offered to control data distribution. User administration is done by maintaining a user identification file with standard UNIX tools. Tools for performance monitoring purposes are not available.

Interfaces

The functional PCTE specification is available for two programming language bindings, Ada and C. Besides the problems addressed earlier, these bindings suffer from the fact that they do not meet the requirement of object identification appropriately. Objects stored in PCTE/OMS can either be identified by *path names* or by *reference objects*. Path names are character strings, which define the navigation path from the common root object to the object to be addressed. These paths are unique throughout the whole database. Unfortunately they cannot be used for object identification purposes as they change when a link name on the path changes. Moreover, they occupy substantial internal storage space and objects addressed by path names can only be accessed inefficiently. This is because the OMS must interpret the name by navigating from the common root to the object. As many disk accesses may be required as there are link names occurring in the path name. To deal with this, reference objects have been introduced. These objects are not unique within the whole database, but only valid in the current process or its sub-processes. They are handled in the same way as UNIX file descriptors, which means that only a few reference objects may be declared. Therefore, they cannot be used for identifying several thousand nodes in the portion of a project-wide abstract syntax graph currently displayed.

PCTE/OMS offers two user-interfaces. The first one is a textual shell where every function of the programming interface is available as a command. This can be used for navigating through the database. A graphical SDS design tool is also provided, which enables the schema to be defined in terms of an extended entity relationship model and can generate and compile an SDS from a model.

Summary

PCTE/OMS has been built for the management of coarse-grained objects and is, therefore, only of limited use for managing project-wide abstract syntax graphs. The data definition language lacks facilities for defining attribute types. Moreover, it cannot express ordered multi-valued edges. PCTE/OMS does not have any data manipulation language, thus operations accessing and modifying abstract syntax graphs can only be implemented in a host programming language. The concept of working schemas can be used for implementing views. Schema updates are possible, and PCTE updates any existing objects that have been instantiated from the schema before. Version management of syntax graphs can be implemented with the PACT VMCS though only with significant performance overheads. PCTE offers a concept to implement ACID transactions, but activities are not supported. The performance of the available PCTE/OMS implementation meets the requirements imposed by syntax-directed tools not in all cases. Both distribution and administration are well supported. The programming interfaces provided have difficulty in meeting the requirement of object identification in syntax-directed tools.

4.4 Object Database Systems

In the mid-eighties, a new class of database systems combining object-oriented programming languages with database technology became available. Recently, a standard for this class of database systems has been released [Cat93]. In accordance to the standard definition we call these systems *object database systems* (ODBS). Meanwhile a substantial number of systems such as GemStone [CM84], O_2 [LRV88], Ontos [AHS91], Versant [Gor87], Orion[8] [KBC+89] and ObjectStore [LLOW91] are available as products.

The common features that any ODBS must support were initially defined in [ABD+90]. Besides these mandatory features a high number of extensions such as versions, views, active capabilities or schema updates have also been proposed. Most of them are important for PSDE construction and have been implemented in one or the other ODBS. Among the systems that offer the greatest functionality is the O_2 ODBS since extensions required for PSDE constructions have been added in the GOODSTEP project [GOO94]. As we have this sys-

[8]The product name of Orion was changed to ITASCA. As all of the concepts of the ITASCA product have been built for the Orion system, we consider Orion in this thesis only.

tem available for practical experiments, we present the review of this class of DBSs using the particular functionality offered by the O_2 system and only mention other ODBSs when they have important differences. While doing so, we enhance and refine the arguments presented in [EKS93].

Data Definition and Manipulation Language

To implement project-wide abstract syntax graphs, node types are implemented as *classes* of the database schema. Classes in O_2 are defined in the O_2C data definition language. GemStone provides a dedicated language called *OPAL*, and Orion uses CommonLisp, whereas all other ODBSs use C++ as data definition language. They could, therefore, equally well be considered as persistent C++ programming language implementations. Nodes are represented by instances of these classes then. They are *objects* whose instance variables represent edges and attributes. Navigation along these edges is done by dereferencing instance variables. It is a slight drawback that instance variables only support navigation in one direction. Therefore, whenever navigation in both directions is required, a pair of instance variables must be defined to implement an edge. For implementation of multi-valued edges, type constructors such as lists (if the edges are ordered) or sets (otherwise) are used. Navigation is then expressed in terms of an *object query language* (OQL) or iteration primitives.

The type-compatibility in an O_2C schema is checked at compile-time which, compared with run-time type checking in OPAL, achieves better type safeness and improved performance. The set of target nodes of a particular edge should, therefore, be restricted to those types of nodes that are allowed according to syntax and static semantics of the language. Therefore, we exploit the *type-system* provided by typed ODBSs (such as O_2) to define the types of instance variables as a first step towards type safeness.

For schema simplification, multiple *inheritance* is used to define common properties of nodes such as outgoing syntactic or non-syntactic edges or attributes in a super class, only once. Subclasses of this super class then inherit the definition. Moreover, edges do not always connect nodes of the same type. Heterogeneous edges, i.e. edges leading to different types of nodes can be implemented using *polymorphism*.

Integrity constraints are enforced by *encapsulation*, i.e. applications are not allowed to modify instance variables directly, but must use the methods defined. This form of encapsulation is only made possible by the *computational completeness* of the schema definition language.

In any ODBS, persistence of objects is defined by reachability from a persistent object. The DDL of O_2, therefore, includes the concept of *names* that can be defined in a schema. An object that is assigned to a name at run-time is called *named object*. Named objects are persistent. A tool schema will, therefore, include a name whose type is a set of document root nodes. Then any node in a document's subgraph whose root node is included in the set becomes persistent, because they are all reachable from the root node.

Schemas in O_2 can be structured into different sub-schemas. Therefore, particular sub-schema components can be designated as exports and then other sub-schemas can import these components. Hence the information hiding paradigm is not only applied for single classes but also on a more coarse-grained level for schemas, i.e. sets of classes.

Views

Various view definition facilities for ODBSs [SLT91], [Ber92], [AB91] and [HZ90] have recently been suggested. The fact that they allow definition of different interfaces for the same objects is a feature common to all of them. For the O_2 ODBS the view mechanism proposed in [AB91] has been implemented [SAD94]. As the implementation is available for practical use, we now consider this view mechanism in more detail.

The view mechanism of O_2 allows a tool builder to specify *virtual schemas* and *virtual databases*. A virtual schema definition is based on a conceptual schema called *root schema*. A virtual schema can hide classes defined in the root schema and can modify the interface of classes defined in the root schema. To achieve this modification, virtual classes can be defined on the basis of root class definitions. Objects contained in databases that instantiate the root schema are represented according to the respective virtual class definitions when they are accessed through a virtual schema. We then refer to these objects as *virtual objects*. Therefore, the virtual schema definition implicitly defines virtual databases.

Object-oriented views partly overcome the view update problem of relational databases. As argued in [SLT91] this is due to the concept of object identity. Opposed to relational databases, where the schema designer must designate unique key attributes to address tuples, ODBSs define object identity in a way transparent to the schema designer. In relational databases, views can be constructed that hide primary key attributes of base relations and then these views are no longer updatable. In object-oriented views, virtual objects always store the object identity of their base object. Then it is always defined into which base object to migrate a virtual object update.

The view mechanism of O_2 is particularly suitable for a tool builder for defining different views on a project-wide abstract syntax graph as was required in Section 3.3. As discussed in [Bec95], the overall structure of the graph can be defined in a conceptual schema using O_2's schema definition language. Based on this schema a number of virtual schemas can be defined for tools so that each tool is provided with its own view of the abstract syntax graph. For a class that implements a node type in the conceptual schema, the virtual schema for a tool includes a virtual class that shows only those edges that are of concern for the tool and hides any others. Node types that must not be seen at all can be hidden by not defining a virtual class for this class. Moreover, the virtual schema can hide those methods that implement modifications that ought not be invoked by a tool. It can add additional methods that have not been defined in the conceptual schema for instance to implement different unparsing schemes or different parsers in different tools. Using the view mechanism in this way, a tool builder enables different tools to use different schemas particularly suited for their purposes, while sharing nodes in the project-wide abstract syntax graph with other tools.

Schema Updates

Almost each of the above mentioned ODBSs supports incremental updates to an already established schema. Only ObjectStore, Objectivity, Versant and O_2, however, enable an existing database to migrate to a changed schema. Of these systems, the O_2 ODBS offers the most sophisticated support for controlling migration after a schema update. We, therefore, discuss the problem of schema updates in ODBSs using the particular choices taken in the O_2 system as discussed in [FMZ94b].

In the O_2 system, changes to bodies of methods can be performed without any additional measures. The change is in place as soon as the transaction that performed the change is completed. Updates to signatures of methods can be done as well. O_2 then compiles all depending method bodies anew. The change becomes effective if all depending bodies have been successfully compiled and the transaction that performed the change has been completed. These schema updates do not affect the consistency of existing databases at all.

Unlike changing a method definition, changes to instance variable declarations in classes will affect the database, if it contains objects of these classes. To support such a change and have all existing objects of the respective class migrating to the new class definition, O_2 offers *conversion functions* [HVZ90]. A conversion function is an O_2C function with an input parameter of the old class' type and a result type of the new class' type. After a change, such a conversion function can be associated with the modified class and the database system executes this function for each object of the changed class before it is accessed the next time.

When to execute conversion functions can be determined by the tool builder by choosing between an immediate or lazy strategy. In the immediate strategy all objects of the changed class are converted during the schema update transaction. In the lazy strategy an object is only converted when a transaction is about to access the object. Whichever strategy is chosen, for tools operating on the changed schema the effect is always the same [FMZ94a].

These facilities can be exploited in order to change the structure of an existing abstract syntax graph stored in O_2 consistently. New types of nodes can be introduced by defining new classes. Edges can be added to or deleted from existing nodes by adding or deleting an instance variable to or from the class. If an instance variable is added, a conversion function can determine the target node of the edge. Depending on the number of nodes, immediate or lazy conversions of the existing nodes can be performed.

Versions

Only few of the available ODBSs provide support for versioning at all. ObjectStore and Versant support versions of objects by providing a predefined class from which other classes can inherit the property of being versioned. This can be used to maintain versions of single nodes but is of very limited use for the implementation of versions of subgraphs of the project-wide syntax graph.

Only Orion and O_2 provide support for the versioning of composite objects. In Orion composite objects are defined statically within the schema, whereas in O_2 composite objects are determined in a more flexible way during run-time by including objects in a versionable container object. We, therefore, consider this approach in more detail.

O_2's version manager provides a predefined class `Version` [DM93]. An object of class `Version` acts as a container for a set of objects that are under version control together. The class includes data structures for maintaining the version history of the composite object and methods for navigating through the version history graph. `Version` offers methods to add or remove objects to or from the composite object. As soon as an object is added to an object of class `Version`, this object is under version control. Moreover, `Version` offers methods to set the *current* version of a container, to *derive* versions from the current version and to set *default* versions, i.e. to determine which version of a composite object is to be used if other objects do not address a particular version.

The class `Version` is particularly suited for implementing versions of subgraphs of the project-wide abstract syntax graph as shown in [Bru94]. In the implementation of each node of the abstract syntax graph that represents a document (document node), an instance variable of class `Version` can be added. Then, whenever a node is created, the object implementing the node is added to the container object of the document node to which the node belongs. When a node is deleted, the node is removed from the container object. To derive a new version of a document, the implementation of the document node need only call the derive method of the container object. A similar strategy is chosen for navigation through the version graph of a document, establishing current and default versions and so on.

Transactions

By definition every ODBS supports conventional transactions with ACID properties. These transaction mechanisms can be exploited to group a set of operations that modify a project-wide abstract syntax graph so that they are either completely performed and then persistent or, in the case of a failure, they are not performed at all.

In most of the systems, a PSDE administrator can decide to give up one or the other of the ACID properties statically, i.e. for complete sessions. In GemStone, for instance, the administrator can decide not to perform concurrency control thus giving up the isolation property of transactions. In O_2, the administrator can decide to access databases without logging, i.e. the atomicity and durability properties are abandoned. We have not found any system, however, in which a tool builder could decide for each transaction whether or not concurrency control should be performed. Hence our requirements regarding adjustable transaction mechanisms have not been fulfilled completely.

Performance

The Merlin Benchmark was implemented and executed with GemStone Version 3.0.3 and O_2 Version 4.3. In single-user modes, these systems have shown more or less the same results. We, therefore, only discuss the results of O_2. The performance figures for GemStone are presented and discussed in [EK92].

Figure 4.22 displays the performance of update operations for various initial database sizes. The figures include the time for executing the operation as well as the time needed for the successive commit. The performance of these operations is fairly balanced. Not a single operation required more than 900 milliseconds. Six operations required more than 500 milliseconds. It is remarkable that there is no significant decrease in the performance if the database gets larger. This is because consistency checks were performed with dictionaries that were implemented with external hashing techniques as suggested in [Pea90]. The slowest operation is `CreModul`. This is because the operation was performed at the beginning of each benchmark execution while the database was in a cold state, i.e. the database caches were empty. The impact of database caches becomes evident by comparing the performance of `CrModCom` and `CrOpCom`, which perform the same operation, except that the module comment is created in a colder state and, therefore, takes about 250 milliseconds longer. The second slowest operation is `ChModTyp`, which changes a type and thus has to perform a change propagation to a high number of increments using this type. In short, update operations in O_2 are significantly faster than in GRAS and PCTE, though about 250 milliseconds slower than in Oracle.

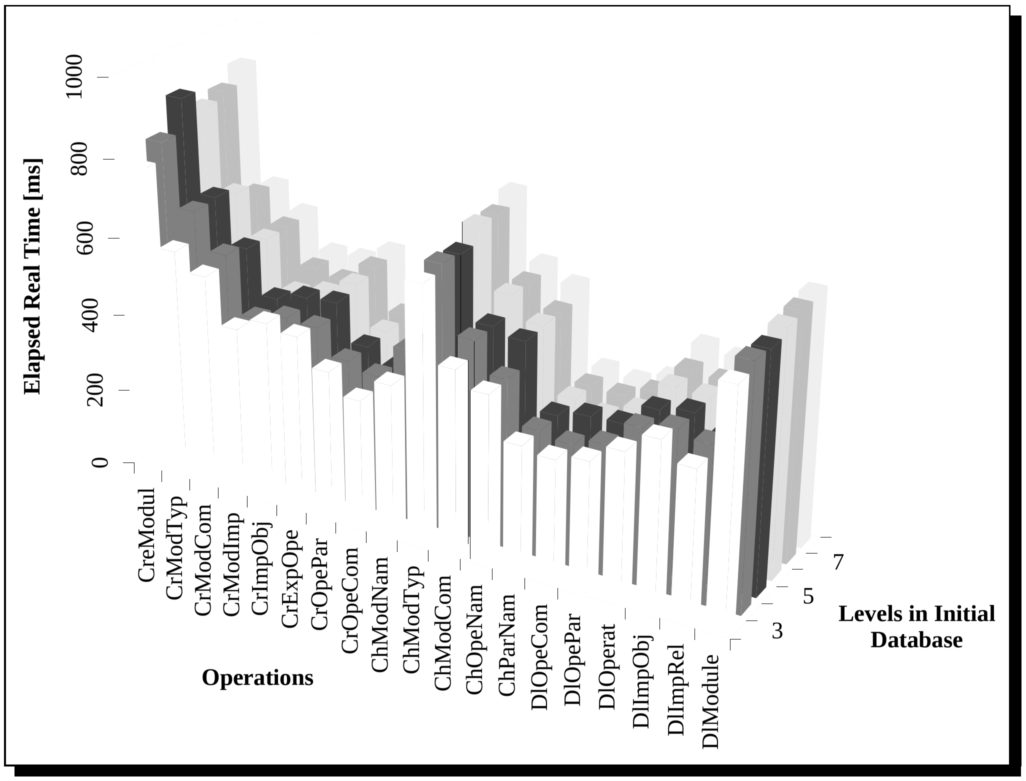


Figure 4.22: Update Operations in O_2

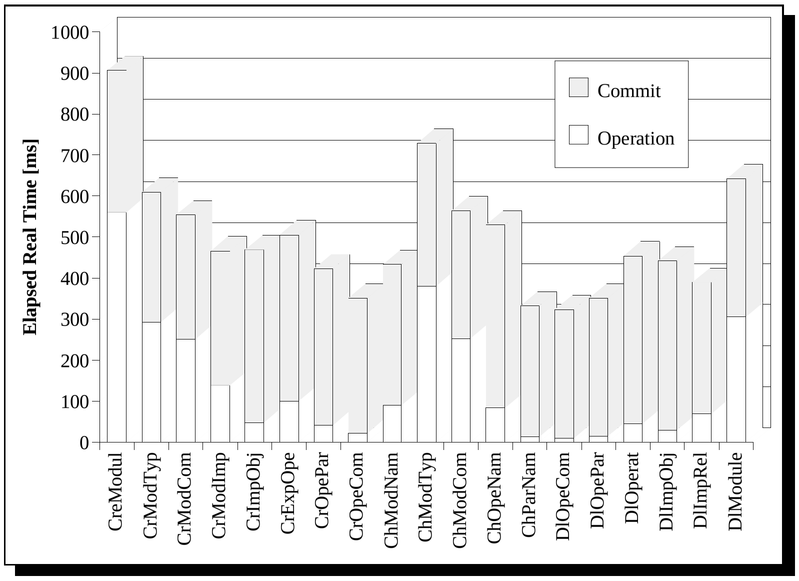


Figure 4.23: O_2 Update Operations and Commit in a Large Database

Figure 4.23 displays the relationships between benchmark operations and commit times for an eight-level database, i.e it refines the last row of Figure 4.22. The times required for a commit are about 350-400 milliseconds. Thus the commit time dominates the performance of most operations. It is also the reason why Oracle is faster. The operations themselves merely perform as fast as in Oracle, but Oracle seems to be optimised to a high transaction throughput with a commit requiring only between 50 and 150 milliseconds.

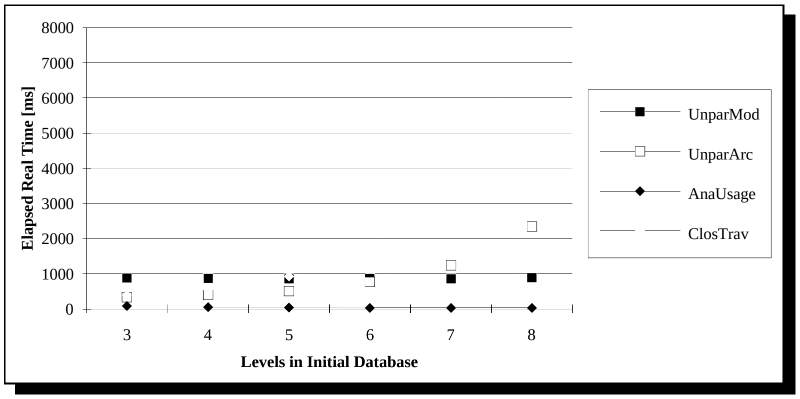


Figure 4.24: Dependency of Traversal Operations on Database Size

Figure 4.24 displays the performance figures for traversal operations. In O_2 the performance of the operation to unparse a module is constantly about 800 milliseconds. It is thus much faster than in Oracle with a normalised schema, as fast as in PCTE and Oracle with a denormalised schema and 400 milliseconds slower than in GRAS. In O_2 it takes 2,300 milliseconds to unparse an architecture with 256 modules and is thus faster than in any other database system. It is more than twice as fast as in Oracle with a normalised schema, in PCTE and in GRAS and still about 800 milliseconds faster than in Oracle with a denormalised schema.

Figure 4.25 displays the disk space utilisation we obtained for O_2. The required disk space for an initial database grows linearly with the number of modules in the database. The ratio is about 95 KBytes per module, which is the worst result of all investigated systems.

In short, the performance of O_2 is reasonably good. If we consider only operations performed in a warm database state (which is the usual situation during editing sessions), performing an operation, the successive transaction commit and unparsing a complete module can be executed in less than 1,300 milliseconds. The heavy disk space requirements of O_2 are clearly a drawback, but we assume that due to advances in secondary storage devices, larger and faster disks will become available at lower costs. Thus disk space will be a less important factor in the future.

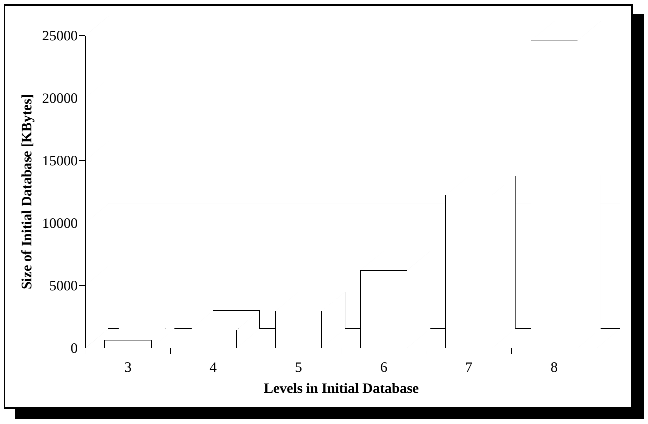


Figure 4.25: Disk Space Used by O_2

Distribution

All ODBSs offer distributed access to a database by a client/server architecture. We found client-based and server-based architectures. O_2 and ObjectStore are client-oriented, whereas GemStone is server-oriented. In experimentations with both the client-oriented and the server-oriented architecture, we found that the client-oriented architecture is more tolerant of a higher load. This is because most of the computation load for executing abstract syntax graph operations is taken by the clients.

None of the ODBSs we have looked at is capable of managing distributed databases. As long as this is the case, ODBSs cannot be used in large projects.

Administration

In GemStone and O_2 the following facilities are offered to the database administrator:

- incremental backup facilities for storing databases on secondary storage media. Gemstone even supports replicated databases,
- secondary storage management facilities in order to monitor physical disk space utilisation, and
- database maintenance facilities in order to improve the fragmentation of the database and to perform garbage collections.

In short, these facilities can be considered sufficient for administering project-wide abstract syntax graphs in PSDEs.

Interfaces

All of the ODBSs mentioned in the introduction to this section have a programming interface to C++. In most of the systems, C++ is even used as schema definition language in the sense that the schema is derived from a set of C++ class definitions by a preprocessor. These systems then inherit the unsafeness from C++, which we consider a severe drawback.

Only in GemStone and O_2, are dedicated languages, namely OPAL and O_2C, used and a schema that has been established in one of these languages can be exported to C++. This export mechanism generates a twin C++ class for each class defined in the schema. Objects of these classes can then be used as object identifiers in the programming language. In O_2, even O_2C method definitions can be exported to C++. The export mechanism then generates methods, with the same name and the same parameter list as in the respective O_2C class for each object identifier class. The methods defined in the schema can then be called using these generated methods and the exported object identifiers encapsulated according to the schema definition. These object identifiers can, therefore, in general be used for the association between increments and nodes of the abstract syntax graph.

The user interfaces offered by GemStone and O_2 provide the database user with facilities for:

- incrementally creating, modifying and deleting schema definitions,
- performing ad-hoc queries and browsing through the schema definitions,

- debugging schema definitions,
- browsing through the objects contained in the database, and
- creating, modifying and deleting objects in the database.

In particular, these facilities support a tool builder during the tool constructing process.

4.5 Summary

Our detailed investigation of different kinds of database systems revealed that relational database systems should not be used as DBSE because their data model is inappropriate for expressing project-wide abstract syntax graphs, they do not support versioning and relational views are not updatable in general.

From a data-model perspective, structurally object-oriented systems have managed to overcome some of the deficiencies of RDBSs. They still suffer from not supporting encapsulation of data structures and application specific attribute types cannot be defined. Moreover, all the systems we looked at had a granularity problem. The graph-based systems have been built to manage a high-number of small objects and relationships efficiently. To achieve the required performance, concessions have been made with respect to multi-user support, transaction management and distribution. The entity relationship model oriented systems were built in order to manage coarse-grained objects and they do not achieve the performance required for management of project-wide abstract syntax graphs.

Our investigations revealed that ODBSs in general are the most promising systems to take. They provide a powerful data model that is very suitable for defining abstract syntax graph structures, as well as graph access and modification operations. O_2 and the Orion system can identify composite objects in order to offer version management primitives. The primitives have been proved to be sufficient for version management of subgraphs that represent documents. View mechanisms have been proposed for a number of object database systems. The mechanism implemented in O_2 is suitable for defining and updating virtual abstract syntax graphs based on a conceptual abstract syntax graph. Schema updates in O_2 can be done in a way that allows abstract syntax graph structures to be changed. Existing graphs migrate to the new schema incrementally. This is very important for the maintenance of tools in a commercial context, but we will not consider schema updates further and focus on tool construction rather than on their maintenance. A number of abstract syntax graph operations can be grouped as a transaction that has ACID properties. These transactions will later be used to implement syntax editor commands. They will ensure that commands can be executed concurrently and that the effect of completed commands is preserved against any failures. Unfortunately ODBSs do not support activities. Graph access and update operations can be performed from distributed workstations based on the multi-level client/server architecture that is commonly used in ODBSs. Finally, the Merlin Benchmark implementations on O_2 and GemStone provide evidence that ODBSs perform faster than relational or structurally object-oriented database systems. We, therefore, suggest using object database systems as the platform for development of syntax-directed tools. Of the class of ODBSs O_2 is the system most suitable. This is due to the extensions done to O_2 in the GOODSTEP project. They provide a higher standard of mechanisms, such as views, schema updates and versions, than those offered by other systems. We continue in the next chapter with the design of an architecture for syntax-directed tools that is based on object database systems.

Chapter 5

A Tool Architecture Based on Object Databases

In this chapter, we present design rationales for a generic, object-oriented architecture for tools. Tools constructed according to this architecture will meet both end-user and environment builder requirements as discussed in Chapter 2. The architecture is based on using an object database. Documents are represented as subgraphs of a project-wide abstract syntax graph according to the discussions in Section 3.1. These abstract syntax graphs are stored in an object database according to the considerations in Section 4.4.

For the underlying notation for the architectural considerations in this chapter, we slightly modify[1] the concepts and the notation that have been suggested for the Groupie system [ES94]. They trace back to [Lew88]. As we are developing an object-oriented architecture, we only use abstract data type modules. Abstract data type modules are, moreover, used in a restricted way so that they only export one *type* with a set of *operations*. We, therefore, consider these modules as *classes* in the following chapters. The *use relationship* between classes is displayed as a solid arrow. Dashed arrows represent a special kind of use relationship which we denote as *call-back relationship*. *Inheritance relationships* are represented by dotted arrows. A class inheriting from another class is a *subclass*. Operations may be *deferred* in the sense that they must be redefined in subclasses. A class that has a deferred operation is referred to as *deferred class*. Besides the exported operations, classes may also define *hidden operations*. In a graphical depiction, classes are represented as rectangles. In order to deal with the complexity of this architecture and thus perform the discussion at several levels of abstraction, we use the concept of *subsystems*. A subsystem is graphically represented by a rectangle with an underlying shadow. At a higher-level of abstraction, subsystems represent a set of classes or even subsystems. In a subsystem's refinement, some classes are declared to be exported by the subsystem. In the graphical notation, this is marked with a small black square in the upper left corner of the rectangle. The other classes are internal to the subsystem and cannot be used by subsystems or classes contained in other subsystems. Hence, the information hiding paradigm is applied not only to instance variables and operations contained in classes, but also at a more coarse-grained level to classes contained in subsystems. A use relationship between two subsystems means that classes in the source subsystem can import classes that are exported by the target subsystem.

[1]A more thorough investigation of these modifications is provided in [Bay95].

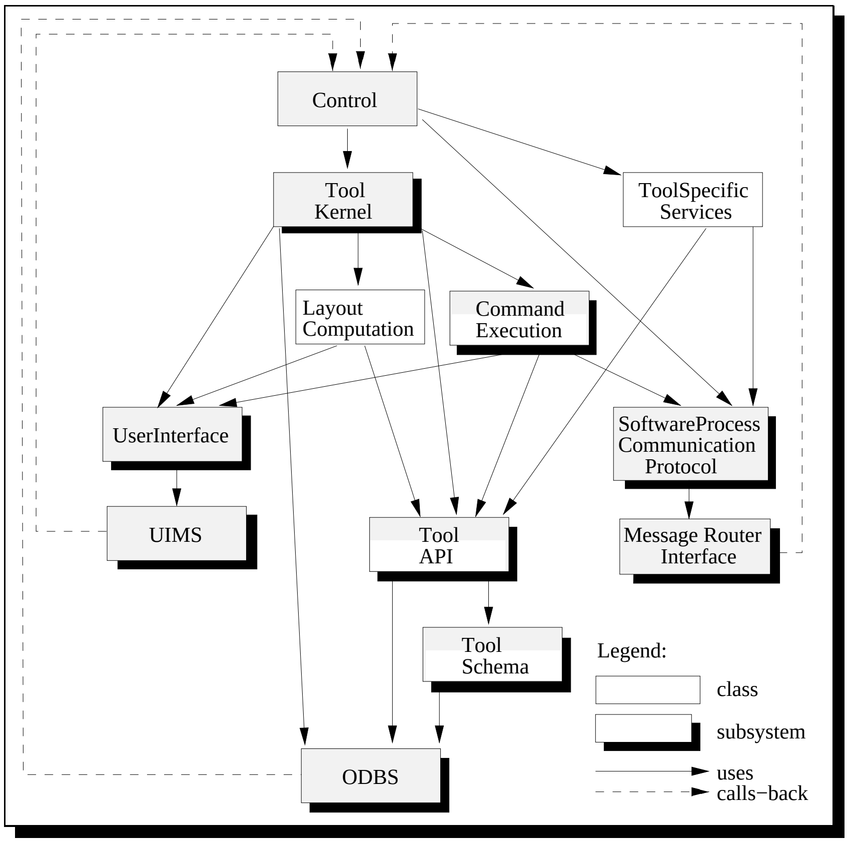


Figure 5.1: Architecture of Tools based on Object Databases

Figure 5.1 displays a first overview of the tool's architecture. We will refine it further in this chapter. The `ToolSchema` is a subsystem of classes defined in an object-oriented schema definition language. These classes implement the structure and the available operations for those subgraphs of the project-wide abstract syntax graphs that represent documents, which are persistently stored in an object database. All other components will be implemented in C++. The `ToolAPI` subsystem, therefore, provides the other components with a programming interface to the schema. The `LayoutComputation` class arranges for textual or graphical representation of the abstract syntax graph. The `CommandExecution` subsystem is capable of computing pop-up menus based on the current selection context and of executing a user's choice. The `UserInterface` subsystem exports a number of tool-specific window classes, the construction of which are based on a user interface management system (`UIMS`). The `ToolKernel` subsystem encapsulates the basic tool functionality. It can create new documents, open documents, delete documents, keep track of the currently opened editors, create pop-up menus and arrange for command execution. Moreover, it uses the transaction mechanism exported by `ODBS` in order to preserve integrity and execute commands concurrently. The `ToolSpecificServices` class exports operations that can be used to implement tool-specific services. It imports from a `SoftwareProcessCommunicationProtocol` that, in turn, uses an interface to a *message router*, such as Sun ToolTalk. Finally, the `Control` class controls the tool execution and translates various incoming events, such as user-input, requests of tool services or a deadlock in the database, into calls of operations exported by underlying components.

As discussed in Subsection 2.4, we are faced with the requirement of efficient tool generation. In order to meet this requirement, the tool architecture is constructed in such a way that a substantial number of components can be reused in arbitrary tools. During the course of this chapter, we will explain why each component can or cannot be reused. The distinction between *reusable* and *tool-specific classes* is shown in Figure 5.1. Rectangles representing reusable components are shaded in grey. If a subsystem does not only consist of reusable classes, it is only partly shaded in grey. In fact, the strategy to simplify tool construction based on reusing components is not new, but has been applied to earlier approaches to tool generation as well. Editors generated by the Centaur system, for instance, share a virtual tree processor, a user interface library and a rule interpreter [BCD+88].

5.1 Control

The architecture contains a number of external components, which we obtain from third party suppliers, which means that we cannot change them. These components are the `UIMS`, the `ODBS` and the `MessageRouterInterface`. These components need to interact with the internal components of the architecture. They are not only called by internal components, but also need to call internal components.

Situations in which external components have to notify internal components about particular incidents are called *events*. The common way to implement them is to pass a *call-back* operation, as an argument, to a registration operation supplied by the external component and thus associate the call-back with the event. If the external component detects the event, the registered operation is called back.

From an architectural point of view, the relationship between an external component and a call-back operation could be considered as a use-relationship, since the external component uses the call-back operation when an event is detected. The immediate consequence, however, is cyclic use-relationships, which should be avoided. These cause problems, for instance, during the determination of an order for compiling components when the components depend on each other.

We, therefore, mark those usages that lead to cyclic use-relationships explicitly as call-back relationships and only use them with external components, i.e. when they cannot be avoided. Thus, the use-relationship between all internal architecture components remains acyclic. In addition, we declare all the call-back operations within a single class so that call-backs are not distributed over the whole architecture. This class is the `Control` class.

For reasons that will become clear in Section 5.5, there are events that are caused by user-input and are detected by the `UIMS`. Some of these events cannot be handled by the `UIMS` but have to be communicated to higher-level architecture components. Therefore, we require call-back operations for reaction to user-input events in the `Control` class.

Most ODBSs follow a two-phase lock protocol. As already discussed in Section 3.6, this protocol may cause a transaction to wait for the completion of another concurrent transaction. Furthermore, it is not deadlock-free. These situations are detected as events by the ODBS system. If these events occur, an operation of the `Control` class will be called back. This operation can then invoke operations exported by the `ToolKernel` subsystem in order to notify the user about these events and ask whether to wait or to abort the current command execution.

If a service is requested from a tool, this request will be detected as an event by the message router interface. It then invokes a call-back in the `Control` class. This operation investigates the request to see whether the requested service is generic or tool-specific. If it is a generic service such as creating a document, deriving a version or opening a document in an editor, it will call operations exported by the `ToolKernel` subsystem. If the requested service is tool-specific, it will call an operation exported by `ToolSpecificServices`, which implements the particular service.

5.2 Tool Kernel

As discussed in Subsection 2.2.2, a user may want to display or edit multiple documents simultaneously. A tool must, therefore, be seen as a number of editors, each of which is used to display or edit one opened document. One of the purposes of the `ToolKernel` subsystem, is to meet this requirement by managing the set of editors that are displayed by the tool. Furthermore, the `ToolKernel` subsystem implements the command execution cycle as discussed in Figure 3.4 on Page 31. In doing so, it offers operations to the `Control` class that are used for reacting to user-input as well as to service requests. Finally, it offers an operation to the `Control` class for reacting to events detected by the database engine. All these tasks remain the same for arbitrary tools. Therefore, the whole `ToolKernel` subsystem is constructed in such a way that it can be reused among arbitrary tools and it is included in our reuse library.

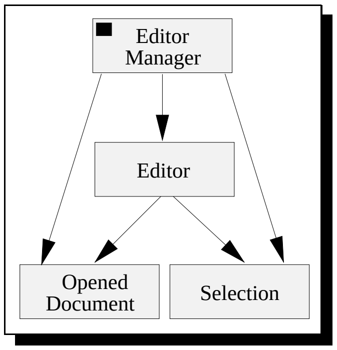


Figure 5.2: Architecture of the Tool Kernel Subsystem

The architecture of the `ToolKernel` subsystem is depicted in Figure 5.2. The export of this subsystem is the class `EditorManager`. During start-up, each tool creates an object of this class and stores it in an instance variable of the `Control` class. To invoke operations exported by the `ToolKernel`, the `Control` class then invokes a method from this object. The method itself uses other methods from `Editor`, `OpenedDocument` and `Selection`. We now discuss each of these classes in more detail.

Editor Manager

A user of a tool can open multiple editors in order to view or edit multiple documents at the same time. The purpose of class `EditorManager` is thus to implement the set of editors that are currently in use. Its main data structure is, therefore, a set where it inserts or deletes objects of class `Editor` whenever documents are opened or closed respectively.

Class `EditorManager` exports a set of operations which are used by the `Control` class to react to user input as well as to service requests. For most of the tasks listed below, an operation is exported, which obtains its parameters, such as document names, version names or file names, from dialogues with the user. Therefore, `EditorManager` imports from the user interface subsystem. Furthermore, most of these tasks represent generic services that should be offered by tools. During the execution of these services, there is no user who could be asked for parameters. Therefore, for each of the tasks, a second operation is exported, which obtains its parameters from arguments passed during operation invocation. Class `EditorManager` supports the following tasks:

- creating a new document and opening it in a new window in edit mode,
- importing a document from a textual representation stored in the file-system,
- opening an existing document version in edit or view mode,
- exporting a document version to the file-system,
- deleting a document,
- selecting an increment of a document version to become the current increment,
- computing a menu of commands applicable to the current increment,
- executing a command selected from the menu,
- aborting the current command execution to recover from a deadlock,
- freezing a version of a document,
- deriving a version of a document,
- merging two versions of a document and
- deleting a version of a document.

Among the tasks listed above, there are some, such as creating a new document or executing a command chosen from a menu, that modify the underlying abstract syntax-graph and must, therefore, be performed as ACID transactions. Class `EditorManager` thus imports operations from the `ODBS` to start and end a transaction. A transaction is started before the first object is modified and finished as soon as the last object has been modified. Besides ACID transactions, some ODBSs also support the concept of read-only transactions that are sequences of operations that do not modify any objects. In O_2, for instance, *programs* implement read-only transactions. These read-only transactions do not acquire any read locks on objects they access. We exploit this concept to improve the performance of tools and transaction throughput. Task that do not modify the underlying abstract syntax graph, such as selecting an increment or computing a menu, are thus executed as read-only transactions. Performance is improved, since time consuming locking is not performed during the execution of these tasks. Transaction throughput is increased, since locks are held for shorter periods of time and the probability of blocking concurrent transactions is decreased.

The method `ExecuteCommand`, which executes a command, imports classes and operations exported by the `CommandExecution` subsystem. After a command execution is completed, `ExecuteCommand` commits the current transaction and tool execution resumes as a read-only transaction. It must then redisplay any changes the command has caused in the project-wide abstract syntax graph. To improve the performance, changes must be redisplayed in an incremental manner. `ExecuteCommand`, therefore, implements the following strategy. If the command execution did not modify any increments, nothing needs to be done. The information as to whether or not any object was modified during a command execution is returned by the operation from the `CommandExecution` subsystem. If objects were modified, the current increment is redisplayed, for it is most likely that this has been modified. Moreover, each increment that is additionally changed is registered in a set called `ModifiedIncrements` associated with each document. `ExecuteCommand` then iterates over all documents that are currently displayed and then iterates over their `ModifiedIncrements` sets and redisplays each of the increments in these sets. To redisplay an increment, `ExecuteCommand` imports a method called `UnparseToUserInterface` that is exported by class `OpenedDocument`. Finally, each set is emptied.

Editor

The main purpose of class `Editor` is to implement the editors where documents are displayed. Class `Editor`, therefore, maintains references to a document, which is an instance of class `OpenedDocument`, an edit window (i.e. an instance of class `EditWindow` that is imported from the `UserInterface` subsystem), an instance of class `Selection` to keep track of the currently selected increment and an instance of class `EditorErrors`, which displays errors of the current increment in more detail[2]. Instances of class `Editor` are created in class `EditorManager` whenever a new document is opened. They are deleted whenever a document is closed.

The operations exported by `Editor` are used by methods from class `EditorManager`. A common feature is that they access or modify the state of an editor. The operations in particular are:

- selection of an increment,
- obtaining the current increment, if any,
- opening and closing error windows, and
- raising an editor that may be hidden behind other windows.

The `Editor` class imports methods to create and delete edit windows and error windows from the user interface subsystem. It also imports methods to create and delete edit documents and selection objects.

Opened Document

The purpose of the class `OpenedDocument` is to maintain a reference to the database object implementing the root node of the document subgraph. In addition, it has a reference to an object of class `LayoutComputation` that encapsulates the unparsing rules for the document. An argument of the object constructor is the name of a document which is used in a database query to obtain the root node of the subgraph. Also during creation, the instance of class `LayoutComputation` is created.

The operations exported by `OpenedDocument` access and modify the state of an edit document. In particular, they enable the `Editor` and the `EditorManager` classes to:

- obtain the database object implementing the root node of the document,
- incrementally recompute the textual or graphical representation of an increment. Therefore, the old representation is deleted first and then a new representation is inserted.

[2]Remember that errors in documents are only visualised by underlining and a more detailed error description is displayed in a separate window on demand.

Class `OpenedDocument` uses operations exported by the `ToolAPI` subsystem to obtain the reference to the root node of the document subgraph. It imports a method to delete an increment representation from class `EditWindow`. Finally, it imports the `unparseIncrement` from class `LayoutComputation` in order to insert a new increment representation.

Selection

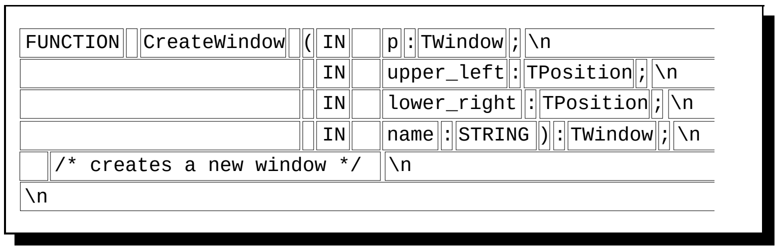
The purpose of class `Selection` is to store a reference to the current increment. The reference is needed in both the `Editor` class and the `CommandExecution` subsystem. The `Editor` class needs this information to compute the contents of the error window. The `CommandExecution` subsystem needs it for computing context-sensitive menus according to the current selection. In order not to store this information redundantly, we included class `Selection` for storing a reference to this database object and for sharing objects of this class among objects of the `CommandExecution` subsystem and the editor objects.

5.3 Layout Computation

The purpose of the `LayoutComputation` class is to implement the unparsing schema and to arrange for associations between increments and portions of text or graphics. As different tools may have different unparsing schemas, this class is tool-dependent and cannot become part of the reuse library for tool construction.

Due to the need to re-compute the layout incrementally, different increment types have to be used as starting points. The different increment types depend on the supported language and, therefore, differ from tool to tool. On the other hand, the `ToolKernel` subsystem should not be aware of the different kinds of increments that can occur in a document in order to reuse the subsystem between different tools. The object-oriented paradigm is used to solve this dilemma as follows. `UnparseIncrement` has an argument which determines the increment to be redisplayed. The type of this argument is the class `Increment` exported by the `ToolAPI`. As we will see later, any tool-specific increment class defined in the tool's schema inherits from this class. Exploiting polymorphism, arbitrary increments may, therefore, be passed as arguments to the `unparseIncrement` operation. Therefore, the signature of the `unparseIncrement` operation remains stable between different tools. Furthermore, for each type of increment that can occur in the respective language, the class includes a hidden operation that implements the unparsing scheme for this type of increment. These hidden layout computation operations are invoked by `unparseIncrement`. Which particular operation is invoked by `unparseIncrement` is determined during run-time by the type of increment passed as argument.

The hidden operations that are used to compute the textual or graphical representation of an increment insert *segments* into the edit window. A segment is an atomic portion of text or a bitmap. To insert text or graphics into the window, the `LayoutComputation` class needs a reference to the `EditWindow` which is passed as an argument to the object constructor. The operations, therefore, use an operation `InsertIncrementPart` provided by class `EditWindow` and pass the text as an argument. To insert the textual representation of a child increment, they call the respective layout computation operation available for this type of child increment. Figure 5.3 indicates the segments for the function displayed in an edit window of the Groupie module interface editor on Page 14.



```
FUNCTION CreateWindow ( IN p : TWindow ; \n
                        IN upper_left : TPosition ; \n
                        IN lower_right : TPosition ; \n
                        IN name : STRING ) : TWindow ; \n
   /* creates a new window */ \n
\n
```

Figure 5.3: Segments of Text in an Edit Window

The `LayoutComputation` class for this tool contains an operation `unparseFunction` that first inserts a segment containing the keyword `FUNCTION` and then a blank segment. After that, an operation `unparseOpName` is called. As argument to that operation, the increment identifier of the operation name is passed, which, in turn, is obtained from a traversal function exported by the `ToolAPI` subsystem. `UnparseOpName` then inserts the value of the function identifier as a new segment into the window and returns. `UnparseFunction` then continues inserting a blank segment, calls `unparseParameterList` and so on.

In order to allow the user interface subsystem to associate segments with increments, the operation `InsertIncrementPart` has an additional parameter where the layout computation operations pass increment identifiers, which they obtain from the `ToolAPI` subsystem during traversal. Hence, the first two segments in Figure 5.3 are associated with the function increment, the third segment is associated with the operation name increment, the fifth segment is associated with the parameter list and so on. If the user then clicks on a segment, e.g. the first segment containing `FUNCTION`, the `UIMS` passes the function increment identifier to the call-back operation in the `Control` class which uses this information to call the select operation exported by the `EditorManager` class.

As required in Subsection 2.2.1, editors have to visualise inter-document consistency constraint violations and static semantic errors. This visualisation can best be implemented in the `LayoutComputation` class. Before inserting a segment into the edit window, the layout computation operations check whether the corresponding increment is error free. They do so using the operation `has_error`, which is defined in the underlying database schema in the most general class `Increment` and is thus available for arbitrary increments. The operation `InsertIncrementPart` has another parameter which determines whether the new segment is to be marked or not. To implement the visualisation of errors, the layout computation operations only pass the result of the call to `has_error` as an argument to `InsertIncrementPart` and the implementation of `InsertIncrementPart` in class `EditWindow` marks the segment or not.

The class `LayoutComputation` has to access a number of operations from the schema. It, therefore, imports operations from the `ToolAPI` subsystem. The operations include traversal operations for navigating through the abstract syntax graph stored in the ODBS. Furthermore, `has_error` is imported from a class exported by `ToolAPI`. In addition, the `LayoutComputation` class intensively uses the `InsertIncrementPart` operation, exported by the `EditWindow` class contained in the user interface subsystem.

5.4 Command Execution

The purpose of the `CommandExecution` subsystem is to export two operations. The first one is used for computing a list of strings that represent command names for a given selection. The second operation is used for executing a command. These two operations are imported by the `EditorManager` class in order to implement operations that are used by call-back operations in the `Control` class. The `Control` class, in turn, reacts to events detected by the user interface subsystem, such as pushing the right mouse-button or selecting an item from a pop-up menu.

Three properties characterise a *command*. It has a name, a precondition and a list of operations that are executed when the user has selected the command. This list may include operations implementing user dialogues as well as operations on the project-wide abstract syntax graph, which are implemented in the tool's schema. The list of operations has ACID properties as required in Section 2.3.2. A *menu*, in turn, consists of a collection of commands that satisfy the precondition for the current selection.

Each command is implemented in a separate class. We call these classes *interactions*. Upon construction of objects of such a class, an object of class `Selection` representing the current increment is passed as an argument and stored in an instance variable. The class has three methods, which implement the properties of commands: `GetName` returns the name of the command as a string, `IsAvailable` returns a boolean value that determines whether or not the command is available for the current selection and finally, `Execute` executes the command. It returns true, if the command was successfully completed and false, if it was aborted. This return value is evaluated by the `EditorManager` in order to call a database transaction commit or abort operation.

Menus are implemented as collections of interactions in class `InteractionCollection`. The class, therefore, stores a list of interactions in an instance variable. It exports an operation to obtain a list of strings that represents the names of commands to be presented in a menu. It also exports an operation to execute a particular operation. It is implemented by invoking the `Execute` operation of the respective interaction. The constructor of this class obtains an object of class `Selection`, which identifies the current increment. It then constructs the context sensitive menu by creating instances of interaction classes, testing for their availability and including them in the collection if `IsAvailable` returns `true`. It is a problem to decide which commands to consider when an interaction collection is built. If any command the tool supports is considered, menu computation is certain to be too inefficient.

Commands in structure-oriented editors are always applied to the current increment, or its parents in the abstract syntax. This observation can be exploited for an efficient computation of menus. Therefore, interactions are grouped by increment types. Furthermore, an interaction collection class is defined for each increment type. During construction of an interaction collection, the collection only considers interactions defined for one increment type. To compute the available interactions, we only have to consider the interaction collection that is defined for the type of the current increment, and the collection that is defined for the parent type of the current increment. These collections have to be merged.

Now we again have the problem that increment collection classes are language- and thus tool-specific, though their use in the `ToolKernel` subsystem requires a uniform interface. We solve the problem by following the same strategy as that used during layout computation. We create a new class `MenuConstruction` and declare this class to be an export of the

CommandExecution subsystem. The only parameter of the constructor of this class is an object of class Selection representing the current increment. Upon construction it decides which interaction collections to construct based on the actual type of the current increment. It then exports an operation which returns the computed increment interaction collection. The increment type specific interaction collection classes are, in turn, implemented as subclasses of class InteractionCollection. Exploiting polymorphism, MenuConstruction may thus return a language specific interaction collection to the EditorManager class. As this inherits its signature from the predefined class InteractionCollection, the ToolKernel subsystem remains language independent and thus can still be reused in different tools.

Inheritance can also be exploited to share commands between different, but similar, increment types. As an illustrating example, consider procedures and functions as used in the interface definition language of Figure 2.4 on Page 14. Commands for creating and changing names, parameter lists and comments had to be included in both menus. Therefore, a lot of interactions had to be defined redundantly for both functions and procedures. This can be avoided completely if an interaction collection for an abstract increment type Operation is defined. Interactions implementing commands that are applicable to functions and procedures are associated with increment type Operation. During the construction of this interaction collection for Operation, the interactions for type Operation have to be considered. The collections for functions and procedures then have to inherit from the Operation interaction collection. During the construction of these child interaction collections, we need only ensure that the constructor of the parent interaction collection is executed.

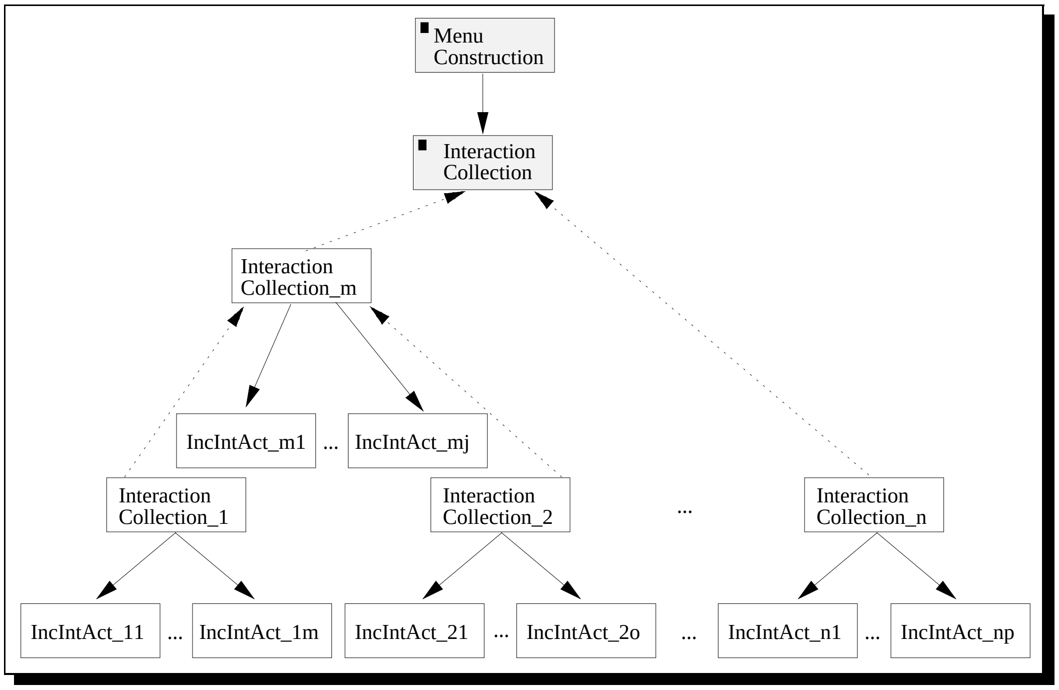


Figure 5.4: Design of Command Execution Subsystem

This inheritance mechanism can also be exploited for reusing command definitions that are common to arbitrary tools. Every tool, for instance should have commands for version and configuration management of documents. An abstract increment class DocumentVersion that has the property of being the unit of version management can, therefore, be introduced. All predefined interactions implementing commands for version management can then be attached to this increment class and the interaction collection for class DocumentVersion arranges for inclusion of these interactions in a menu. If a tool is to support version management, the interaction collection for the root increment type of this tool can then be declared to inherit from the collection for the predefined abstract class `DocumentVersion`. Similarly, interactions implementing commands for building configurations are attached to an abstract increment class `UsingIncrement` from which language-specific increment classes, representing imports from other documents, inherit.

For a summary of the discussion consider the architecture of the `CommandExecution` subsystem displayed in Figure 5.4. The subsystem exports classes `InteractionCollection` and `MenuConstruction`. Their export interface remains stable in different tools. During instantiation of an object of class `MenuConstruction` for a current increment, two interaction collections are constructed and merged afterwards. During construction of an interaction collection, all interactions that are defined for an increment type or its super types are constructed and included in the collection, only if they are available. This is determined by operation `IsAvailable` that is defined in each interaction class. After that an interaction collection can be obtained by the `EditorManager` class from the `MenuConstruction` class. The collection will contain only those interactions that are applicable in the current selection context. Interactions can be inherited by defining inheritance relationships between the respective interaction collection classes. This allows for the sharing of commands between different increment types as well as for the reuse of predefined commands between different tools.

5.5 User Interface

The purpose of the `UserInterface` subsystem is to export a small number of classes that implement all user dialogues occurring in syntax-directed tools. The subsystem is, therefore, reusable among different syntax-directed tools. The subsystem imports from a `UIMS` that exports a high number of high-level primitives, called *widgets*, for construction of application specific window types. The user interface subsystem then aggregates these widgets to application specific *window types* and adds classes implementing dialogues. It, therefore, hides the complex use of the `UIMS` from the other architecture components. The `UIMS`, in turn, is built on top of a basic window system such as X-Windows, OpenLook, SunView or GKS. The same `UIMS` is most often available for different basic window systems. It thus arranges for portability of the whole tool on top of various user interface platforms.

The dialogues implemented by the user interface subsystem are either *non-exclusive* or *exclusive*. In a non-exclusive dialogue, a user can start interacting with a tool and change to another dialogue without having to complete the first dialogue. As an example consider editing a document in one edit window. The tool must support the editing of another document in another window without having to close the first window. In some situations, however, it is not important to have these non-exclusive dialogues or even required to force completion of a dialogue before another dialogue can be started. This is most often the case when the dialogues are short. Examples of this are dialogues that display a message to the user or have the user selecting a choice from a selector window. The property, whether or not a dialogue is exclusive is implemented by the window that is used during the dialogue. We, therefore, distinguish *non-exclusive* from *exclusive windows* in the following.

From an architectural point of view, exclusive dialogues can be implemented by a single exported operation. These operations return the result of a dialogue as soon as the dialogue is completed. Implementing a non-exclusive dialogue by a single exported operation, however, is impossible. If we tried to do so, that operation would return when the dialogue was completed. Then, however, any other dialogue would be blocked. To implement these non-exclusive dialogues, we have to give control to the user-interface management system instead, as soon as the tool is idle. The user-interface management system must then inform the tool as soon as a user-input event happens. In order to do so, we require call-back operations in the `Control` class. Hence implementation of the non-exclusive windows requires definition of call-back operations.

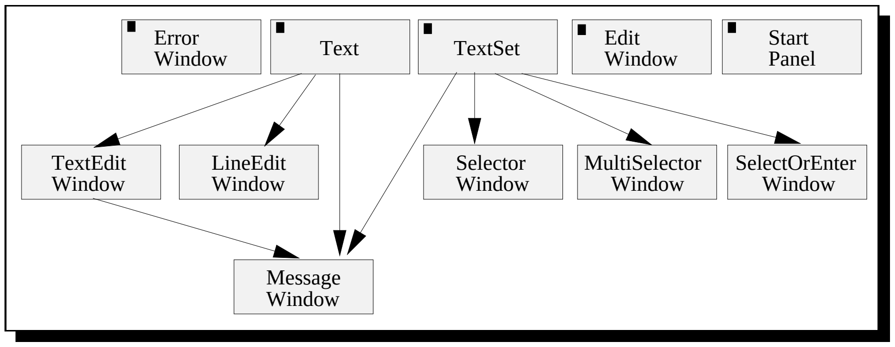


Figure 5.5: Architecture of the User Interface

The overall architecture of the user interface subsystem is displayed in Figure 5.5. Classes `Text` and `TextSet` export a number of operations that implement exclusive dialogues. These dialogues are implemented using elementary window types such as line-edit or message windows that are not exported. Classes `EditWindow`, `ErrorWindow` and `StartPanel`, in turn, implement non-exclusive windows. They export a number of operations such as inserting a new segment into an edit window or highlighting all the segments that belong to a given increment. These operations can then be used by higher-level layers of the architecture in order to implement output operations during non-exclusive dialogues.

In order to implement the user interface subsystem, a number of features are required from the underlying `UIMS`. First of all the `UIMS` must offer all the widgets required for construction of the window types depicted in Figure 5.5. These include text widgets with sliders, which implement scroll bars, menus, buttons, dialogue boxes, icons, various kinds of polygons and the like. The construction of window types will become significantly easier if the `UIMS` offers an interactive graphic editor for aggregating widgets to window types. Finally, an important requirement arises during the implementation of increment selection. When the user clicks on a text segment contained in an edit window, the `UIMS` must be able to communicate the identifier of the increment that is associated with the segment to the `Control` class. Only then is the `Control` class enabled to invoke the respective operations from the `EditorManager` class efficiently. Hence, we require that the invocation of call-back operations after event detection in the `UIMS` can pass application specific arguments to a call-back operation.

5.6 Tool Specific Services

In Section 2.2.2, we classified services into generic and tool-specific services. As we have seen, all generic services are implemented by the operations of the `EditorManager` class. The purpose of the `ToolSpecificService` class is then to implement services, such as the creation of a particular relationship, that are specific to a particular tool.

If a service request arrives at the communication subsystem, an operation exported by the `Control` class is called back. This call-back operation invokes an operation from the `SoftwareProcessCommunicationProtocol` subsystem to obtain a message object that represents the particular service request. This message object obtains service parameters from the communication subsystem and temporarily stores them. If the message represents a tool-specific service, a new service object will be constructed by the `Control` class as an instance of the `ToolSpecificService` class. The respective message object is passed to the constructor as an argument. The constructor then investigates the type of the message and invokes one of the hidden operations in class `ToolSpecificService`.

Each of these operations implement a tool-specific service. The constructor passes the service parameters stored in the message object to the operations as arguments. If the service is synchronous, the operation returns a result. Upon termination of the service this result is passed to the message object which, in turn, arranges to transfer the result to the process that requested the service and is waiting for its completion.

To implement services, the hidden operations must be able to access and modify documents. Hence, this class has to access and modify the underlying abstract syntax graph. It, therefore, imports classes, which implement nodes and edges, from the tool API subsystems. Moreover, the constructor of the class imports from the `SoftwareProcessCommunicationProtocol` subsystem. It uses operations to obtain a service's arguments and to provide the service requester with the result of the service execution.

5.7 Software Process Communication Protocol

The purpose of the `SoftwareProcessCommunicationProtocol` subsystem is to provide tools with a communication protocol for sending events to the process engine and receiving service requests from there. The subsystem will be built on top of a session-oriented message router such as Sun ToolTalk [Sun93], HP SoftBench [Cag90], DEC FUSE. In these systems, several processes can join a session and afterwards communicate with each other by exchanging messages. We aim to use one of these systems, rather than operating system primitives such as sockets or pipes, for inter-process communication because they enable communication to be defined at a much higher level of abstraction. Using operating system primitives we would gain a slight performance benefit. The results of benchmarks performed with SoftBench and ToolTalk [Ger94], however, suggest that these systems require between 50-100 milliseconds for a complete communication cycle, which we consider to be fast enough.

The rationale for providing a subsystem on top of one of these basic message routers is the same as for the `UserInterface`. The message routing is hidden from the rest of the tool architecture in order to allow for portability and to provide a dedicated, safe and application-specific protocol for communication between tools and process engine. Figure 5.6 displays an overview.

The `CommunicationChannel` class provides the channel for all communications between tools and the process engine. Each tool has a reference to an object of class `CommunicationChannel`, which is stored in an instance variable of the `Control` class. Upon creation of this object the basic message router interface is initialised and the tool joins a particular session with the process engine. If the message router interface has informed the `Control` class about a new service request, a `read` operation will be invoked reading the message from the communication

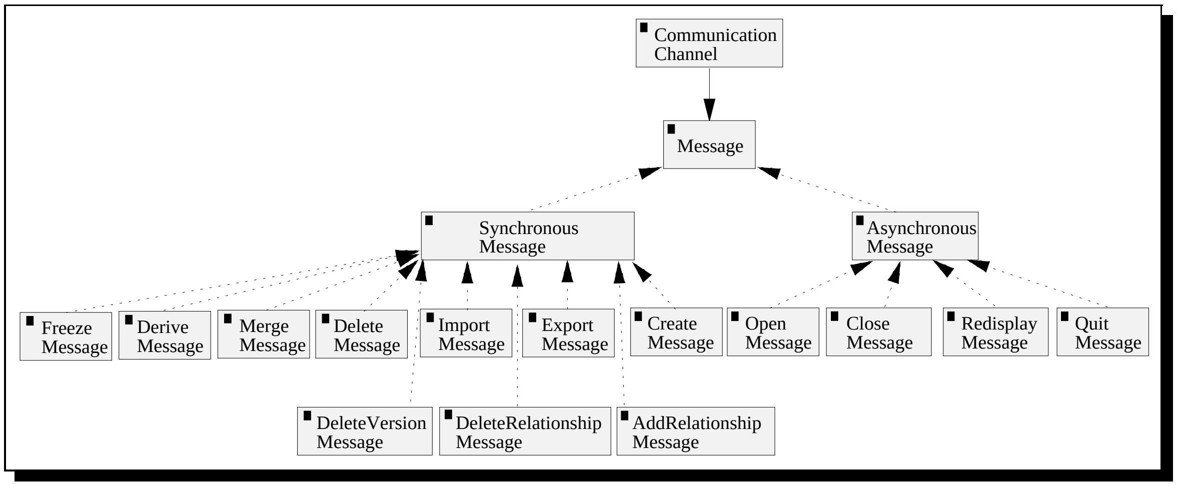


Figure 5.6: Architecture of the Software Process Communication Protocol Subsystem

channel object. The `read` operation investigates what kind of request it is and constructs an instance of one of the subclasses of class `Message`. Similarly, a `write` operation is provided in order to enable the tool to inform the process engine about process events.

Service requests, as well as events, are represented as messages. Hence the `read` operation of the communication channel returns an object of class `Message` and the parameter of the `write` operation is an object of `Message` as well. During the construction of a message that represents a service request, the parameters of the service request are obtained from the communication subsystem and stored in instance variables of the respective message objects. Access operations for each of these service parameters are provided. During construction of a message that represents an event, the event parameters are passed as arguments to the constructor. As most of the services and events have different parameter types, different message classes are defined. They inherit from class `Message`, which is a deferred class. Polymorphism is used for the argument and result type of the `write` or `read` operation of the communication channel in order to pass the appropriate kind of message.

In order to implement asynchronous services and asynchronous events, it is sufficient to read or write a message from or into the channel. The tool can continue operating as soon as the communication channel has completed execution of the `read` or `write` operation respectively. To implement a synchronous service or event, however, additional measures have to be taken in order to inform the process engine of the completion of the service or to obtain the response to an event from the process engine. To write the result of a synchronous service, each subclass of class `SynchronousMessage` exports an operation `write_result`. The parameter of this operation is message specific. Besides transferring the result, it also implements the notification that the service has been handled. A design problem is then how to obtain the result of an event notification from the process engine. A first option would be to send the respective message using a synchronous protocol, as offered by the various basic message routers, and return the result of the event as a result of the write message. This option, however, is inappropriate since the tool would then be waiting under the control of the message router for the handling of the message. As a consequence, actions, such as resizing a window of the tool or moving a window to another position while the tool is waiting, are not handled appropriately in the tool. This is because the required updates to the window contents are not performed: the window manager, which handles these actions, is independent of the tool and runs in a separate operating system process. It requests updates on window contents from the UIMS of the tool in the case of a resize or move operation. If the tool is waiting for a response to a message (which may take a while, depending on the load of the process engine), the `UIMS` cannot handle the required updates accordingly. The second and better option is, therefore, to wait under the control of the `UIMS`. Therefore, the message has to be sent using an asynchronous protocol. In the `write` operation of the communication channel, we then invoke an operation from the `UIMS`, which returns after a fixed amount of time (say a tenth of a second). After the UIMS operation has returned, we check whether the result has arrived. If not, the UIMS operation is invoked again. Otherwise, we return from the `write` operation. The result of an event may then be obtained or determined by the operation `read_result`, which is exported by all subclasses of class `SynchronousMessage`.

The software process communication protocol as suggested above can be reused in arbitrary tools. It implements the required communication between process engine and tools for all generic services and events. It even implements some tool-specific services and events. Therefore, message types are parametrised. Rather than having different message types for creating different kinds of relationships, for instance, the message `AddRelationship` has a parameter which denotes the relationship type. Nevertheless, a need may arise to add tool-specific messages to the protocol in order to implement tool-specific services or tool-specific events.

We have to require several facilities from the underlying message router in order to be able to implement the communication protocol as suggested above. The overall requirement is that the communication protocol must allow us to implement message objects transfer, i.e. instances of the various message classes, between the process engine and tools and vice versa. Therefore, these message objects have to be transformed into low-level messages, i.e. as sequences of atomic message components, which the message router can transfer. With respect to these low-level messages, we have the following requirements:

Message types: As we have seen above, synchronous services and events cannot be implemented using synchronous communication primitives. To implement them we require instead a low-level message type from the communication mechanism that we call *request*. These requests are a bi-directional means of communication between tool and process engine. The first direction is used to send a message representing a request. The message router then returns control to the tool or process engine, respectively. As soon as the request is handled, the requested process adds an acknowledgement to the request message and sends it back to the requesting process. The message is then destroyed by the requesting process. To implement asynchronous services and events, we require a low-level message type *notification*, which implements one-way communication. The sender of this message does not care about whether the service/event has actually been handled. The message is destroyed by the notified process.

Parameter types: The router should allow for composition of atomic message components, which we call *message parameters*, to complex messages. As types for these parameters, `boolean`, `integer`, `char` and `string` are required at the very least.

Message routing: It may be too complex for the process engine to keep track of which tools have been started and to explicitly route messages to particular tools. Therefore, the messages defined above do not include any routing parameters. For messages that are sent from tools to the process engine, routing is obvious. For these messages, *peer-to-peer* routing may be used. For messages from process engines to tools, *broadcast* routing is required.

Fault tolerance: The mechanism must tolerate and correct errors. As an example consider that an addressee of a message is not reachable. In this case the message router may correct the error by starting the tool or process engine. If the mechanism cannot correct errors, tools and process engines must be notified about failures in message delivery.

Efficiency: Message delivery by the router should not result in any significant decrease in the overall tool performance. We, therefore, require a round trip for a request type message to perform faster than 50 milliseconds.

Distribution: We required tools to run in a distributed environment. Therefore, tools must be able to join sessions from remote workstations. The message router should be fully supportive in this respect and be able to transfer messages between different workstations using a local area network.

In short, we have defined an architecture that implements a communication protocol between the process engine and tools. We have discussed how this architecture can be used to implement both services and requests. The communication protocol implementation is reusable among arbitrary tools. To implement tool-specific services, additional message classes may have to be added. Of the available message routers, ToolTalk fulfils these requirements, but the available versions of the HP BMS and DEC FUSE do not meet the distribution requirement. We, therefore, select Sun ToolTalk for our implementation as a message router.

5.8 Tools Application Programming Interface

The components of the tool architecture that we have discussed so far have to be implemented in C++. The reason is that the third party components, namely the message router interface and the UIMS have programming interfaces for C++, but not for other object-oriented languages such as SmallTalk, Eiffel or Beta. The object database schema defining the structure and available operations for syntax graphs will not necessarily be defined in C++. In GemStone, for instance, a dedicated schema definition language called *OPAL* must be used, Orion schemas have to be defined in CommonLisp, and O_2 schemas are defined in O_2C. Thus a need arises for interfacing the higher-level architecture components with the database schema definition. This is necessary, for instance, in the layout computation class in order to navigate through an abstract syntax graph or for the command execution subsystem to invoke methods in order to change the syntax graph. The purpose of the *tool application programming interface* (`ToolAPI`) is to arrange for this inter-operability. In object database systems where the schema is defined in C++, for instance, Ontos or ObjectStore the `ToolAPI` can be omitted.

All databases that have their own schema definition language support a C++ interface to access the database from applications written in C++. To achieve these accesses, the database system can be requested to generate a twin C++ class for a class of the database schema. The twin C++ class can be generated in such a way that it exports the same methods as the methods of the schema class, or a subset thereof. Objects of the twin C++ class are called *persistent pointers* in the following paragraphs because they point to persistent database objects. Their classes are called *persistent pointer classes*, accordingly. Invoking a method on a persistent pointer then effectively invokes the method for the database object.

As an example, consider an O_2 class `Function` that is written in O_2C. It implements nodes of type `Function` of the abstract syntax graph that was depicted in Figure 3.1 on Page 21. To access and modify objects of this class, a number of methods are defined whose purpose is of no concern at the moment.

```
class Function inherit Increment
type tuple(...)
method
  ...
  public expand_name(Str:string):boolean,
end;
```

The following O_2 export command creates a persistent pointer class `Function` in C++. This class declares the given methods as its export. The class hierarchy of the persistent pointer classes is identical with the hierarchy in the database schema.

```
export in "."
class Function
methods expand_name
to C++;
```

The persistent pointer class `Function` may then be used as follows to instantiate the template class `o2` for instance in a method of a C++ class contained in the `CommandExecution` subsystem for function increments.

```
Boolean FunctionInteraction1::Execute() {
  o2<Function> f;
  Text t;
  TextSet * err;

  f=(o2<Function>)SelectionContext->GetTheSelection()->father();
  t=Text("");
  if(t.LINE_EDIT("Enter Function Name")) {
    if (f->expand_name(t.CONTENTS())!=TRUE) {
       err=new TextSet(f->get_set_of_errors());
       err->DISPLAY();
       delete err;
       return(FALSE);
    }  else
       return(TRUE);
  }
}
```

The second line declares `f` as a persistent pointer to an O_2 object of class `Function`. The next two lines declare user interface objects that are used to implement particular user dialogues during the command execution. Assuming that the currently selected increment is a function name place holder, the first statement obtains the increment and navigates to the enclosing increment, which is a function, and assigns a reference to the O_2 object implementing this function node to `f`. Then the user is requested to enter a new name for the function. If the dialogue is completed by the user, the method `expand_name` is invoked with the persistent pointer object. This, in turn, causes the O_2 C++ interface to invoke the O_2C method `expand_name`, which performs the modifications to the syntax graph as defined in the schema. It returns the O_2C value `true` if the modification has been successfully completed. Then the O_2C value `true` is translated by the interface into the C++ counterpart and the command execution is successfully completed. This is signalled to the `ToolKernel` subsystem, which, in turn, commits the transaction, by returning `TRUE`. Otherwise the C++ method `get_set_of_errors` is invoked. This method is defined by the C++ persistent pointer class for a predefined class `Increment` and inherited by the pointer class `Function`. This method returns a textual representation of the errors that have been obtained at an increment. The result is translated into a C++ string by the C++ interface and displayed with a newly created dialogue element. Then the dialogue element is deleted and failure of the syntax graph update is signalled to the `ToolKernel` by returning `FALSE`. The `ToolKernel` in that case aborts the current O_2 transaction in order to undo any changes.

5.9 Tool Schema

As argued in Section 3.1, structure and available operations of a tool's abstract syntax graph should be defined and controlled within a database schema. In Section 4.4, we sketched at a quite high level of abstraction, how this can be done with object database systems. In this section, we discuss in detail how a schema for storing and manipulating abstract syntax graphs is defined in an object-oriented schema definition language. We keep the discussion general so that it can be applied to several object database systems. To illustrate it, we use examples that are defined in O_2C, the schema definition language of O_2, which we identified as the most appropriate ODBS for tool construction.

Along the lines of our previous architectural considerations, we focus on reuse of tool schema components. We, therefore, first identify properties of tool schemas that are common to arbitrary tools. These properties are implemented in classes that can be reused within different tool schemas. As these classes are implemented up-front, we call this part of a tool's schema *predefined schema* and the classes *predefined classes*. The other part contains tool-specific classes, which use properties from predefined classes. The use can be by inheritance or by defining instance variables whose types are predefined classes. This reuse of predefined schema components again simplifies the tool construction process.

5.9.1 Predefined Classes

Properties that are common to any tool schema and that should, therefore, be implemented in the predefined schema are:

- nodes of the abstract syntax graph are persistent,
- nodes either represent a place holder or an expanded increment,
- nodes may be optional,
- nodes may be leaf nodes or inner nodes of the abstract syntax tree,
- nodes may have outgoing multi-valued aggregation edges,
- nodes may represent erroneous increments,
- nodes may represent a scoping block,
- nodes belong to a subgraph that represents a document,
- nodes may have reference edges to nodes contained in other documents and these edges are used for configuration management,
- edges must be traversable in both directions[3],
- documents have an external unique name,

[3]N.B. Edges, if implemented by instance variables, are only traversable in one direction.

- documents are the unit of ownership,
- documents are the unit of version management and
- documents have a time stamp of their last modification.

These properties are implemented in classes contained in the predefined schema as depicted in Figure 5.7. It includes two disjoint class hierarchies. The subclasses of `Increment` are called *increment classes*. They are used to define common properties of node types. The subclasses of `Attribute` are referred to as *non-syntactic classes*. They define properties of types for node attributes.

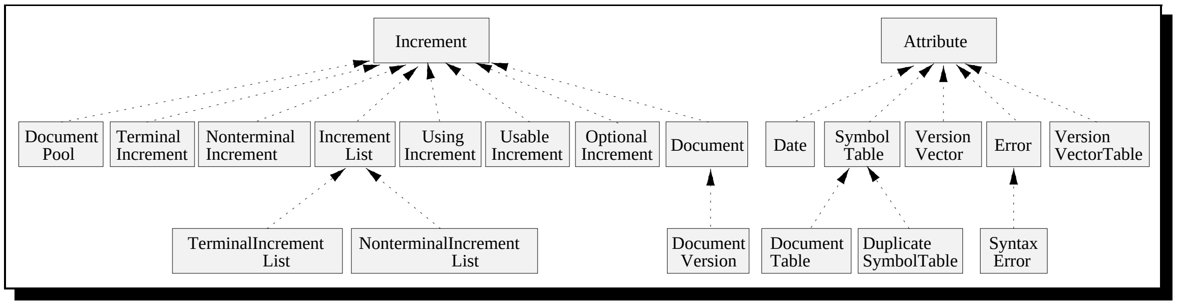


Figure 5.7: Classes Contained in the Predefined Schema

Class `Increment` is the generalisation of arbitrary node types. Therefore, each class implementing a node type should be a subclass of `Increment`. It has an instance variable `father` that implements an edge to the father node with respect to the abstract syntax. For each node, it implements the reverse edge to the syntactic edge that leads to the node. This instance variable is initialised during object creation. It is updated whenever the node is moved to some other position in the abstract syntax graph. Moreover, class `Increment` maintains a set of error descriptors that represent the semantic errors the increment is involved in. The increment is considered to be erroneous if this set is not empty. The class exports an operation that can be used for checking whether the node represents an erroneous increment. Moreover, it exports an operation that returns a set of strings as textual representations of the error set. A further instance variable of class `Increment` is used to indicate whether the increment is already expanded or still a place holder. Objects of class `Document` implement root nodes of syntax trees that span up subgraphs of the project-wide abstract syntax graph representing documents. Class `Document` has an instance variable of type `string` in order to store the owner of the document and an instance variable of class `Date` to store the time stamp of the last document modification. Class `OptionalIncrement` specialises `Increment` in that objects of this class represent nodes that are optional. It defines methods in order to delete or expand an optional increment.

The classes `TerminalIncrement` and `NonterminalIncrement` define the properties of leaf nodes in the abstract syntax tree and of inner nodes, respectively. Subclasses of `TerminalIncrement` are called *terminal increment classes* and subclasses of `NonterminalIncrement` are called *nonterminal increment classes*. The common properties of nodes that have an outgoing multi-valued aggregation edge are defined by class `IncrementList`. The class `TerminalIncrementList` and the class `NonterminalIncrementList` are more specialised in that the target of the multi-valued edge are inner nodes and leaf nodes, respectively. These list classes define a multi-valued instance variable to implement the multi-valued aggregation edge and then offer methods, for instance to add, insert and delete elements of the list.

Class `DocumentVersion` represents an abstraction for those document types that are versionable. In our implementation it uses the O_2 Kernel class `Version` (c.f. Page 80) for version management purposes. During creation of an object of class `Increment`, which is implementing a node, the object must be entered into the version unit object of the respective document and, during destruction of an object, the object has to be removed from the respective version unit object. `DocumentVersion` then offers a number of methods for version management such as freezing a document, deriving a new version, merging two versions, establishing a default version, navigating through the version history graph of the document and so on. `UsableIncrement` and `UsingIncrement` implement nodes with incoming and outgoing inter-document reference edges. These classes offer methods for implementation of configuration management, such as selecting a particular version of a used document. For configuration management purposes class `DocumentVersion` internally uses instance variables of classes `VersionVector` and `VersionVectorTable`.

An object of class `DocumentPool` is declared as a persistent root in each schema in order to implement persistence of documents. This object is persistent by definition, as an object reachable from a persistent root is persistent. `DocumentPool` then has an instance variable of class `DocumentTable`. Root nodes of documents may then be inserted in this table. As a consequence, each root node of a subgraph that represents a document is persistent, too. As each node of the subgraph is reachable from the root node of the respective subgraph, all nodes are persistent.

Objects of class `SymbolTable` arrange for associations between symbols and nodes, i.e. objects of class `Increment`. Hence they can be used for implementing scoping rules of the static semantics of a language. They offer methods to declare a new symbol, to look up whether a given symbol has been defined, to retrieve a node associated with a given symbol and to delete an association. `DocumentTable` is a more specific symbol table where increments are instances of class `Document`. `DuplicateSymbolTable` is a specific symbol table that can manage duplicate symbols. It is required if violations of scoping rules are to be temporarily permitted.

5.9.2 Tool-specific Classes

The classes that have to be added to the predefined schema to complete the definition of a tool schema depend on the syntax and static semantics of the language to be supported by the tool. The syntax determines the different node types in the abstract syntax tree. Static semantics determine reference relationships that must be implemented between node types. For each node type, a separate class must be defined. The class must be given a name and be positioned in the class hierarchy. Therefore, the class must be defined as a subclass of one or more predefined classes. Then instance variables must be defined and finally method signatures and their implementations must be provided. In the following paragraphs, we discuss guidelines as to how this can be achieved for the tool-specific part of a schema. Moreover, we discuss schema integration that is required for the implementation of inter-document consistency constraint checks and their preservation.

5.9.2.1 Instance Variables

For a non-terminal class, the tool builder must declare instance variables that implement edges to abstract syntax child nodes. Most ODBSs are statically typed and, therefore, types must be defined for instance variables. For single-valued edges, the type of a variable is simply the class implementing the target node type of the edge. For multi-valued edges, variables may be used whose type is constructed from a base type by list or set type constructors. Type constructors for lists and sets exist by definition in all ODBSs [Cat93].

As an example consider the O_2C type declarations given below for nodes of types `Function` and `ParamList` from the abstract syntax graph on Page 21. `Function` includes a number of single-valued edges leading to child nodes. `ParamList` defines an ordered multi-valued edge to parameters. Class `Parameter` is in fact an abstract class from which classes `InParameter` and `InOutParameter` will inherit. By exploiting polymorphism, a function's parameter list may then contain instances of these two classes.

```
class Function inherit Increment
type tuple(read name:OpName,         class ParamList inherit OptionalIncrement
           read pl:ParamList,          type tuple(read params:list(Parameter))
           read type:UsingType,        ...
           read com:Comment)         end;
...
end;
```

Terminal classes do not have outgoing aggregation edges. Instead, an instance variable is required that stores the *lexical value*, that is the character string, which matches the respective terminal symbol of the grammar. This instance variable is inherited from the predefined class `TerminalIncrement`.

Instances of both non-terminal and terminal classes may be source or target nodes of reference edges that represent semantic relationships. Then additional instance variables have to be declared for each incoming or outgoing reference edge.

As an example, let us consider class `UsingType` from the Groupie module interface tool schema. In the figure on Page 21, reference edges connect nodes of type `UsingType` with declaration nodes of types `TypeName` or `TypeImport`. We, therefore, introduce an abstract class `TypeDecl` as super class of `TypeName` and `TypeDecl` and define instance variables in `UsingType` and `TypeDecl` in order to implement this reference edge.

```
class UsingType inherit Increment        class TypeDecl inherit Increment
  type tuple(read DefinedIn:TypeDecl)      type tuple(read UsedIn:set(UsingType))
   ...                                      ...
end;                                     end;
```

Finally, additional instance variables may be added in order to implement node attributes. These node attributes will most often be used for static semantic checks. They, therefore, store symbol tables or type information.

As an example, consider `Module` nodes from the figure on Page 21. These module nodes carry an attribute for storing scope information such as the set of identifiers that have been defined within the scope of the module. To implement this attribute, an instance variable of class `SymbolTable` may be added:

```
class Module inherit DocumentVersion
  type tuple(name:ModName,
             com:Comment,
             typ:TypeName,
             op_list:OperationList,
             imp:ImportInterface,
             DefinedNames:SymbolTable)
  ...
end;
```

In short, instance variables are used to implement syntactic and reference edges, lexical values and attributes required during static semantic checks.

5.9.2.2 Methods

Any class should define a *constructor* method, which initialises all instance variables and invokes the `init` methods of all super classes in order to initialise inherited instance variables as well. The constructor in O_2C is the `init` method, which is implicitly called by O_2. The class should also declare a *destructor* method `collapse`, which is invoked when the object is deleted. This method should delete all child increments, reference edges and free space occupied by instance variables that implement node attributes.

For non-terminal classes the following methods should also be defined:

- an `expand` method that expands the increment from the state where it represents a place holder to a template where all its child increments are place holders,
- a `parse` method that, for a given increment, examines a string passed as an argument and returns an abstract syntax tree if the string is syntactically correct,
- an `unparse` method that traverses the abstract syntax tree starting from the increment and computes a textual representation.

These methods implement access and modification operations to the instance variables that implement syntactic edges. They also access and modify the instance variable that stores the information as to whether or not an increment is a place holder.

For each terminal class the following methods should be defined:

- a `scan` method that checks for the lexical correctness of the terminal increment,
- an `unparse` method that returns the lexical value of the terminal increment.

These two methods provide the access and modification operations to the instance variable `value` which is hidden from the outside.

If a class implements a node type whose instances are source nodes of reference edges, two methods have to be added for each of these edges in order to control the value of the instance variables implementing the edge. The first method `set` implements an operation that establishes the edge. The parameter of the method is the increment to which the edge is to be drawn. It not only sets the instance variable to the object passed as a parameter, but also invokes an operation from the object in order to establish the reverse direction of the edge. The second method `delete` implements the operation to delete a reference edge. It sets the value of the instance variable to the undefined value `nil` and invokes an operation on the target of the edge in order to delete the reverse direction of the edge as well.

Each class implementing a node type that participates in static semantics or inter-document consistency constraints must include additional methods that check for violation of the constraints. The purpose of these methods is to include error descriptors in the error set attribute that any increment inherits from class `Increment`. In addition, these methods have to control the existence of reference edges and invoke the methods outlined above for this purpose. Another aim of these methods is to update symbol tables as soon as new identifiers are declared or existing identifiers are modified or deleted.

In addition to these standardised methods, a tool builder may freely decide to add particular methods that use the methods suggested above, in order to implement particular operations required from the schema. These methods may then be used to implement particular commands required from the tool.

As an example, consider the methods defined for class `Function` below. Methods `init`, `collapse`, `expand`, `parse` and `unparse` are those common to arbitrary non-terminal classes, though some of their signatures vary. Methods specific to class `Function` are the other methods which implement modification operations on a `Function`'s child increments.

```
class Function inherit Increment
type tuple(read name:OpName,
           read pl:ParamList,
           read type:UsingType,
           read com:Comment)
method
  public init (f : Increment),
  public collapse,
  public expand,
  public parse(Str:string):Function,
  public unparse:string,
  public expand_name(Str:string):boolean,
  public change_name(Str:string):boolean,
  public expand_type(Str:string):boolean,
  public change_type(Str:string):boolean,
  public expand_comment(Str:string):boolean,
  public change_comment(Str:string):boolean
end;
```

In short, terminal and non-terminal classes must export a number of methods. Some of them such as the constructor and destructor methods exist in arbitrary classes. Others should be available only for terminal (e.g. `scan`) or non-terminal classes (e.g. `parse`). In addition to these methods that should be exported by all classes, there are methods that are specific to particular increment classes. The implementation of method bodies, however, varies from class to class and cannot be defined in advance.

5.9.3 Data Integration

So far, we have only considered the schema of one tool. In a PSDE, however, many tools have to be included. If these tools have to be integrated a-priori, we will have to consider *data integration*, i.e. integration of schema definitions belonging to different tools.

This integration could be achieved in a very crude way by simply accumulating all class definitions of the various tools within a single schema. However, the immediate consequence would be that the development and maintenance of different tools would no longer be independent of, but would strongly interfere with, each other. As development and maintenance is most often carried out by a team of tool builders rather than individuals, these tool builders would have to agree on class names and inheritance hierarchies if all definitions were stored in one schema. In addition, the schema is the granularity of access control for schema changes. If all definitions were stored in a single schema, we could not prevent tool builders from changing definitions belonging to tools they were not responsible for. We, therefore, reject this option and require that all schema definitions belonging to one tool are stored in one schema.

There remain two options for the integration of different tool schemas: *horizontal* and *vertical*. In horizontal integration different tool schemas coexist and export/import schema definitions to/from another. In vertical integration, tool schemas are considered as views that are constructed on top of a common *conceptual* schema. In the rest of this subsection we consider these two issues in more detail and discuss when each of them should be applied. We finally demonstrate how horizontal and vertical data integration can be implemented using concepts offered by O_2.

5.9.3.1 Horizontal Integration

In order to implement efficient checks of inter-document consistency constraints and even automatic preservation thereof, inter-document reference edges are required. Intra-type reference edges need not be considered during data integration because the respective class definitions are included in the same schema. To define inter-type reference edges, however, class definitions contained in other schemas have to be used as types of instance variables and method parameters.

For an example, let us revisit Figure 3.1 on Page 21. There are inter-type reference edges between nodes representing entities of the E/R diagram and nodes of an architecture document representing modules. The syntax graphs for E/R diagrams are defined in schema `ER-Diagram` and the graphs for architecture documents are defined in schema `Architecture`. Below, the class definitions implementing node types `Entity` and `ADTModule` are given. The type definition of class `Entity` in schema `ER-Diagram` contains an instance variable of type `ADTModule`. Likewise, class `ADTModule` in schema `Architecture` contains an instance variable of type `Entity`. This pair of instance variables implements the reference edge between nodes of the two different document subgraphs.

```
schema ER-Diagram;
...
class Entity inherit Increment
  type tuple(read name:EntityName,
             read pos:Position,
             read ToADTSpec:ADTModule)
  method
    public change_name(Str:string):boolean,
    ...
end;
```

```
schema Architecture;
...
class ADTModule inherit Increment
  type tuple(read name:ModName,
             ... ,
             read ToEntity:Entity,
  method
    public set_entity(e:Entity),
    public change_name(Str:string):boolean,
  ...
  end;
```

To be able to define the two classes in this way necessitates the use of classes that are defined in other schemas. From an architectural point of view schemas play the role of subsystems and we have, therefore, to import classes from other subsystems. As discussed in [ES94] this should be done in a restricted way. In particular, there should be a means to restrict the visibility of classes defined in one schema in order to achieve information hiding not only for data structures of classes, but also in-the-large for schemas. We, therefore, have to define those classes as exports of a schema representing node types that participate in inter-document type reference edges. These classes may then be imported from other schemas, but all other classes of a schema are hidden.

O_2C supports this information hiding in-the-large. A schema definition can be exported and then it might be imported from other schemas. In the above example, we would declare the following exports and imports and then we could define classes `Entity` and `ADTModule` as indicated above.

```
schema ER-Diagram;                  schema Architecture;
export schema                       export schema
       class Entity,                       class ADTModule,
       name AllERDiagrams;                 name AllArchitectures;
name AllERDiagrams:DocumentPool;    name AllArchitectures:DocumentPool;
import schema Module_schema         import schema ER-Diagram_schema
       class ADTModule,                    class Entity,
       name AllArchitectures;              name AllERDiagrams;
       ...                                  ...
```

From a type-level perspective, which is most often the only one considered in architectural discussions, the measures outlined so far are sufficient. Since we deal with persistent objects in this thesis, we also have to consider an instance-level perspective. A schema might be instantiated in several databases. This means that several databases may contain objects whose classes are defined in one schema. This is particularly useful because several projects of a software house might be using the same tool and with that the same schema, but their documents might have to be stored in different databases to avoid interference with each other. If a tool has to instantiate reference edges between nodes whose implementations are stored in different databases, the tool must identify the database where to search for the node. To do so, we not only have to import schemas, but must also import a particular database from the set of databases that were instantiated from a schema.

In O_2, databases are referred to as *bases*. In addition to schema import, O_2C provides the concept of a base import to address other databases. The O_2C statements given below display the required imports for the example given above. The statements create new databases for entity relationship diagrams and ADT module definitions for two projects `A` and `B`. The `set base` command establishes a database as current base and then all changes to the database apply to the current base. Hence, the tables for storing documents that are created next are stored in the two databases of project `A`. The import base commands then enable the `ER-Diagram` database of project `A` to have references to documents in the `Module` database of project `A`. References between the entity relationship diagrams of project `A` and modules of project `B`, however, are inhibited by these definitions.

```
schema ER-Diagram;                   schema Architecture;
name AllER-Diagrams: DocumentTable;  name AllArchitectures: DocumentTable;
export schema class Entity;          export schema class ADTModule;
       name AllER-Diagrams;                 name AllADTModules;
import schema Module_schema          import schema ER-Diagram_schema
       class ADTModule,                     class Entity,
       name AllADTModules;                  name AllER-Diagrams;

create base ER-Diagram_ProjA;        create base Module_ProjA;
create base ER-Diagram_ProjB;        create base Module_ProjB;
set base ER-Diagram_ProjA;           set base Module_ProjA;
AllER-Diagrams:=new DocumentTable;   AllADTModules:=new DocumentTable;
import base Module_ProjA;            import base ER-Diagram_ProjA;
...                                  ...
```

In short, horizontal data integration requires, from a type-level point of view the use of classes and names that are defined in other database schemas. Therefore, the schema definition language must offer import and export statements to allow for information hiding in-the-large. From an instance level point of view, a schema might be instantiated in several databases. Then we must define, for a database, to which other databases it refers.

5.9.3.2 Vertical Integration: Views

As mentioned in Section 3.3, there are document types, such as module interface definitions and successive implementations, which include redundant information and should, therefore, be implemented using views. The rationale for using a view mechanism for vertical data integration is then to have several virtual abstract syntax graphs definitions for the different document types with slightly different node and edge types and different available operations, while only one common representation of the graph is stored persistently. In this subsection, we discuss how the common persistent representation should be defined and how views should be constructed on top of such a representation.

The structure and available operations of virtual abstract syntax graphs are implemented in a virtual schema in the same way as discussed for plain schemas. The rest of the tool architecture, therefore, will not be affected by the fact that virtual syntax graphs are not stored persistently. Thus, each virtual node type is implemented in a virtual class of the virtual schema. Each edge type starting from the virtual node type is declared by a pair of instance variables in the connected virtual classes.

The structure of the common persistent representation of several virtual abstract syntax graphs must be defined in a common conceptual schema. Therefore, a base class must be available in the conceptual schema for each virtual class of each view that will be constructed on top. Note that several virtual classes may have the same base class, also there may be base classes that are only used in some views, while they are hidden from other views. Moreover, instance variables must be available in each base class for each instance variable of those virtual classes that were derived from the base class. Again one instance variable of the base class might be used for instance variables of several virtual classes.

A virtual schema is then defined based on such a conceptual schema. Virtual classes are derived from base classes. During that process instance variables of the base class, which should not be visible in the virtual class, are hidden. Operations that were available in the base class can be hidden as well. In addition, the virtual class definition can define new operations, for instance to implement different parsing or unparsing schemes for the virtual abstract syntax graph.

To illustrate the use of views in the O_2 ODBS, we define two views on top of a conceptual schema as an example. The views partly implement the two virtual syntax graphs depicted in Figure 3.2 on Page 25. The conceptual schema defining the structure of the real syntax graph of that example contains the class definitions given below.

```
schema Module_schema;
base Module_base1;

name AllModules:DocumentVersionTable;

class OperationList inherit Increment         class Operation inherit Increment
type tuple(opl:list(Operation))               type tuple(name:OpName,
method                                                   pl:ParamList,
  public init(f:Increment)                               com:Comment,
  public collapse,                                       vd:VarDeclList,
  public expand,                                         st:StatementList)
  public add(o:Operation,after:Operation),    end;
  public insert(o:Operation,before:Operation)
end;

class Procedure inherit Operation             class Function inherit Increment
method                                        type tuple(typ:UsingType)
  public init(f:Increment),                   method
  public collapse,                              public init(f:Increment),
  public expand,                                public collapse,
end;                                            public expand
                                              end;

class StatementList inherit Increment         class VarDeclList inherit Increment
type tuple(sl:list(Statement))                type typle(vl:list(Statement))
method                                        method
  public init(f:Increment),                     public init(f:Increment),
  public collapse,                              public collapse,
  public expand,                                public expand,
  public add(s:Statement,after:Statement),      public add(s:VarDecl,after:VarDecl),
  public insert(s:Statement,before:Statement)   public insert(s:VarDecl,before:VarDecl)
end;                                          end;
...
```

Based on this conceptual schema, consider an excerpt of the definition of virtual classes in two views. The first is a view for the module interface tool and the second defines a view for the module implementation tool displayed in Figure 2.1. The two view definitions implement the virtual node types displayed in Figure 3.2.

```
virtual schema Interface_view          virtual schema Source_view
        from Module_schema;                    from schema Module_schema;
virtual class IOperationList           virtual class SOperationList
        from OperationList;                    from OperationList;
        inherit Increment; ...                 inherit Increment; ...
virtual class IOperation               virtual class SOperation
        from Operation;                        from Operation;
        inherit Increment;                     inherit Increment; ...
        hide attribute vd,st; ...
virtual class IProcedure               virtual class SProcedure
        from Procedure;                        from Procedure;
        inherit IOperation; ...                inherit SProcedure; ...
virtual class IFunction                virtual class SFunction
        from Function;                         from Function;
        inherit IOperation; ...                inherit SProcedure; ...
hide class VarDeclList;                virtual class SVarDeclList
                                               from VarDeclList;
                                               inherit Increment; ...
hide class StatementList;              virtual class SStatementList
                                               from StatementList;
                                               inherit Increment; ...
```

The first two lines declare two new views, namely `Interface_view` and `Source_view`, based on the common schema `Module_schema`. The declaration of `Interface_view` hides classes `VarDeclList` and `StatementList` from the view. Consequently, the instance variables `vd` and `st` implementing syntactic edges between operations and variable declarations or statement list nodes, respectively, are hidden in the view as well. On the other hand, view `Source_view` implements virtual node types for variable declarations and statement lists, as they occur in syntax graphs of programming languages. Consequently, `Source_view` does not hide the respective instance variables from class `SOperation`.

As an example of how different methods are defined for two virtual classes, we consider the two virtual classes that were derived from class `Function` in more detail.

```
virtual class IFunction                          virtual class SFunction
        from Function;                                   from Function;
        inherit IOperation;                              inherit SProcedure;
        method                                   method
          public unparse:string,                   public unparse:string,
          public parse:string,                     public expand_var_decl,
          public expand_type(s:string):boolean,    public expand_stmt_list,
          public change_type(s:string):boolean   end;
        end;
end;
```

Both virtual classes declare an unparse method with the same signature. The implementations of these methods, however, differ in that they implement different unparsing schemes of functions. Moreover, the view definitions enforce changes to interfaces of functions to be performed in the design, since `Source_view` does not offer any methods for changing a function's interface. The `Source_view` definition of a function, however, does provide methods for expanding variable declarations or the statement list implementing the body, which are not available in the interface view.

Each view must declare a counterpart to the named object defined in the conceptual schema in order to get a common root for all virtual abstract syntax graphs. This root is, for instance, used as a starting point for navigations through the virtual abstract syntax graph. This starting point is defined as a virtual named object in the view. Its contents may be restricted by an object-oriented query to a subset of real objects. During this query, real objects are transferred into virtual objects by conversion functions.

The example below defines the virtual named object for the `Interface_view`. It selects all modules that exist in the database, translates them into modules of the interface view and includes them in the virtual named object `AllInterfaces`.

```
virtual schema Interface_view from Module_schema;
virtual name AllInterfaces:set(IModule);
has value
    select As_IModule(d)
    from d in AllModules->VALUES();
```

We can also explain now why virtual abstract syntax graphs, which are implemented with views as outlined above, can be updated. The required updates are the creation of a new virtual node, changes of virtual node attributes, the creation and the deletion of a virtual edge and the deletion of a virtual node. The creation of a virtual node requires the creation of a new virtual object in the view. To create the new object we apply the `new` operator to a virtual class, for instance `IFunction`. The `new` operator invokes the constructor. The view mechanism, in turn, has generated the constructor for `IFunction` in such a way that it first creates a new object of the base class `Function`. This creation implies the execution of the constructor of `Function`, which, in turn, properly initialises all instance variables of the base object. That means that, during the creation of a virtual object, not only the instance variables for the virtual object, but also the instance variables that are hidden in the view are properly initialised. In the example, the constructor would also initialise instance variables `vd` and `st`, although they are not visible in the view. In RDBS views, this is not the case due to the lack of appropriate object constructors. Finally, the constructor of the virtual class stores a reference to the base object in the virtual object. This reference is, in fact, the only instance variable that is really stored in the virtual object. Any other instance variable access or modification is mapped to an instance variable access or modification of the real object, which is identified by the stored reference. Node attribute changes and the creation and deletion of edges are implemented in terms of such instance variable modifications. They persist because they are effectively mapped to changes of instance variables in the real object. Suppose, for example, that we want to create a new edge between a virtual function and a virtual parameter list in the interface view. We then have to assign the virtual parameter list object to instance variable `pl` of the virtual function. The view mechanism will effectively assign the base object of the virtual parameter list to the instance variable `pl` of the base object of the virtual function. In this way the update persists in the database and is also visible in the implementation view. Deletion of a virtual object from the view is simply done by assigning the undefined value of nil to the reference that referred to the object. In the persistent representation, the garbage collector of the database takes care that any objects without references to them are deleted.

In short, the view mechanism of O_2 facilitates vertical data integration. Using this mechanism, a number of tools can share a conceptual schema in a controlled way. While sharing the schema, tools also share the persistent representation of syntax graphs. Using the O_2 view mechanism, it is then possible to declare different virtual schemas based on the conceptual schema, one for each tool, and implement different structures and behaviours of the same persistent representation.

5.9.3.3 Combining Horizontal and Vertical Integration

Given that there are two methods for data integration, we are faced with the problem of when to use which. In principle, vertical data integration could be implemented with horizontal integration. We would have to materialise basically all the virtual syntax graphs that we would have in the case of vertical integration. Therefore, we would have to export each node type from one schema, import it to the other schemas and vice versa. We would then have to create objects and change instance variables in the materialisations of all virtual syntax graphs, whenever the respective objects and instance variables materialising some other virtual syntax graph are changed. If the schemas and their instantiations in bases are, from a structural point of view, very similar there are a number of serious implications. Firstly, tool schemas will be very hard to maintain since a change in the structure of one schema affects all other schemas. Secondly, tools will perform slower as they always have to perform updates in implementations of other virtual syntax graphs. Finally, physical disk space utilisation will be worse than with vertical integration, since each object is physically duplicated as often as it occurs in different syntax graphs.

On the other hand, vertical integration cannot be used to implement horizontal integration. Using import/export statements, we do not impose any strategies concerning when to propagate a change into a document of another type. In the example given above, changing a name of an entity does not imply an immediate change to the name of the ADT specification of that entity. For instance, it may well be deferred in order to give some other developer, who is in charge of the ADT specification, the chance to reject that change. This is not possible if syntax graphs are virtual and implemented with views. If a node is changed, the change is visible, after commit of the transaction that performed the change, in all the counterparts of that node in other virtual syntax graphs. In the example, if class `Entity` was a virtual class and `ADTModule` was another virtual class and both were derived from the same base class, then the change of the name of a virtual object of class `Entity` would have been visible immediately in the corresponding virtual object of class `ADTModule`.

This means that views cannot be used for data integration if changes made to nodes are not to be propagated immediately into other nodes. If immediate visibility of changes is required, however, views are the preferred means of integration. In practice, therefore, a *hybrid data integration* approach that combines vertical and horizontal integration will be used in most PSDEs. If we take above examples, the most beneficial integration of the schema, for the entity relationship editor with the schema for the module interface editor and the source code editor, is the following. The interface and source code editors both declare a view each based on a common conceptual schema and, therefore, share their physical syntax graph representations. The entity relationship editor schema is integrated with the former two by importing from and exporting to the conceptual schema of the two other editors. Thus, changes of an entity name can be performed without immediately changing the module name.

5.9.4 Summary

We have seen that the schema of a tool can be divided into a reusable and a tool-specific part. The reusable part contains class definitions which implement properties that are common to arbitrary tools. These classes can be used by inheritance or declaration of instance variables in classes contained in the tool-specific part. As tool-specific classes vary from tool to tool, we have discussed how to identify these classes based on the syntax and static semantics of the underlying languages and have suggested strategies for defining inheritance, instance variables and methods of these classes. In the last paragraphs, we have discussed how to integrate schemas of different tools in order to implement inter-document (type) consistency constraints. We have suggested a horizontal and a vertical strategy for data integration and have discussed when the strategies are to be employed and how they can be combined.

5.10 Related Work

A number of syntax-directed tools have been discussed in literature. Among the first were tools developed in the Gandalf project [HN86] (e.g. IPE [MF81], GP [HN86] or GNOME [GM84]), the Cornell Program Synthesizer [RT81] or tools developed in the Mentor project [DGHKL84]. The architectures of all these tools contain a subsystem of some sort that maintains abstract syntax trees of documents in main memory. These subsystems offer operations in order to dump syntax trees to a persistent representation on an operating system file and to restore trees from such a representation. Working on a syntax tree representation in main memory that is not the persistent representation has a number of serious consequences. First of all, checking for inter-document consistency constraints cannot be based on edges that span between documents. Therefore, these tools do not adequately address the issue of checking for consistency between different types of documents. Sometimes different unparsing schemes are used to display different document types. In this case, however, changes made to one document are immediately visible in its corresponding document, which may not always be appropriate. Secondly, updates of concurrent users that affect the document a user is working on cannot be handled appropriately, as there is no common persistent representation. Moreover, tools are not tolerant of hardware or software failures and users might lose significant effort in the case of a failure. Finally, none of these tools support version and configuration management of documents. Different revisions of a file representation of an abstract syntax could be maintained by controlling them with basic versioning mechanisms such as SCCS or RCS [Tic85]. Then, however, neither predecessor and successor relationships between documents nor consistency of configurations can be maintained by the tools.

For tools contained in the IPSEN environment [Nag85, ENS87], a fundamentally different architecture is chosen [Sch86]. First of all, the tools do not operate on a transient document representation, but work directly on the persistent representation managed by the GRAS database system (c.f. Subsection 4.3.1). All documents are stored in a single GRAS graph. Each document is represented, in turn, by a subgraph with reference edges leading to other documents' subgraphs, thus facilitating efficient checks of static semantics and inter-document consistency constraints. The GRAS transaction mechanism is used and, therefore, loss of effort in the case of failures is restricted to the last completed command. Revisions of documents are supported in a later version of the IPSEN prototype [Wes89b] based on the functionality offered by the GRAS database system. Configurations that select different versions of a document, however, are not supported. Concurrent editing by multiple users is not possible in IPSEN, since all documents are stored in one graph which has to be locked exclusively as soon as a user wants to modify a document. If documents were stored in separate graphs in order to have a finer granularity of locking, reference edges between different graphs and exploitation of the transaction mechanism would be inhibited due to the limitations of the GRAS database (c.f. Page 66). Further drawbacks are the lack of schemas and views in GRAS. Finally, the IPSEN architecture is not open. It might be extended with a subsystem for service execution, but then concurrency control problems might occur since services have to be executed concurrently to editing sessions. This is not possible with the GRAS database if both service and editing sessions have to access the same graph.

The problem of openness is, to a certain extent, solved by the architecture of the Field programming environment [Rei90]. The environment contains a number of tools including textual editors and graphical viewers for source code. These tools are open and communicate with one another based on a message passing subsystem. They send a message of a particular type to a broadcast message server[4] which, in turn, broadcasts the message to any other tool which has registered as being interested in messages of that type. The problem with Field and its broadcast message server is that it is still a single-user solution as the server cannot route messages to remote workstations. In addition, exchanging messages in no way solves the problem of different users concurrently changing related documents.

5.11 Summary

The architecture developed in this thesis overcomes all the above deficiencies by storing abstract syntax graph representations of documents in object databases. The structure of, and available operations on, these syntax graphs are defined and controlled by the `ToolSchema` subsystem, defined in an object-oriented schema definition language. Version management of documents is supported, based on version management primitives for composite objects, which are to date only offered by Orion and O_2. Simple support for configuration management is implemented in the `ToolKernel` of the tool architecture. Data integration of tools, i.e. measures for preserving inter-document (type) consistency constraints, is based on export/import between different schemas, an object-oriented view mechanism or a combination of these.

Communication between tools and the process engine is done via a communication protocol built on top of a message router. Commands are executed as ACID database transactions, which are controlled by the `ToolKernel` in order to secure the integrity of the syntax graph against failures, as well as arranging to save, concurrent updates of related documents. Distributed execution of tools is supported, though not reflected in the tool architecture. It is transparent to tool builders since it is achieved by the client/server architecture of current object database systems.

During the course of this chapter, we have separated components of the tool architecture into reusable and tool-specific components. The reusable components are the `Control` class, the `ToolKernel`, `SoftwareProcessCommunicationProtocol` and `UserInterface` subsystems and some of the classes contained in the `CommandExecution`, `ToolAPI` and the `ToolSchema` subsystems. The `ODBS`, the `MessageRouterInterface` and the `UIMS` subsystems are third party components and have been reused in any case. The reusable components can now be reused in any tool architecture. This significantly simplifies the tool construction process. Nevertheless, there are a number of components which still have to be constructed anew for each tool. These are the `LayoutComputation` class, the `ToolSpecificServices` class and most of the classes in the `CommandExecution`, `ToolAPI` and `ToolSchema` subsystems.

[4]The Broadcast Message Server product from Hewlett Packard was, in fact, built according to the architecture of the server in the Field Environment.

Chapter 6

The GOODSTEP Tool Specification Languages

In the last chapter we identified the architecture components that are tool-specific. They are tool-specific because they depend on the syntax and static semantics of the language supported by the tool, the consistency constraints to other documents and the particular editing, analysing and browsing commands that are to be offered by the tool. Although a significant amount of code for architecture components can be reused, the implementation of the tool-specific components is still far too time consuming. We thus suggest a number of domain-specific graphical and textual languages that a tool builder can use to systematically and effectively engineer the tool-specific components. We call these languages GOODSTEP Tool Specification Languages (GTSL)[1].

These specification languages must be able to express the different concerns that vary from tool to tool. They should provide different levels of abstraction in order to support a tool builder in the step-wise refinement of a tool specification. The highest-level language should provide a concise *overview* of the tool specification components. This overview will aid the tool builder in getting a specification development started. We are going to define an extended and normalised BNF that serves this purpose. The productions of the BNF will then be used to identify *increment classes*. A lower-level language will allow a tool builder to define the *structure* of increment classes, such as semantic attributes and relationships with other increment classes. We will customise OMT entity relationship diagrams for that purpose. At the lowest level of abstraction, we will then provide a language for the definition of increment class *behaviour*. This language will enable increment layout and increment-specific commands as well as static semantics and inter-document consistency constraints of increment classes to be defined.

The specification languages will provide primitives for reusing and customising predefined increment class specifications[2]. Customised specifications will, in turn, be reusable in the same way as predefined specifications. The specification languages that we suggest will follow the *object-oriented paradigm* in order to achieve this. The structural and behavioural concerns defined in increment classes are inherited by subclasses and are thus reused there. Classes can

[1]The languages that we present here have been developed and evaluated within the ESPRIT-III Project 6115 (GOODSTEP) [GOO94]. We dedicate the name of the languages to this project to acknowledge the benefits that we gained from working in its stimulating scientific atmosphere.

[2]These are, in fact, the specifications of the classes that are contained in the predefined schema.

be customised to address more specific problems by redefining concerns. The effort required for the specification of a tool will then merely depend on the degree to which the required properties differ from predefined properties.

In the next section we will discuss the requirements that the different languages must meet. In Section 6.2, we will suggest a language for defining extended and normalised BNFs (ENBNFs) so as to define the concrete syntax of a language that is supported by a tool. Section 6.3 suggests customisations to the OMT entity relationship model to make it particularly suitable for the structure definition of increment classes. An entity relationship model then appropriately visualises navigation paths between increments that are required for defining static semantics and inter-document consistency. The detailed behavioural increment class definition is structured into an external *class interface* definition and an internal *class specification*. Section 6.4 discusses class interface definition, while we present the language primitives for class specifications in Section 6.5. We conclude this chapter with a comparison with related work where we indicate the contributions of this thesis to state-of-the-art tool specification.

In the next chapter, we will discuss a tool builder's requirements for an integrated development environment for tool specification. This environment will contain a tool for each of the languages that are discussed in this chapter. Moreover, the environment will check and even preserve inter-document consistency constraints that exist between the different languages. We cannot reasonably discuss these inter-document consistency constraints in this chapter, since their definition can only be understood after all the different languages have been introduced.

6.1 Requirements of Tool Specification Languages

In order to set the scene for this chapter, we have to discuss language design issues, i.e. we must define the requirements that the different specification languages must meet. The languages should be specific to the domain of tool specification in order to support tool builders as much as possible. This implies that the languages can express all the information required to generate the tool-specific components of the tool architecture as identified in the last chapter. Only then will it be feasible to build a compiler that generates executable tools from the specification without having to code further components manually. The languages should, therefore, provide dedicated language constructs for the following concerns.

Abstract Syntax: A tool specification for syntax-directed tools must include a definition of the language supported by the tool. The syntax of a formal language is usually defined by a grammar given in terms of an EBNF. Production rules define the component increments for each increment of the language. On a lower level of abstraction, the specification of the structure of increments should be distinguished from the specification of the concrete increment representation in order to allow for a more flexible adoption of document layout or even definition of several document layouts [KLM83]. The syntactic structure of a language is denoted as *abstract syntax*. To define the abstract syntax, a tool specification language must be capable of defining the different increment types and their component increments. Moreover, it must include appropriate primitives for specifying increment lists and optional increments.

Unparsing Schemes: Since the abstract syntax only specifies the internal structure of a language, the external representation of the language must be defined elsewhere. This external representation must be defined in terms of *unparsing schemes*. The unparsing schemes define the layout of documents with line breaks and indentations. Moreover, they define the concrete syntax in terms of keywords and an order in that abstract syntax children occur in the external representation.

Static Semantics and Inter-Document Consistency Constraints: Tool specifications must define *static semantics* such as scoping and type compatibility rules of a language. Moreover, inter-document consistency constraints need to be defined. From a structural point of view, these constraints define additional attributes and relationships between increments. From a behavioural point of view, the specifications must determine how attribute values are computed and the way relationships are established. Besides defining the constraints themselves, specifications must also define whether violations of static semantics or inter-document consistency constraints are tolerable or not. If errors are tolerated, specifications must also define the error messages that are presented to the user. In order not to complicate the specification language unnecessarily, the concepts defined for definition of static semantics should also be applicable to the specification of inter-document consistency constraints.

User Commands: Besides generic commands, such as opening or closing a document and copying or pasting increments, tools will have to offer increment specific *commands*. Hence the tool specification languages must be able to define these commands. The language must be capable of expressing the functionality of commands, the preconditions for their appearance in a context-sensitive menu and also the user-dialogue that is carried out during the execution of the command. As we have required user commands to be the unit of concurrency control and persistence, the tool specification language should offer transaction control statements such as commit or abort, for specifying the success or failure of a command.

Structure and Reuse: The above paragraphs have identified the different concerns that a tool specification must define. The complexity of these concerns is so great that the languages must provide structuring facilities so that a tool specification can be properly structured into different independent components. Among these components it will be possible to identify components that are similar in structure and behaviour. The specification languages should, therefore, allow the tool builder to identify these similarities and to specify common structure and behaviour in one component and reuse it for similar components. The object-oriented paradigm appropriately meets both requirements. It can be used to structure the overall definition into components, namely classes, and can define inheritance relationships to reuse properties in subclasses.

Level of Abstraction: In the last chapter we exploited the concepts of classes and inheritance relationships of object-oriented schema definition languages for structured and reusable schema components. We must, however, consider schema definition languages as not providing the appropriate level of abstraction for specifying a tool. From a structural point of view, pairs of instance variables are not the appropriate for defining relationships among increments. From a behavioural point of view, tool builders would have to code unparsing schemes, multiple-entry parsers and checkers for static semantics and inter-document consistency constraints in terms of methods, which means in an imperative way. Instead, tool builders demand higher-level specification languages where they can declaratively describe abstract syntax, unparsing schemes, tool commands, static semantics and inter-document consistency for a tool and have a compiler generating the implementation in the schema definition language. The different specification languages should be at different levels of abstraction. The highest-level language should provide a concise overview of the components. At a lower-level, the structure of each component should be refined, and at the lowest-level the behaviour of components should be declared. In the next section we will suggest a language for the highest level of abstraction, which is, therefore, particularly useful as a starting point for a tool development since it identifies the different components of the specification.

6.2 Extended, normalised Backus-Naur Forms

The syntax of a formal language is usually defined by a grammar of some sort. In languages that occur in software engineering practice the syntax is *context-free*. We, therefore, need not consider languages at higher-levels of the Chomsky hierarchy. Most often a Backus-Naur Form (BNF) is used for defining context free grammars. We found plain BNFs to be not expressive enough. As already discussed in [ES89] their way of defining list, alternative and optional increments is not very concise. We, therefore, extend BNFs to deal with these concerns more efficiently. For the later use of extended BNFs, in particular as an input for the generation of specifications at lower levels of abstraction, it is necessary to normalise the extended BNFs. The language definition for our extended and normalised BNF (ENBNF) is displayed in Figure 6.1. The notation is plain BNF.

```
<enbnf>           ::= <production-list>
<production-list>::= <production>
<production-list>::= <production> <production-list>
<production>      ::= <symbol> "::=" <alternative>
<production>      ::= <symbol> "::=" <structure> "."
<production>      ::= <symbol> "::=" "|" <structure> "."
<production>      ::= <symbol> ":" <reg-exp> "."
<production>      ::= <symbol> ":" "|" <reg-exp> "."
<alternative>     ::= <symbol> "|" <alternative>
<alternative>     ::= <symbol> "|" <symbol>
<structure>       ::= <component-list>
<component-list> ::= <component>
<component-list> ::= <component> <component-list>
<component>       ::= <keyword>
<component>       ::= <symbol>
<component>       ::= <list>
<list>            ::= "{" <symbol> "}" <opt_delimiter>
<opt_delimiter>  ::=
<opt_delimiter>  ::= "(" <keyword-list> ")"
<keyword-list>   ::= <keyword>
<keyword-list>   ::= <keyword> <keyword-list>

<keyword>         :  ["].*["]
<reg-exp>         :  ['].*[']
<symbol>          :  [a-zA-Z_][a-zA-Z_-0-9]*
```

Figure 6.1: Syntax Definition of ENBNFs

An ENBNF consists of a list of *productions*. The normalisation of the EBNF is enforced by the language syntax. It defines five different kinds of productions: *alternatives*, *structures*, *optional structures regular expressions* and *optional regular expressions*. A symbol is *terminal*, if it is defined by a production with a regular expression on the right-hand side, otherwise it is *non-terminal*. Alternative productions provide for a choice of a number of non-terminal symbols. Note that structures or lists as alternatives are not supported in this normalisation. Structure productions contain symbols and keyword definitions. A structure production is a *list production* if it contains an element that has been produced by production `<list>`. This list might then define delimiter items that separate several list elements. A static semantic constraint will ensure that only further keywords are included on the right-hand side of these productions. Regular expressions define the lexical syntax of a terminal symbol. Finally, optional productions provide for a choice between the empty string and a structure or the empty string and a regular expression.

Obviously syntax definitions defined in ENBNFs must obey certain static semantic constraints. These are as follows:

SV1: Each symbol used on the right-hand side of a production must be declared by a production where it appears on the left-hand side, otherwise the grammar would be incomplete.

SV2: Symbols must be declared only once, otherwise there would be alternatives in the grammar that are not defined as alternative productions in the ENBNF.

SV3: If a structure production contains a list on the right-hand side, there must not be any other symbols included in the component list of that production.

Throughout this chapter, we will take examples from the Groupie module interface editor. An excerpt of the syntax of the module interface definition language supported by Groupie is defined as an ENBNF below.

```
Module        ::= ADTModule | FModule | ADOModule | TCModule .
ADTModule     ::= "DATATYPE" "MODULE" ModName ";"
                    Comment
                    "EXPORT" "INTERFACE" "TYPE" TypeName ";" OperationList
                    "END" "EXPORT" "INTERFACE" ImportInterface
                  "END" "MODULE" ModName "." .
OperationList ::= {Operation} .
Operation     ::= Function | Procedure .
Function      ::= "FUNCTION" OpName ParameterList ":" UsingType ";" Comment .
Procedure     ::= "PROCEDURE" OpName ParameterList ";" Comment .
ParameterList ::= | "(" {Parameter}(";") ")" .
Parameter     ::= InParameter | InOutParameter .
InParameter   ::= "IN" ParName ":" UsingType .
InOutParameter::= "INOUT" ParName ":" UsingType .
...
ModName       ::= '[A-Za-z][A-Za-z0-9]*' .
ParName       ::= '[A-Za-z][A-Za-z0-9]*' .
TypeName      ::= '[A-Za-z][A-Za-z0-9]*' .
OpName        ::= '[A-Za-z][A-Za-z0-9]*' .
UsingType     ::= '[A-Za-z][A-Za-z0-9]*' .
Comment       ::= | '/"*"([^*/]|[^*]"/"|"*"[^/])*"*"/' .
...
```

The alternative production `Module` defines the four different types of modules supported by the Groupie module interface definition language. The next structure production defines the syntax of an ADT module in more detail. In particular, an ADT module has children for the module name, a comment, an exported type, an exported operation list and an import interface. As examples of list productions, consider operation lists and parameter lists. Notice how we used the delimiter construct to define the way parameters are delimited. Note also that parameter lists as well as comments are optional productions. Finally, comments and several kinds of identifiers are defined by a number of regular expression productions.

Summarising the discussion, ENBNFs are suitable for defining the abstract and concrete syntax of a language that is to be supported by a syntax-directed editor. The ENBNF defines the syntax in a very concise way and in addition the different productions identify the increment classes that have to be refined. It thus serves, at a very high-level of abstraction, as a starting point for a tool development. However, it does not yet define any semantic relationships. We, therefore, continue at a lower level of abstraction with the definition of a language that serves this purpose.

6.3 OMT Entity-Relationship Diagrams

An ENBNF definition can be used to generate an abstract syntax tree definition, which defines the abstract syntax of the language at a lower level of abstraction. Each ENBNF production represents a node type of the tree and each symbol on the right-hand side of a production represents an outgoing aggregation edge to a child node. The target node type of an outgoing edge is determined by the respective symbol. The edge may be heterogeneous, i.e. lead to different node types, if the symbol is defined by an alternative production. For a list production, the outgoing edge is an ordered multi-valued collection of edges. Due to the normalisation of the ENBNFs, these are the only cases that we have to consider.

One could argue that the ENBNF definition is obsolete because an abstract syntax tree definition would be sufficient to define the syntax. We object to this argument since, firstly, an abstract syntax tree definition does not define the concrete syntax. Secondly, the ENBNF provides the syntax definition in a very concise way, and thirdly, the ENBNF is an excellent starting point for a tool development since it guides the tool builder towards the identification of node types.

It has been concluded in Section 3.1 that abstract syntax trees are insufficient for the efficient implementation of static semantics and inter-document consistency checks. Therefore, we have generalised abstract syntax trees to abstract syntax graphs by introducing reference edges. Thus, we have to develop facilities to support a tool builder in the enhancement of an abstract syntax tree definition, which will be generated from the ENBNF, with semantic relationships and attributes. Besides defining these relationships and attributes, the language to be defined should also allow for a concise graphical overview of navigation paths that will occur in the definition of consistency constraints. Moreover, the static semantics should be defined in a way that specification errors are excluded as far as possible.

Entity relationship models[3] can be used to appropriately define these abstract syntax graphs. They are graphical notations and are appropriate for visualising graph structures. Entities can be considered as types that, in turn, provide the basis for the definition of a type system. The type system can then contribute to the detection of specification errors. Entities can be used to represent the different node types that occur in a syntax graph and relationships model the

[3]In fact, we have already used the model successfully for the structure definition of abstract syntax graphs during the development of the Merlin Benchmark.

different types of edges. The entity relationship model as defined in [Che76], however, does not give adequate support if we consider heterogeneous or ordered multi-valued relationships. Since these are common problems, numerous proposals have been made to extend the original approach. From these variants, the OMT entity relationship model [RBP$^+$91] gives the best support for our problem because it

- distinguishes between aggregation and reference relationships,
- includes ordered multi-valued relationships,
- suggests multiple inheritance of attributes and relationships and
- supports polymorphism.

The distinction between aggregation and reference relationships is required because syntactic relationships have aggregation semantics, whereas semantic relationships have reference semantics (Refer to the discussion of aggregation and reference edges on Page 20). Ordered multi-valued relationships are required for modelling multi-valued collections of syntactic edges. Inheritance of attributes and relationships increases the modelling power significantly and finally polymorphism enables heterogeneous edges, that is edges that lead to different node types, to be defined.

From now on we refer to entities as *classes* to emphasise the object-oriented nature of entities in our context. Classes are depicted as rectangles in the entity relationship model. Attributes of a class, if any, are depicted in the lower half of the rectangle that represents the class. Relationships are drawn as arrows. Solid arrows represent the aggregation relationship between classes. Dashed arrows, in turn, represent reference relationships. Arrows start from and lead to circles that are connected to classes. The colour of these circles determines the cardinality of the relationship. A white circle denotes a one-end and a black circle denotes a multi-valued relationship. Circles drawn on an arrow show that a multi-valued relationship is ordered. We subsume attributes, incoming and outgoing relationships under the term *property* in the following paragraphs. Lines with triangles denote inheritance relationships between classes: The class(es) below the triangle inherit from the class(es) the triangle points to. Classes have a name, and relationships have a first name and an optional second name. The first name denotes the navigation in the arrow direction and the second name denotes the reverse direction. First and second names are delimited by a slash. Aggregation relationships model syntactic relationships only. Each increment is, therefore, reachable from exactly one other increment via an aggregation relationship. We can, therefore, omit the second name for aggregation relationships and refer to it uniquely as *father* if required.

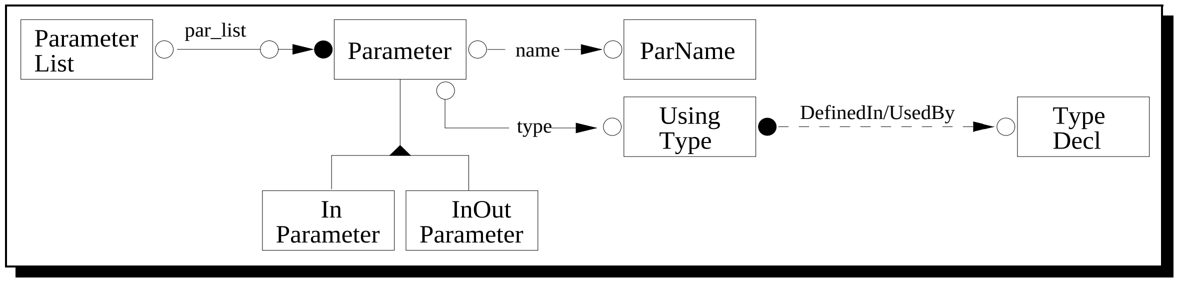


Figure 6.2: An OMT Entity Relationship Diagram

Figure 6.2 displays how we exploit the OMT extensions for the definition of abstract syntax graphs. The example is taken from the entity relationship model of the Groupie interface editor. Note how classes and aggregation relationships have been directly computed from the respective ENBNF productions. The names of aggregation relationships identify the syntactic edges that start from an increment. They cannot be derived from the ENBNF but have to be added in the entity relationship model. The reference relationship models the semantic relationship between usage and declaration of types. The relationship `par_list` is an ordered multi-valued relationship and models the multi-valued edge between a parameter list and its several children. A parameter, in turn, can be an in parameter or an in-out parameter. This alternative is modelled by the inheritance relationship. Note that both `InParameter` and `InOutParameter` inherit the aggregation relationships `name` and `type` from `Parameter`. Moreover, we exploit polymorphism, since instances of both subclasses of `Parameter` can be included in the multi-valued relationship `par_list`.

Even though the extensions of the OMT entity relationship model are very useful, the notation is still not fully appropriate for the purpose of modelling abstract syntax graph structures. We have to customise the notion to provide a tool builder with adequate modelling power. Our modifications are

1. definition of application-specific static semantic constraints that exclude a number of specification errors,
2. definition of precise static semantics for multiple inheritance,
3. introduction of hierarchical decomposition,
4. introduction of covariant redefinition of relationships and
5. removal of obsolete language constructs.

Static Semantics: Obviously, an entity relationship model must respect certain static semantic constraints. Class names must be unique in order to be able to uniquely identify classes by their name. Property names must be unique among the properties that are defined for a class. Aggregation relationships model syntactic edges. If we only consider aggregation relationships, the entity relationship model defines the structure of a syntax tree. Therefore, aggregation relationships must not start at a multiple end, otherwise the structure definition would enable increments with more than one father to occur. Multi-valued aggregation relationships must be ordered because they embody ENBNF list productions. We can exclude unordered aggregation relationships because a user of a tool will become confused if the tool does not keep the order of list elements as they were inserted by the user. The inheritance relationship must be acyclic, otherwise a class could be a subclass of itself and late binding might not terminate.

Multiple Inheritance: We have seen that polymorphism is used to model heterogeneous edges. In the behavioural specification of increment classes, edges will be created or changed by assigning increments to *variables*. Polymorphism will be restricted by the inheritance hierarchy, in that assignments to variables are only correct if the static type of the variable is a super type of the static type of the expression that is assigned. This polymorphism rule is in place in most typed object-oriented languages (e.g. Eiffel, C++ or Beta). It contributes to type safety since it ensures that properties defined for the static type of a variable are available for objects the variable refers to at run-time.

We might now have to use a class in polymorphic assignments with respect to different contexts. As an example consider types in the Groupie interface editor. A type declaration in an interface definition can be made in terms of either exporting or importing a type. In both ways we declare a type that can then be used in a parameter list for example. We have to distinguish the different type increments, because type name increments denoting an imported type must not occur in an export interface and, vice versa, type name increments denoting an exported type are not allowed in import lists. For creating the semantic relationship `DefinedIn/UsedBy` (c.f. Figure 6.2), however, we should not be aware of the distinction. We, therefore, declare `TypeName` and `TypeImport` as subclasses of class `TypeDecl`, which, in turn, is the target class for `DefinedIn/UsedBy`. Exploiting polymorphism, both kinds of type name increments can then be used as target increments at run-time.

Imports that occur in import lists in Groupie interface definitions denote either type imports or operation imports. These have to be distinguished because operation imports must not be used as parameter or result type. For the definition of import lists, however, this distinction is irrelevant. Import lists only define a multi-valued aggregation relationship to an abstract import class. `Import`, in turn, has two subclasses, namely `TypeImport` and `OpImport`. `TypeImport`, therefore, multiply inherits from `TypeDecl` and `Import` as shown in Figure 6.3. Increments of class `TypeImport` can thus occur in import lists and also in the semantic relationship `DefinedIn/UsedBy`.

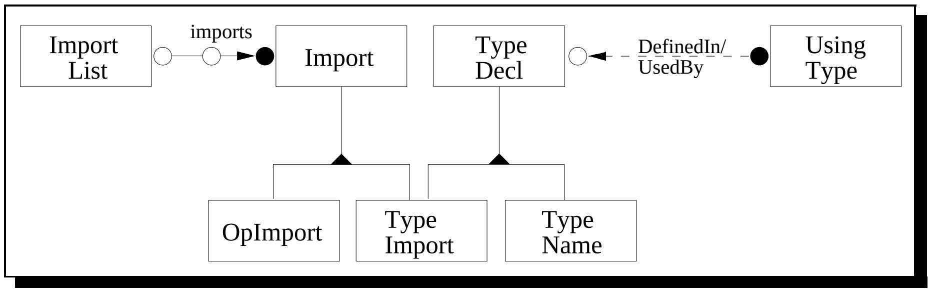


Figure 6.3: Multiple Inheritance for Polymorphism of different Properties

Multiple inheritance can be implemented with single inheritance. The result, however, will not be as concise and safe as with multiple inheritance. The different alternatives of implementing our example with single inheritance are displayed in Figure 6.4. In the first alternative, on the left-hand side of the figure, the problem is resolved by declaring `TypeDecl` as a subclass of `Import`. Then `TypeImport` transitively inherits the property of being an import via `TypeDecl`. In the second alternative, `Import` is defined as a subclass of `TypeDecl` and then `TypeImport` inherits the property of being a type declaration transitively via `Import`. The pitfall with these approaches is that classes inherit properties that they should not have. In the first alternative, exported type names could be inserted into import lists and in the second alternative imported operations could be used as types. The type system cannot exclude these specification errors, whereas they cannot occur in the solution with multiple inheritance. In the third alternative, on the right-hand side, we resolve the problem by not using polymorphism for the semantic relationship `DefinedIn/UsedBy` but duplicate it instead. The obvious consequence then is that later behavioural specifications that control the instantiation of these relationships become more complex. In addition, this approach does not work with ordered relationships, because we would not be able to retain the order of relationship elements if the relationship is split into two or more other relationships. In order to make the language safer and at the same time support concise specifications, we support multiple inheritance in entity relationship models as OMT does.

The introduction of multiple inheritance is not as simple as suggested above. Ambiguities might arise in a subclass due to name clashes in different super classes. An un-resolvable

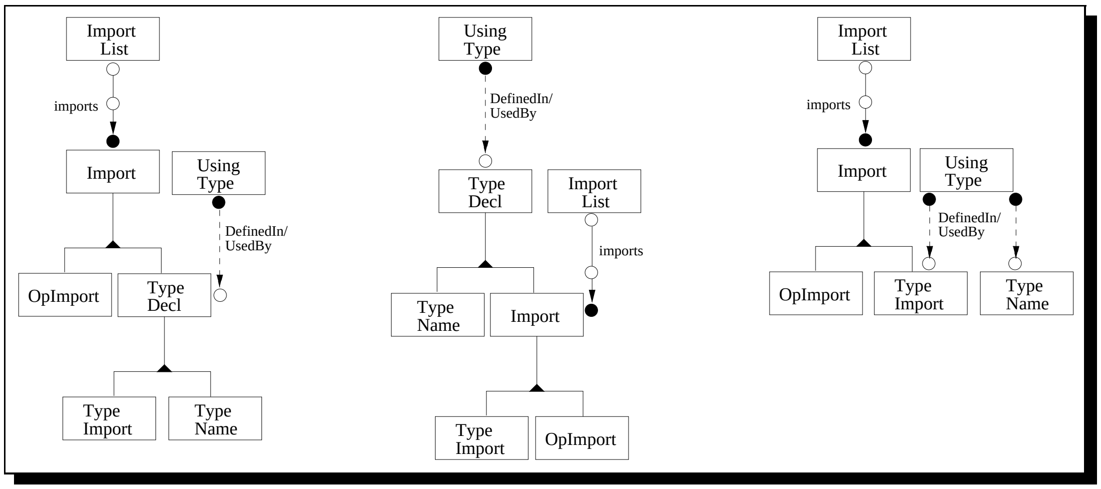


Figure 6.4: Implementation with Single Inheritance

ambiguity arises if a class `D` inherits from two (or more) classes `B` and `C` while both classes define a property `a`. If `a` is now applied to an instance of `D`, an ambiguity arises as to whether `a` from `B` or `C` is to be used. This situation is referred to as *incorrect multiple inheritance*. Rumbaugh et. al. consider this problem as an implementation detail: *"Conflicts among parallel definitions create ambiguities that must be resolved in implementations"* [RBP$^+$91]. We contradict to Rumbaugh and consider it as a design problem. As we shall see, resolving ambiguities might require changes to the class hierarchy which must be reflected in an entity relationship model. We, therefore, define static semantic constraints for the entity relationship model that highlight ambiguous declarations.

There are several options for resolving ambiguities. If the properties denote distinct concepts that both have to be visible in the subclass, the tool builder should change the property name in either `B` or `C` in order to reflect the difference. Another option to resolve the ambiguity is by *redefining* `a` in subclass `D`. This redefinition can then choose the property to take. This in fact hides the other property. If the two properties really denote the same concept that by chance is defined twice, the duplicate definition of the property should be removed. Therefore, a new class `A` should be introduced, which then defines `a`. The definition of `a` is removed from `B` as well as `C` and both classes are declared as subclasses of `A`. Class `D` then still inherits property `a` twice (via `B` and `C`). At this time, however, the ambiguity can be resolved automatically because `A` is known as the origin of the property definition. This situation is also known as *repeated inheritance*.

Another option for dealing with these multiple inheritance problems, would have been to provide explicit language constructs for renaming multiply defined entities in the subclass where the conflict appears. This approach is chosen in Eiffel [Mey92] and in the schema definition language of the O_2 ODBS [LR89]. O_2 even performs automatic renaming. Renaming properties in subclasses complicates both the understanding of a specification and the specification language definition. We do not consider the overhead to be justified here. Explicit renaming of subclass properties is only required when super classes cannot be changed. This is the case for reusable classes such as the ones that we suggested. Those class libraries will, however, not be very large. The library of predefined classes that we suggest only consists of twenty classes. Tool builder can, therefore, carefully design class libraries in order to avoid potential name clashes upfront.

Hierarchical decomposition: Entity relationship models that are used to model the structure of abstract syntax graphs in the way discussed above become reasonably complex. For each ENBNF production, the entity relationship model contains a class with as many outgoing aggregation relationships as there are symbols on the right-hand side of the respective production. The number of relationships may decrease through the introduction of abstract classes that pass on relationships to multiple subclasses. This, however, does not reduce the complexity significantly. The total number of classes will even increase. For languages whose syntax definitions require a significant number of productions, the entity relationship model will become too complex to be understandable. The syntax for C++ class definitions, for instance, contains some 80 productions and the corresponding model can hardly be understood. Structuring mechanisms are required that allow a tool builder to hide the fragments of an entity relationship model defining the structure of fairly independent subgraphs. These fragments should then be defined on a lower level of abstraction, within some refinement. Rumbaugh et. al. suggest *modules* as a structuring facility for their model. Modules, however, do not allow for *hierarchical decomposition* but rather coexist loosely. Therefore, they do not support different levels of abstraction, and understanding a set of modules is not significantly easier than understanding the overall model. In addition, consistency constraints that exist between different modules are not defined in OMT at all. We, therefore, introduce *subsystems*[4] as a means of hierarchical decomposition and perform the discussion about consistency constraints that Rumbaugh et. al. omit. The overall entity relationship model of a syntax graph will then be structured into a hierarchy of several *entity relationship diagrams*.

A subsystem is represented by a rectangle with an underlying shadow in an upper-level entity relationship diagram. The subsystem is then *refined* by another entity relationship diagram at a lower level. In the higher-level diagram, the subsystem may have aggregation, reference and inheritance relationships with classes and other subsystems. These relationships at the upper-level impose design obligations on the refining diagram. These obligations ensure that the diagrams at the different levels of abstraction are properly interconnected and that the semantics of the hierarchical diagram can be defined by an equivalent flat diagram. The basic concept for defining these obligations is the concept of a *port*. Ports have been successfully used to define hierarchical Petri Nets [Feh93]. What we define here is the adaption of the concept to entity relationship diagrams. Ports always represent classes from other diagrams and are, therefore, displayed in the same way as classes except that their colour is grey.

Each class that is connected to a subsystem at the upper level is included as a port in the diagram that refines the subsystem. The relationship that connects the class and the subsystem then has to be redirected in the refinement from the port to one or more classes or a nested subsystem that is declared in the refinement. For subsystems that are nested within a refinement and have ports connected to them, those ports have to appear again as ports in the refinement of the nested subsystem. The relationship then has to be redirected within the nested subsystem. Thus the way in which classes defined in nested subsystems are connected to classes defined in higher-level diagrams is well-defined.

As mentioned above, we also want to support relationships between subsystems. If we did not support them the classes that interconnect subsystems would have to be declared in the upper level diagram and could not be hidden within refinements. Relationships between subsystems, however, have to be refined in a different way from relationships between classes and subsystems. A relationship between two subsystems `A` and `B` shall denote that there is at least one

[4]We have chosen a different term, as in OMT, for the structuring concept in order to reflect the precise definition also in the notion.

class in the refinement of `A` that has this relationship with at least one class in the refinement of `B`. To ensure this, we require that the refinement of `A` includes at least one port that is defined as a class in the refinement of `B`. Vice versa, the refinement of `B` must include at least one port that represents a class of the refinement of `A`. The port in the refinement of `A` which represents a class from the refinement of `B` must then be connected to some class or subsystem declared in the refinement of `A` with the relationship between `A` and `B`. If it is connected to a subsystem, then the port must again be included as a port in the refinement of the subsystem. This is to be repeated until the relationship finally connects the port with a class. The same constraints must hold for the ports representing classes from `A` included in the refinement of `B`. In that way it is ensured that subsystems are never used as ports. Let us now consider an example.

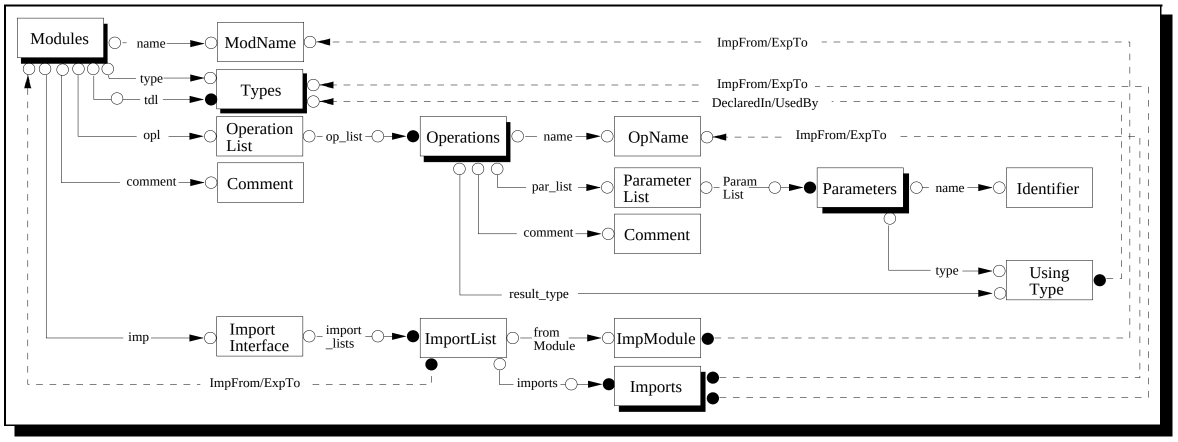


Figure 6.5: Entity/Relationship View of Groupie Interface Editor

The top level diagram of the entity relationship model of the Groupie interface editor is displayed in Figure 6.5. Subsystem `Modules` defines the classes that inherit from class `Module` and define the various Groupie module types. `Operations` contains the classes that define operations. `Imports` defines operation and type imports. `Parameters` contains the classes that refine the different kinds of parameters that are available in the Groupie interface language. Note that `Modules` has relationships with several classes and the subsystem `Types`. The refinement of `Modules` and these relationships are displayed in Figure 6.6.

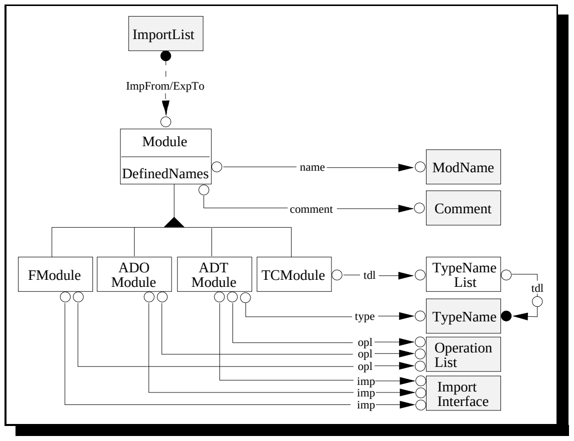


Figure 6.6: Refinement of Subsystem Modules given in Figure 6.5

The classes that were connected to subsystem `Modules` appear in the refinement as ports. In addition, port `TypeName`, which is defined in subsystem `Types` appears as a port in order to meet the constraints that we had defined for relationships between subsystems. Vice versa, classes `TypeNameList` and `ADTModule` will appear as ports in the refinement of subsystem `Types`. The diagram then defines how the various relationships that were defined between subsystem `Modules` and various classes, are refined by relationships between classes of the refinement and ports. The diagram defines, for instance, that any kind of module has a name and a comment as children. Therefore, relationships `comment` and `name` are inherited by all module classes. Moreover, any module inherits the property that it is the target of the semantic relationship `ImpFrom/ExpTo` from `Module`. A type collection module has no import interface, but only a further type definition list that is an ordered list of type names. All other module classes have an import interface and an operation list. `ADTModule` additionally has a type.

We note that the definition of subsystems and ports in our entity relationship model is considerably simpler than the original one for Petri Nets. One reason for that is that Petri Nets are bipartite graphs and the notion of refinement must respect this. A further reason is the formal definition of the dynamic semantics of Petri Nets. The semantic definition has to be extended due to the definition of subsystems. The fact that we do not need to define any dynamic semantics for our entity relationship model, since we only use them to define structural concerns of abstract syntax graphs, is a further reason why our notion of subsystem is so much simpler than the one in Petri Nets.

The semantics of a hierarchical entity relationship diagram is then defined in terms of a mapping to an equivalent flat diagram. We refer to this mapping as *flattening*. In order to flatten a hierarchical diagram, subsystems in the upper-level are replaced by their refinements. The relationships with the replaced subsystem are redirected, as determined by the relationships of the ports in the refining diagram. This substitution is repeated until all subsystems have disappeared.

Covariant redefinition of relationships: During the discussion of multiple inheritance, we identified the redefinition of properties as a means of dealing with name clashes. Moreover, property redefinition can be extremely useful for redefining properties in a subclass even though there is no name clash. Since OMT does not include relationship redefinition, we define it here. As a rationale consider an abstract class `IncrementList` that defines the common properties of arbitrary lists of increments. It will then define an ordered multi-valued aggregation relationship `il` with class `Increment`. The behavioural specification of the class will then define, for instance, the user commands that are required for arbitrary increment lists. In order to inherit these commands in some other increment class, say `OperationList`, we will have to declare the other class as a subclass of `IncrementList`. Then `OperationList` also inherits the aggregation relationship `il` whose target can be instances of arbitrary subclasses of `Increment`. To restrict the targets to being operations of some sort, we redefine the target class of `il` to become `Operation`. This restricts polymorphism in the required way.

In fact, for reasons of type safety we cannot allow arbitrary redefinitions but only specialisations. Redefinitions, therefore, have to be *covariant* [Car85], i.e. the redefinition of relationships is only allowed to subclasses of the original class. We refer to this condition as *covariant redefinition rule* in the following paragraphs. As an example of what will happen if we allow redefinitions of any kind, consider Figure 6.7.

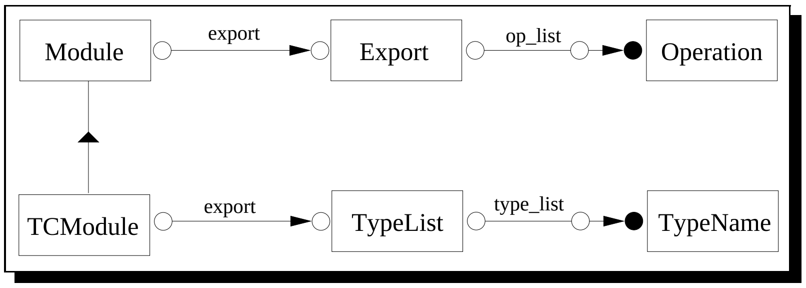


Figure 6.7: Unsafe Property Redefinition

The redefinition of aggregation relationship `export` in class `TCModule` is unsafe for the following reason. In a behavioural specification we could assign an instance of `TCModule` to a variable `e` whose static type is `Module` in order to instantiate a relationship, for instance. The traversal along relationships `export` and `op_list` would be valid from a static point of view. The last step of this traversal, however, is invalid at run-time because the traversal from `e` over `export` leads to a `TypeList` increment, which does not have an outgoing relationship `op_list`. With the covariant redefinition rule in place, we can detect this specification error.

A further problem can arise from the combination of relationship redefinition and multiple inheritance. Suppose there are two super classes `B` and `C` of some class `D`. Assume further that `B` and `C` have a common super class `A`. `D` repeatedly inherits the relationships defined in `A`. If one of these relationships is redefined in `B` or `C`, or even in both classes, then we again have an ambiguity. The target class of the relationship in `D` is ambiguous. We refer to this situation as *incorrect repeated inheritance*. It has to be resolved by additionally redefining the respective relationship in class `D`.

Unrequired Language Constructs: OMT defines a number of additional concepts beyond the language constructs discussed above. OMT provides, for instance, the means to define relationship attributes and the set of operations that are exported from a class. We do not use any of these additional constructs. Relationship attributes are not required, because relationships in our case connect classes that represent nodes in a syntax graph and are thus of very fine granularity. Attributes are, therefore, always attached to classes rather than relationships. We shall see later that this will significantly reduce the complexity of behavioural specifications since we need not consider relationships as first class objects. For relationships with attributes, we would have to define additional behavioural specification components for each relationship. We do not define operations in the entity relationship model, because it is to provide an overview of the syntax graph structure while operations already contribute to the behavioural specification. In addition, attaching operations to classes would again significantly increase the complexity of the diagrams. Operations will be defined in the behavioural definition of classes instead.

Summary: We have adapted OMT entity relationship diagrams so that they are particularly suitable for the definition of these abstract syntax graph structures. We have increased the modelling power of OMT diagrams by defining multiple inheritance, hierarchical decomposition and covariant redefinition precisely and have indicated why other OMT concepts are not required. In short, the entity relationship model now defines the following application specific static semantic constraints, which will enable a number of specification errors to be detected:

ER1: Class names are unique within all diagrams that belong to the entity relationship model.

ER2: Property names are unique among the set of properties defined for a class.

ER3: Aggregation relationships always start at a one-end, i.e. a class cannot be a component of more than one other class.

ER4: Multi-valued aggregation relationships must be ordered.

ER5: The inheritance hierarchy is acyclic.

ER6: Each subsystem is refined by one, and only one, diagram.

ER7: Each class that has a relationship with a subsystem is given as a port in the diagram that refines the subsystem.

ER8: For each subsystem that has a relationship with another subsystem, there are classes of the refinement that are depicted as ports in the refinement of the other subsystem.

ER9: For each relationship of a subsystem, the refinement contains at least one relationship of the same name, direction and cardinality between a port and a class or subsystem.

ER10: Name clashes between multiple super classes are resolved by redefinition or by explicit renaming in the super classes.

ER11: Redefinition of relationships is covariant.

ER12: Incorrect repeated inheritance is resolved by redefinition in the subclass that inherits repeatedly.

We are now in a position to use ENBNFs to define the abstract and concrete syntax of the language that must be supported by a tool. An ENBNF definition can then be translated into an abstract syntax tree definition in terms of an entity relationship diagram. Additional attributes and semantic relationships can be added to the entity relationship model, which then define the structure of the underlying abstract syntax graph. We do not yet have any specification primitives that could be used for defining the behaviour of the tool. From the requirements list discussed in Section 6.1, we cannot yet define unparsing schemes, static semantics, inter-document consistency and tool commands. Primitives for the specification of behavioural concerns of classes will be defined in the next two sections. For each class identified in the entity relationship model, we define its external and internal behaviour separately. The external behaviour definition includes properties that other classes can use for their internal definition. We, therefore, call the external definition *class interface*. The internal behaviour definition is then referred to as *class specification*.

6.4 Class Interfaces

Due to the heterogeneity of the different behavioural concerns, it will hardly be possible to find a unique formalism that will be appropriate for their specification. Instead, we will separate the different concerns and offer the most appropriate formalism for each of them. The class interfaces and specifications will, therefore, be structured into different sections that offer different paradigms to specify unparsing schemes, static semantics, inter-document consistency and tool commands appropriately. We integrate these different formalisms into a multi-paradigm language and define, again, the static semantics so that a number of specification errors between different sections can be detected.

In Section 6.1 we required from a specification language that it should enforce structuring of the overall specification into manageable components. The language should then support reuse of specification components. Object-oriented languages are known to support both reasonably. We, therefore, continue along these lines and consider the increment classes that were identified in the entity relationship model as the components of a tool specification. The behavioural definition of the overall tool will, therefore, be structured into the behaviour definitions of increment classes. In the same way as subclasses inherit structural definitions, such as attributes and relationships, they will inherit behavioural definitions from their super classes. As we will see, this allows reuse and greatly reduces the overall effort for specifying a tool.

Note that structuring a tool specification is simplified due to the fact that we do not have to consider relationship attributes in our application domain. If relationships had attributes that model particular relationship states, the behavioural definition would also have to determine transitions between these states. Then we would not only have to provide structuring facilities for increment classes, but also for the behavioural specification of relationships. Since relationships have no state apart from their existence, we can consider relationships as properties of the classes that have the relationship. Thus, the remaining concept for structuring a tool specification is the increment class.

The increment classes that are defined in the entity relationship model play different roles in the behavioural specification. If we ask a tool builder to make these roles explicit, we will be able to detect a number of specification errors. Abstract increment classes specify common properties of their subclasses. In a behavioural specification, however, a tool builder must not create any instances of an abstract class, because the class does not model nodes of the abstract syntax graphs. Classes that define leaves of the abstract syntax tree must not have commands to expand child increments, whereas classes for inner syntax tree nodes require these commands. Classes for inner nodes must specify the unparsing scheme that defines the textual representation of their instances. In order to be able to detect specification errors that would violate the above and further constraints, we distinguish different kinds of increment classes, namely *abstract classes*, *non-terminal classes* and *terminal classes*. We call instances of non-terminal classes *non-terminal increments* and instances of terminal classes *terminal increments*. We refer to them as *increments*, if their position in the syntax tree is not important. Besides increment classes, we additionally introduce *non-syntactic* classes that will be used for the declaration of non-atomic attribute types, such as error lists or symbol tables. Instances of these classes are referred to as *attributes*. If the distinction between attributes and increments is not important, we will denote instances of classes as *objects*.

The behavioural definitions in class interfaces and specifications must rely on the structural specification defined in the entity relationship model and refine it further. Semantic relationships and attributes that have been defined in the entity relationship model, for instance, will be used in the definition of static semantics and inter-document consistency constraints. Obviously, the consistency between structural definitions in the entity relationship model and behavioural definitions in the increment class definitions must be defined and checked. The question arises whether definitions from the entity relationship model should be included in class definitions or not. If we include them, the abstract syntax graph structure will be partly redundantly specified. If we do not include them, the structural concerns of classes are not fully determined. We might recover from that by extending the entity relationship notation to define additionally, for instance, attribute types, non-syntactic classes and the distinction between increment classes. Then, however, diagrams are overloaded with definitions that are not necessary to understand the syntax graph structure. In addition, the compilers for class interfaces and specifications then have to take a graphical diagram as input to check for consistency between declaration of syntax graph structures and their use in behavioural definitions. As an immediate consequence, we are no longer able to use standard techniques for the generation of an increment interface class compiler, but have to hand-code a compiler that understands graphical entity relationship diagrams. We, therefore, take the first alternative and include the definition of syntax graph structures into the increment class definition. As we will see later, this is not a real disadvantage, because the structural definitions in the classes are incrementally generated by the entity relationship editor and their consistency is checked by the tools contained in the GENESIS environment.

In the next two sections, we will restrict ourselves to presenting the main rationales for the definition of various concepts for the behavioural definition of increment classes. We are going to demonstrate their appropriateness with a number of examples. We will discuss alternative solutions and why the introduced concepts are sufficient. We will explain the semantics of the concepts informally. Here we are not going to define the concepts formally. The syntax and static semantics of the different concepts have been formally defined to provide a basis for detailed discussions about the concepts and as a correctness specification for the class interface and specification compilers. This formal definition is included in Appendix A.

6.4.1 Inheritance

The *inheritance section* determines the super classes of a class. It is included in the class interface definition because all the super classes contribute to the export of a class. The inheritance sections below are examples taken from the Groupie interface definition editor specification. The examples also display how the different kinds of classes are syntactically distinguished.

```
ABSTRACT INCREMENT INTERFACE Module        NONTERMINAL INCREMENT INTERFACE ADTModule
  INHERIT Document, ScopingBlock;            INHERIT Module, Commentable;
  ...                                        ...
END ABSTRACT INCREMENT INTERFACE Module.   END NONTERMINAL INCREMENT INTERFACE ADTModule.

TERMINAL INCREMENT INTERFACE Comment       INTERFACE SymbolTable
  INHERIT TerminalIncrement;                 INHERIT Attribute;
  ...                                        ...
END TERMINAL INCREMENT INTERFACE Comment.  END INTERFACE SymbolTable.
```

The classes identified in the inheritance section must be defined in another class interface. The inheritance section is mandatory, like in Smalltalk. Therefore, at least one of the predefined increment class specifications, whose interfaces are discussed and defined in Appendix B, must be defined in the inheritance section. The inheritance hierarchies of increment and non-syntactic classes must be disjoint, i.e. an increment class cannot inherit from a non-syntactic class and vice versa, otherwise tool builders could mix up the concepts of attributes and increments. The predefined library includes a root increment and a root non-syntactic class. Non-terminal and terminal classes must be leaves in the inheritance hierarchy. In that way, we will be able to define concepts for non-terminal and terminal classes, that cannot reasonably be inherited safely.

It has often been discussed that there are different kinds of use relationships in object-oriented languages [Mey88]. The inheritance relationship imposes a very strong use relationship between classes. When defining a class as a subclass of a super class, all definitions of the super class are used. The export of the super class even fully contributes to the export of the subclass. Therefore, the inheritance relationship must be used very carefully. In particular, it should never be used when there is an export of a super class that should not become an export of the inheriting class.

6.4.2 Import Interface

The import/export relationship (sometimes also called client/supplier relationship) imposes a much weaker dependency between the classes involved. It should be the preferred means of using other classes. This relationship can be considered as a contract between an exporter and an importer. Each class has a subset of properties that it *offers* as export. The contract becomes effective if another class, as an importer, *orders* some of the offers of the exporter. The two concerns of offer and order are in fact neglected by most object-oriented languages. They only consider the offer. In fact one cannot easily decide, in these languages, on which other classes a class depends. This is buried within the code of the class. It might be inferable by some tool but the designer is not forced to be aware of the dependencies. In trading, any importer would keep a list of the suppliers that exported goods to him or her. He or she would not want to go into his or her stores and look at the stamps of the goods to see where they come from. Being aware of dependencies is most important for him or her if he or she is to remain in business. For the same reason we support a tool builder in being aware of the dependencies of classes and introduce *import interfaces*. For the tool builder this awareness is important in order to be able to decrease the number of dependencies to achieve narrow interfaces. In the class interface, the import section denotes classes that are used in the export interface. Any other class importing from the class might have to import a subset of the imported classes in order to use the class' exports effectively. The import section of the class interface, therefore, plays a similar role to that played by the common parameter section in the Π-language [CFGGR91].

The import section in the interface of class `ADTModule` is depicted below. It imports classes `TypeName`, `Operation` and `ImportInterface` in order to declare its abstract syntax children. A class that uses `ADTModule` then has to import a class, such as `TypeName`, in order to traverse to the type name increment that represents the exported type of an ADT module.

```
NONTERMINAL INCREMENT INTERFACE ADTModule;
  ...
  IMPORT INTERFACE
    IMPORT TypeName;
    IMPORT Operation;
    IMPORT ImportInterface;
    ...
  END IMPORT INTERFACE;
  ...
```

Obviously, the classes identified in the import interface must exist, i.e. be defined in another class interface. Together with the inheritance section, the import interface then defines the set of classes that are *declared* in the class interface. These are the class itself, all classes that are in the transitive closure of the inheritance relationship and the classes that are declared in the import interface. The set of declared classes then induces the set of *declared types* that may be used in the interface. Classes are distinct from types because we have to support atomic types and we must be able to construct multi-valued types by applying type constructors to classes in order to represent multi-valued relationships. The atomic types that are known without any further declaration are `BOOLEAN`, `STRING`, `INTEGER`, `CHAR` and `ERROR_TYPE`. The latter type represents error descriptors that identify error messages for static semantic errors. Type constructors are applied to a *base type*, which can be any atomic type or any declared class. The available constructors are `LIST`, `SET`, `BAG` and `DICTIONARY`. Lists are ordered collections of elements. Sets are collections of unique elements. Bags may include duplicate elements and dictionaries are indexed collections that provide efficient associative access via a key of type `STRING`. Note that we do not support nested application of type constructors. These nested types are not required in abstract syntax graph structures because inner values of these types would not have an identity of their own and, therefore, would not correspond to syntax graph nodes. In addition, they could not be mapped to any concept that is available in the entity relationship model.

6.4.3 Abstract Syntax

Aggregation relationships of the entity relationship model are reflected in class interfaces within an *abstract syntax section*. The abstract syntax section is available for abstract and non-terminal increment classes. It is not defined for terminal increment classes, because these do not have children by definition. If a child is defined in an abstract class it is inherited by all subclasses. Children are specified in the abstract syntax section with the name that was attached to the corresponding aggregation relationship in the entity relationship model. If the aggregation relationship is multi-valued, the type of the abstract syntax child is constructed by the `LIST` type constructor. The type (or base type respectively) of the abstract syntax child must be the target class. Below there are several examples of abstract syntax sections taken from the Groupie interface definition tool.

```
ABSTRACT INCREMENT INTERFACE Module;          ABSTRACT INCREMENT INTERFACE Commentable;
  ...                                           ...
  ABSTRACT SYNTAX                               ABSTRACT SYNTAX
    name:ModName;                                 com:Comment;
  END ABSTRACT SYNTAX;                          END ABSTRACT SYNTAX;
  ...                                           ...

NONTERMINAL INCREMENT INTERFACE ADTModule;    NONTERMINAL INCREMENT INTERFACE OperationList;
  INHERIT Module, Commentable;
  ...                                           ...
  ABSTRACT SYNTAX                               ABSTRACT SYNTAX
    typ:TypeName;                                 il:LIST OF Operation;
    op_list:OperationList;                      END ABSTRACT SYNTAX;
    imp:ImportInterface;                        ...
  END ABSTRACT SYNTAX;
  ...
```

An ADT module exports a type that is an instance of class `TypeName`. Moreover, it contains a list of operations that is an increment of class `OperationList`. `OperationList`, in turn, has an increment whose type is constructed using the type constructor `LIST` applied to base type `Operation`. The class `Operation` is an abstract increment class. Due to polymorphism, instances of subclasses such as `Procedure` or `Function` may be included in the operation list. Finally, an ADT module has an import interface, which is an instance of the nonterminal increment class `ImportInterface`. The overall abstract syntax components of class `ADTModule` are determined by computing the union of the abstract syntax components defined in the class itself with those defined in the super classes. Therefore, the abstract syntax components of class `ADTModule` also include a component of class `Comment`, since `ADTModule` inherits this component from its super class `Commentable` and a component of class `ModName` inherited from class `Module`.

The distinction between different kinds of classes enables us to exclude a number of potential specification errors. It does not make sense, for instance, to have a terminal increment class that inherits from an abstract class, which, in turn, defines abstract syntax children. In that case the terminal class would inherit these children and no longer be terminal. The static semantic definitions for abstract syntax sections, therefore, exclude these situations. The type of all abstract syntax children must have been declared in the class. If the type is multi-valued, it must have been constructed with the `LIST` type constructor for the same reason as multi-valued aggregation relationships in the entity relationship model must be ordered. In addition, it is unreasonable to have attributes or atomic types as syntax children. The static semantic definition, therefore, requires that types or base types of abstract syntax children have to be increment classes. Without the domain specific distinction between different kinds of classes, such as if we used an object-oriented programming language, we would not have been able to exclude these unreasonable abstract syntax definitions.

6.4.4 Unparsing Schemes

As a first concept for the behavioural definition of a tool we now introduce *unparsing schemes*. They define mappings for input and output between abstract syntax graphs and the external representation. Unparsing schemes are defined for non-terminal increment classes only. They cannot be defined for abstract increment classes. In that case abstract syntax children that might be added in subclasses would not be reflected. Neither are unparsing schemes required for terminal increment classes. For terminal increments the layout computation only needs to output the terminal increment's *lexical value*. This is the character string that was matched with the regular expression defining the lexical syntax of the terminal increment class, and this need not be specified any further.

The unparsing scheme consists of a list of unparsing items delimited by commas. An unparsing item can either be a *keyword*, a *white space*, a *child increment reference* or a *formatting item*. Keywords are given in quotes and are required during parsing and output during unparsing. White spaces are defined as `WS`. During parsing any character string that consists of at least one blank, tabulator or new line matches a white space. During unparsing, a single blank is inserted as white space. References to child increments are given by the name of the child as defined in the abstract syntax section. Such a reference indicates that a child increment has to be parsed and unparsed at the specified position. For multi-valued child increments, an optional delimiter specifier may define unparsing items that must be input and output between elements of the list. Formatting items are defined in round brackets. They are distinguished from keywords since they do not contribute to the syntax definition and, therefore, can be neglected during parsing. They are typically used for defining indentations. `(NL)` is a special formatting item that causes a line break in the output representation.

How the output representation is actually computed is defined by the notion of nested *boxes*. Each increment has a box. The increment representation is inserted into the increment's box. This means that all keywords, white spaces and pretty-printing items are output into the increment's box. In particular, the scope of a new-line is restricted to the increment's box.

This means that an implicit indentation is made according to the left margin of the increment's box. For each child increment a nested box is inserted and the child increment's representation is inserted into that box. The upper left-corner of an increment's box is determined by the current insertion point where the child increment is referenced. As an example, consider the unparsing schemes of `ADTModule` and `OperationList` below, taken from the Groupie interface tool specification.

```
NONTERMINAL INCREMENT INTERFACE ADTModule;     NONTERMINAL INCREMENT INTERFACE OperationList
 ...                                            ...
 UNPARSING SCHEME                               UNPARSING SCHEME
  "DATATYPE",WS,"MODULE",WS,name,";",(NL),(NL),  il DELIMITED BY (NL),(NL) END
  ("   "),com,(NL),(NL),                        END UNPARSING SCHEME;
  ("   "),"EXPORT",WS,"INTERFACE",(NL),(NL),
  ("      "),"TYPE",WS,typ,";",(NL),(NL),
  ("      "),op_list,(NL),
  ("   "),imp,(NL),(NL),
  "END",WS,"MODULE",WS,name,".",(NL)
 END UNPARSING SCHEME;
 ...
```

The unparsing scheme of `ADTModule` defines that an abstract data type module definition is introduced by the keywords `DATATYPE` and `MODULE` that are delimited by a white space. Then a nested box is inserted for the representation of the module name. The position of that box is, in fact, indented 16 blanks from the left margin of the outer-most box. A semicolon is printed immediately after the module name. Then a blank line is inserted before the module's comment. The comment is pretty-printed in a nested box that is indented three blanks from the module's left margin. After that, a blank line is inserted before the export interface is introduced by the keywords `EXPORT` and `INTERFACE`. Again a blank line is inserted and then the exported type is printed with an indentation of six blanks. Beneath the type, the box for the module's operation list is inserted. According to the unparsing scheme of `OperationList` on the right-hand side, operations are delimited by a blank line in the output representation. Note that the left margin of the operation list box is six characters from the left margin of the outer-most box. Then the box for the import interface is inserted with an indentation of three blanks from the left margin. Again a blank line is inserted. The tail of the interface definition consists of the keywords `END` and `MODULE`, the module name and a full stop.

We define again a number of static semantic correctness conditions for unparsing schemes to be able to draw the tool builder's attention to specification errors. Any reference included in the item list must denote the name of an abstract syntax child that is a direct or inherited abstract syntax child. Vice versa all inherited abstract syntax children must occur at least once in the unparsing scheme, otherwise they would be useless. References to child increments that are followed by a delimiter declaration must denote multi-valued child increments.

We have defined a declarative means to specify how tools compute the external representation of documents. It is defined on a type level of abstraction in the unparsing schemes of non-terminal increment classes. Furthermore, we have defined a number of static semantic constraints that restrict the use of unparsing schemes in such a way that many specification errors are excluded. We have not and will not define a means to define different unparsing strategies for the same document type. This would be a straight-forward extension, we would only have to name the unparsing schemes and afterwards let the tool user decide on which particular strategy to use. We have not included multiple unparsing schemes, since the lack of this feature does not affect the overall proof that tools can be effectively constructed on top of object database systems.

6.4.5 Lexical Syntax

The lexical syntax has been defined by the regular expression productions in the ENBNF. It is again included in the interfaces of terminal increment classes in order relieve the class compilers from also considering the ENBNF definition. Consistency between the lexical syntax definition in the terminal increment class interfaces and the corresponding ENBNF will be controlled and maintained by the ENBNF and class interface editors of the GENESIS environment.

The lexical syntax of terminal classes is specified as a regular expression in the mandatory *regular expression section* of a terminal increment class interface. As terminal increment classes in GTSL cannot have subclasses, the lexical syntax definition cannot be inherited. As we aim at specialisation of all inherited properties, inheritance of lexical syntax definition would, in fact, not be very reasonable. In that case we would have to define a concept for the specialisation of regular expressions, which is hardly ever possible. The syntax for these regular expression sections facilitates definition of extended regular expressions. They are identical with those used in `lex` [JPAR68]. As examples consider the following regular expression definitions for identifiers and comments of the Groupie language.

```
TERMINAL INCREMENT INTERFACE Identifier;      TERMINAL INCREMENT INTERFACE Comment;
  ...                                           ...
  REGULAR EXPRESSION                            REGULAR EXPRESSION
    [A-Za-z][A-Za-z0-9$_]*                        /"*"([^*/]|[^*]"/"|"*"[^/])*"*"/
  END REGULAR EXPRESSION;                       END REGULAR EXPRESSION;
  ...                                           ...
```

According to the fairly simple regular expression on the left-hand side, identifiers consist of at least one letter. Then an arbitrary long sequence of letters, numbers and the special symbols `$` and `_` may follow. The definition for comments is more complicated. It uses alternatives that are separated by `|`. The operator `^` matches any character except the character that immediately follows. The regular expression on the right-hand side indicates that a comment starts with `/*` and ends with `*/`. Between these, a sequence of arbitrary characters may be included, provided that this sequence does not contain a subsequence that would end the comment.

The lexical values of terminal increments are important for static semantics and inter-document consistency checks. The lexical values, such as a module name or a type import, are the units of terminal increments that carry semantics. Therefore, we store the character string that is matched with a regular expression of a terminal increment in an implicit attribute `value`. This attribute can then be used during static semantics and inter document consistency definitions to retrieve the lexical value of terminal increments.

6.4.6 Attributes

Globally visible attributes are defined in the entity relationship model. Beyond these, we may have to define attributes that are only locally visible, i.e. that are not exported and, therefore, cannot be accessed from other increment classes. In addition, the entity relationship model does not yet declare attribute types. This is because attribute types are not important in the early stage of a tool development. To prevent unnecessary complexity, our notion of entity relationship diagrams does not include types. When it comes to accessing attributes in behavioural definitions, however, the consistency of the access must be checked in order to exclude specification errors. We need the notion of types for that purpose. The class interface definition, therefore, includes an *attribute section* in order to refine attribute definitions from the entity relationship model.

An attribute definition declares a *name* and a *type* of an attribute. All types that have been declared by inheritance or import within the class definition interface may be used here. In that way, we provide a rich variety of language constructs for defining the structure of attributes. Non-syntactic classes can also be used to impose a particular behaviour on attribute types. We do not address non-syntactic classes any further here. Refer to the appendix for a detailed discussion. In general non-syntactic classes provide the expressive power of an object-oriented language including multiple inheritance, construction of types and encapsulation with methods. As an example, consider the following example from the Groupie interface editor definition.

```
ABSTRACT INCREMENT INTERFACE Module;
  ...
  ATTRIBUTES
    DefinedNames:SymbolTable;
  END ATTRIBUTES;
  ...
```

This attribute section defines an attribute `DefinedNames`. Its type is the non-syntactic class `SymbolTable`. A `SymbolTable` is an association between character strings and increments. As soon as a new identifier is declared within a module, a static semantic constraint definition will establish an association between the value of the identifier and the increment that declares the identifier. This association is then further used during semantic analysis and inter-document consistency checks.

Attributes are inherited from super classes. Attribute types can be redefined by including a declaration, with the inherited attribute name into the attribute section of the subclass. Redefinition of attributes is required for the same reason as for relationships. In the above example, the symbol table is inherited by the different module types. In type collection modules, however, only type names are included in the symbol table. We might want to exploit this knowledge and redefine `DefinedNames` to become a `TypeTable`. The redefinition also has to be covariant to exclude type errors. In the example this means that `TypeTable` has to be a subclass of `SymbolTable` and then all operations that are available for symbol tables are available on the type table, too.

For the notion of covariant redefinition of attributes, however, we need to define the notion of *subtypes*. Therefore, we need to distinguish single- and multi-valued types. In the single-valued case, atomic types do not have any subtypes. If the type is a class, all subclasses are subtypes of the type. In the multi-valued case, those types are subtypes of a given type t that are constructed with the same type constructor applied to a subtype of the base type of t. Then redefinitions are safe for the same reason as for relationships. Note that the type system is considerably simplified compared to the type system of O_2C due to the fact that we were able to omit nested types. In particular, the notion of subtypes in O_2C is much more complicated because several levels of nesting have to be considered rather than one level in our case.

A tool builder can prevent other importing classes from accessing an attribute by restricting its visibility and declaring it as *hidden*. This contributes to the software engineering principle of information hiding. To do so the attribute declaration is preceeded by the keyword `HIDDEN`. Then the attribute cannot be imported from other classes. Note that hidden attributes are also declared in subclasses. We do not want to support selective inheritance, because this is inherently unsafe. Assume, for instance, that class `Increment` declares an attribute `HIDDEN Errors:SET OF ERROR_TYPE` as well as operations to add or delete errors from that set. Some subclass, say `ADTModule`, inherits the operations. Without inheriting the attribute `Errors`, the inherited operations would have no attribute to operate on and would have to raise a run-time error.

The redefinition of attributes must not change the attribute's visibility. Due to polymorphism there is no way to decide statically whether such a change in visibility is safe. Consider again the above example. Suppose now that `Errors` was exported by `Increment`. If the visibility was changed to `HIDDEN` in subclass `ADTModule` and an object of `ADTModule` was assigned to a property `e` whose static type is `Increment`, the attribute access `e.Errors` would be statically valid because `Increment` exports `Errors`. At run-time, however, the access would be invalid because the increment `e` refers to an object of class `ADTModule`, which hides the attribute. As we have not found any situation where this unsafe redefinition would be required, we exclude it.

The definition of attributes may also cause name clashes due to multiple inheritance. Also incorrect repeated inheritance might occur if the attribute type is redefined in one super class but not in the others. These situations are defined in a similar way to relationships and not considered here further.

6.4.7 Semantic Relationships

We refine semantic relationships within increment class definitions rather than defining a concept orthogonal to increment classes. Any increment that has an incoming or outgoing reference relationship in the entity relationship model, will declare a *semantic relationship section*. Semantic relationships are specified by pairs of unidirectional *links* in the semantic relationship sections of the two increment classes that participate in the relationship[5]. The *explicit link* denotes the direction from the source to the target increment class. The *implicit link* denotes the reverse direction. The main rationale for including links in the class interface definition is to define static semantic rules that ensure the safe use of relationships during the specification of static semantics and inter-document consistency constraints.

Relationships are created and deleted during static semantics and inter-document consistency checks. Creation of a relationship is specified on a type level of abstraction by assigning an expression that denotes an increment to an explicit link. Such an assignment first deletes the existing relationship, if any. Then the increment denoted by the expression is assigned to the explicit link. Finally the implicit link is established by including the source increment in the set that stores the implicit link. A relationship is deleted by assigning the undefined value `NIL` to the explicit link of the relationship. Thus creation and deletion of relationships is only controlled by the target increment class of a relationship or its subclasses. This further contributes to the enforcement of structured specifications. As an example, consider the refinement of relationship `ImpFrom/ExpTo` between classes `TypeImport` and `TypeName` that are taken from the Groupie interface editor specification.

[5]The terminology follows the concepts for relationships that have been introduced in the PCTE data model [GMT87].

```
TERMINAL INCREMENT INTERFACE TypeImport;  TERMINAL INCREMENT INTERFACE TypeName;
  INHERIT TypeDecl;                         INHERIT TypeDecl;
  SEMANTIC RELATIONSHIPS                    SEMANTIC RELATIONSHIPS
    ImpFrom:TypeName;                         IMPLICIT ExpTo: SET OF TypeImport.ImpFrom;
  END SEMANTIC RELATIONSHIPS;               END SEMANTIC RELATIONSHIPS;
  ...                                       ...
```

Increment class `TypeImport` defines an explicit link `ImpFrom` that leads to an increment of class `TypeName`. Using that link, a `TypeImport` increment can refer to the `TypeName` increment that it imports. If the relationship does not exist, `ImpFrom` has the undefined value of `NIL`. In the behavioural specification of `TypeImport` we then assume that the corresponding `TypeName` does not exist. The relationship is established by assigning an instance of class `TypeName` to the link. After that, `ImpFrom` may be used to navigate to the corresponding `TypeName` increment. The implicit link `ExpTo` in class `TypeName` contains an implicit reference to the `TypeImport` increment as soon as the assignment of an increment to `ImpFrom` has been made. That link is exploited for the definition of change propagations or a browsing command that visits all increments that use a particular type.

We exclude again a number of potential specification errors with additional static semantic constraints. The type of a link must be an increment class. It must not be a non-syntactic class or an atomic type because relationships between increments and attributes or atomic values would make no sense. The type of an implicit link must match with the declaration of an explicit link. This means that the class identified in the type must declare the explicit link that is defined after the dot in the implicit link type. Otherwise the implicit link would not correspond to an explicit link.

Note that it is unreasonable to define hidden links. The main rationale for defining relationships and thus for defining links is to use them for navigation purposes. The application of links for navigation purposes is obviously not confined to the class where the link is defined. On the contrary, links are most often used in the behavioural specification of other classes. Hidden links could never serve this purpose. We, therefore, do not define means for declaring links as hidden.

6.4.8 Methods

We will require different levels of abstraction during the behavioural specification of those concerns that we have not yet addressed, namely tool commands, static semantics and inter-document consistency constraints. At the lowest level of abstraction, there will be accesses and modifications of attributes, abstract syntax children and semantic relationship links. At a higher level of abstraction several accesses and modifications will be structured into operations. At an even higher level of abstraction, these operations will be used for the definition of tool commands, static semantics and inter-document consistency constraints. At yet a higher level of abstraction, tool commands and static semantic constraints that have already been defined are reused in subclasses. These different levels of abstraction are required to provide a tool builder with the flexibility that is needed to specify arbitrary tools. We shall, however, help the tool builder to avoid using the lower levels of abstraction whenever possible.

To enforce well structured tool specifications, we define the following visibility rules for properties. A property is visible in some other class if it is not hidden and has been properly imported (c.f. Page 147). It may then be used for navigation purposes in *path expressions*. It will not be permitted for classes that have imported a property to change the property's value. Only those classes can modify a property that have either declared it or inherited it from a super class. If some other class needs to modify a property, it must use an operation instead. The visibility rules for properties are, therefore, the same as those defined for instance variables in Smalltalk [Gol84]. Operations are defined in terms of *methods* of a class. The methods defined for an increment class, therefore, provide other increment classes with the only interface for changing properties and also for performing recursive navigations.

The methods that are available for a class are declared in the *method section* of the class interface. This declaration will include the method name, its parameter list and the method's result type, if any. Methods may be declared as *hidden* by preceeding the declaration with the keyword `HIDDEN`. Hidden methods cannot be used by client classes. They are only used in the behavioural specifications of the class and its subclasses. Similar to attributes, the visibility of methods cannot be changed in subclasses to avoid unsafe situations. As an example consider the methods that are defined for class `Module` in the Groupie interface editor specification.

```
ABSTRACT INCREMENT INTERFACE Module;
  ...
  METHODS
    METHOD expand_name(Str:STRING):BOOLEAN;
    METHOD change_name(Str:STRING):BOOLEAN;
  END METHODS;
  ...
```

The first method is used to expand the module name increment and check the string that is passed as parameter for conformance to the regular expression that defines the lexical syntax of module names. The second method changes a module name that has already been expanded. It might perform additional change propagation to import lists in order to change the imported module name there as well.

As we have seen in Section 5.9.2.2 (c.f. Page 108), there are a number of methods, such as `parse` or `unparse`, that should be available in arbitrary increment classes. These methods cannot be inherited because their bodies depend on the increment class and, therefore, really differ. To relieve the tool builder from having to define method bodies for the operations identified above, we introduce the concept of *implicit methods*. These are those methods that we identified and their declarations in the method section are preceeded with the keyword `IMPLICIT`. Their bodies are not specified by the tool builder, but generated from the declarations that have been provided in other sections. In that way a tool builder can use these sections at a lower level of abstraction, e.g. for the definition of particular tool commands. The examples of classes `ADTModule` and `ModName` below display how implicit methods are defined for non-terminal and terminal increment classes.

```
NONTERMINAL INCREMENT INTERFACE ADTModule;     TERMINAL INCREMENT INTERFACE ModName;
...                                             ...
 METHODS                                        METHODS
  IMPLICIT METHOD init(f:Increment);             IMPLICIT METHOD init(f:Increment);
  IMPLICIT METHOD expand;                        IMPLICIT METHOD collapse;
  IMPLICIT METHOD collapse;                      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
  IMPLICIT METHOD parse(Str:STRING):ADTModule;   IMPLICIT METHOD unparse:STRING;
  IMPLICIT METHOD unparse:STRING;               END METHODS;
 END METHODS;                                   ...
 ...
```

The dynamic semantics of these implicit methods can best be explained by using state transition diagrams. Consider, therefore, Figure 6.8. It displays two state transition diagrams, one for non-terminal and the other for terminal increments. If increments exist they can be in one of the states `expanded` or `not expanded`. Initially, they do not exist. To create an increment, method `init` is executed, which transfers it into state `not expanded`. In that state it can be unparsed and the result will be the class name included in pointed brackets, which represents a place holder. By method `expand` a non-terminal increment is transferred into the state `expanded` where all its child increments are created but not yet expanded. In addition, it changes into this state when the method `parse` is applied and the string is syntactically correct. In that case, the abstract syntax tree representing the increment is fully expanded. For terminal increment classes, the equivalent operation is `scan`. It performs the transition only if the string conforms to the increment's regular expression. In state `expanded`, the unparse method can be applied as well. For a non-terminal increment it will compute a character string as determined by the unparsing scheme. For terminal increments it will return the result of the lexical value. Methods `parse` and `scan` can also be applied to non-terminal resp. terminal increments that have already been expanded. In case the string passed as argument is correct, they remain in the state, otherwise the increment will transit back into the state of a place holder. Method `collapse` deletes abstract syntax children of a non-terminal increment or the lexical value of a terminal increment and, after its execution, the increment will no longer be expanded.

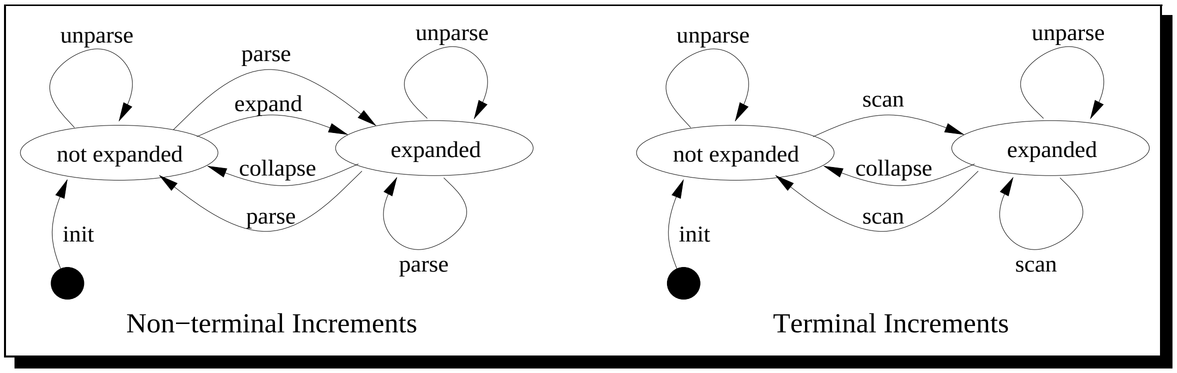


Figure 6.8: Dynamic Semantics of Implicit Methods

Methods will be invoked, in the Smalltalk style, by passing a message to an object. Due to polymorphism of properties, we will have to support inheritance of methods. If we do not support this, we end up with unsafeness that cannot be detected statically. Consider as an example the exported methods of `Module`. Assume we declare some property `e` of type `Module` that at run-time refers to an instance of class `ADTModule`. If `ADTModule` does not inherit the methods from `Module`, method invocations like `e.expand_name` will be statically correct, but have to reveal a run-time error, because `expand_name` is not available for the `ADTModule` object. As an additional advantage of method inheritance, the code of method bodies can be reused. This contributes to our overall goal of simplifying tool definitions.

Given that methods are inherited, tool builders should also be able to *redefine* inherited methods in a subclass. As a consequence, binding of methods to messages can no longer be static, but has to be dynamic. With method redefinition, tool builders will then be able, for example, to define a change propagation in a subclass, when this propagation is unreasonable in the general case and, therefore, cannot be defined in the super class. Whether or not change propagation is or is not performed then depends on the class of the object that receives the message. A tool builder may also want to use more specific types as parameter and result types in order to use properties and methods that have been added in these more specific types. Therefore, method redefinition can be covariant, which enables changes to parameter and result types to occur in the same way as suggested by Cardelli [Car85]. A tool builder may redefine a method with parameter and result types that are subtypes of the respective types in the original method definition. Together with polymorphism, covariant method redefinition may cause type errors in rare cases[6]. As discussed in [CLZ94], type safeness for this problem is statically undecidable. The paper suggests a technique called *type data flow analysis* that will highlight all type errors. Type data flow analysis is a pessimistic approximation and might detect situations that at run-time do not cause errors. The technique has been used for languages such as Eiffel and O_2C, which support covariant definition in the same way as we have suggested. It can, therefore, easily be adapted to ensure type safety of covariant method redefinitions in our increment class definitions.

A tool builder might want to define the behavioural definition of commands that are common to a number of classes in a common super class of these classes and then let the subclasses inherit this definition. If the particular modifications that have to be carried out during these commands differ, the tool builder might exploit late binding in order to define the modifications in methods that are specific to these classes. In order to use these methods safely in a super class, however, the super class must somehow define obligations that enforce the definition of the respective methods in subclasses. We use the concept of *deferred methods*[7] to express these obligations. A method that is declared as deferred in a super class has to be redefined in subclasses. For all non-terminal and terminal subclasses we require deferred methods of super classes to be redefined in the class itself or in a super class that is a subclass of the class with the deferred method.

Name clashes can occur with methods defined in multiple super classes of a class. In conjunction with method redefinition, incorrect repeated inheritance can occur for the same reason as it can for relationships. We define these situations in the same way as we do for relationships. The tool builder will have to resolve them similarly by renaming methods in the super classes, or redefining methods. The precise definition is included in the appendix.

6.4.9 Summary

In this section we have defined the language concepts that are available for the external behavioural specification of increment classes. These are the unparsing scheme and the method section. We have suggested inheritance of methods and implicit methods to raise the level of abstraction that a tool builder has to use for defining a tool. The other sections have been included in class interface definitions because they specify the behaviour of implicit methods and, therefore, need to be known in classes that use the class. We now continue with concepts for class specifications.

[6]During the evaluation of the language in the GOODSTEP project, we have not found a single type error that would trace back to a covariant method redefinition.

[7]Deferred methods fulfil the same purpose as the Eiffel concept of the same name or as pure virtual member functions in C++.

6.5 Class Specifications

6.5.1 Import Interface

The private part of the behavioural class definition is used to define static semantics, inter-document consistency constraints and tool commands for increments of the class. These definitions will be based on declarations that are exported from other classes. With regard to the explicit declaration of imports, our concern not to bury dependencies between classes inside the implementation, is now reinforced. We, therefore, introduce an *import interface section* for class specifications as well. Unlike class interfaces, entries in this interface not only declare the import of classes, but also the particular properties and methods exported by the supplier that are used during the class specification. As an example, consider the import relationship of the specification of class `ADTModule`.

```
INCREMENT SPECIFICATION ADTModule;
 IMPORT INTERFACE
  IMPORT TypeName        INCLUDING scan, value;
  IMPORT OperationList   INCLUDING expand, add_element, insert_element, delete_element;
  IMPORT ImportInterface INCLUDING import_lists, unremove;
  IMPORT SymbolTable     INCLUDING associate,deassociate;
  IMPORT ImportList      INCLUDING imports;
  IMPORT Operation       INCLUDING name;
 END IMPORT INTERFACE;
```

Class `ADTModule` can now use methods defined for abstract syntax children as well as nested abstract syntax children for navigation purposes. These navigations will be defined in terms of path expressions.

6.5.2 Path Expressions

The purpose of a *path expression*[8] is to statically, i.e. on a type level of abstraction, define a navigation from an increment of a particular class along syntactic and semantic relationships to remote increments. Before we actually define path expressions in detail, we discuss a number of properties that tool builders will require from primitives for defining navigation paths.

Expressiveness: Path expressions must serve as a concise primitive for searching, from a given increment, for other semantically related increments. We, therefore, require traversal primitives such as navigating to abstract syntax children, the abstract syntax father or following links of semantic relationships. Furthermore, our language supports the concept of structured attributes for storing semantic information and, in particular, symbol tables. The contents of these attributes will also have to be accessed for searching semantically related increments. Path expressions must, therefore, be defined in such a way that they can include attribute accesses and, in particular, symbol table lookups. Then a number of operators for path expressions in graph grammars [ELS87, Sch91a], such as iterations, unions or intersections, become obsolete. Increments can then be located by symbol table lookups rather than by graph traversals that visit a huge amount of nodes.

[8]The path expressions defined in this thesis should not be confused with those defined in [CH74]. That paper suggests path expressions as a concept for synchronisation of concurrent processes.

Safeness: The notion for path expressions should exclude definitions of invalid navigation paths. We consider a path to be invalid if it can never exist in a given structure definition of the underlying syntax graph.

Efficiency: Path expressions should define navigation paths in such a way that it can be efficiently decided whether the path exists. If the path exists, the increment or the set of increments that are addressed by the path should be efficiently computable from the path expression.

A path expression in our language consists of a sequence of *steps* delimited by a dot. A step, in turn, can be a navigation to an abstract syntax child, to the abstract syntax father or along a link of a semantic relationship. A step can also be a method call. This supports structuring path expressions, since subpaths can then be defined in methods. As methods can call themselves recursively, path expressions can be recursive as well.

We do not support intermediate steps in a path expression that are multi-valued abstract syntax children or links. These multi-valued steps are not required because below we will introduce the dedicated operators `FOREACH` and `EXISTS` that can be used to split multi-valued path expressions. As we will see, this restriction to single-valued intermediate steps will increase performance during evaluation of path expressions and contributes to meeting the safeness requirement, which we consider now.

Path expressions are statically defined in the context of some increment classes c. The first step in the path expression has to denote a property of the class or `SELF`. `SELF` denotes the object for which a path expression is currently being executed. Note that this might be an instance of a subclass of c due to polymorphism. Each step s of a path expression is associated with a static type. For the first step, this is the type of the property identified by the step or c if the first step is `SELF`. Due to the single-value requirement, the type of s is a class if s has a successor step s'. For the correctness of s', we then require the name of the step to be either an abstract syntax child, the abstract syntax father, a semantic relationship link or a method of the class denoted by the previous step s. In that way, we ensure that the path expression can always exist. We are able to highlight specification errors if path expressions do not conform to the underlying abstract syntax graph structure.

Due to polymorphism, steps may lead to increments of more specific types than defined by the static type of the respective property or method used in the step. Sometimes a tool builder has to exploit knowledge about dynamic types and specialise the static type in the path expression. We include a *cast operator* for that purpose. This is inherently unsafe. Therefore, [Mey92] suggests "*No such thing [as a type cast] exists in Eiffel. This is essential if we want to have any trust in our software*". Although it is unsafe, we require type casts. As an example consider the general purpose class `Increment` that will be contained in a library of reusable increment classes (c.f. Appendix B). `Increment` implements a path expression to the root increment of a document in terms of a method `get_doc`. The result type of this method is the abstract increment class `Document`, which is again contained in the library of reusable increment classes and models the common properties of root increments of an abstract syntax tree. Each tool specification will, therefore, define a root increment class as a subclass of `Document`. Any execution of `get_doc` will return instances of these subclasses, since abstract increment classes cannot be instantiated. In order to access properties or invoke methods from these results, however, the static type of `get_doc` needs to be specialised. This can be done with covariant redefinition or assignments against the polymorphism rule. For covariant redefinition, an additional subclass of `Increment`, which covariantly redefines the `get_doc` method, would have to be defined. This is reasonable if the method is applied on enough occasions to justify the introduction of a new class. Therefore, GTSL includes covariant redefinitions. Another option would be to redefine `get_doc` in all classes. Then, however, we do not reuse methods but, on the contrary, have to multiply them in the specification. Meyer also admits the need for type specialisations and has included the `?=` operator in the most recent Eiffel language definition. The polymorphism rule is abandoned for this operator. The assignment is performed if the dynamic types are compatible, otherwise the operator assigns `NIL`. This approach has two main disadvantages. First, it is as unsafe as type casts due to the unsafeness of invoking a feature on `NIL`. Moreover, requiring assignments for performing a specialisation would run contrary to expressive path expressions. A tool builder would then have to declare additional local variables and split path expressions in order to apply the `?=` operator. We, therefore, include type casts in GTSL. Unlike casts in C or C++, however, we severely restrict their applicability. We require that the static type can only be specialised, whereas in C++ arbitrary casts, even between atomic types, are allowed. In addition, the language includes a number of operators, such as `IS_OF_CLASS` or `IS_KIND_OF`, that implement inquiries on the dynamic type of a property or method result. These operators can then be used for safeguarding the use of type casts. Moreover, the type data flow analysis technique discussed during covariant method redefinition can be applied to ensure the safe use of type casts [CLZ94]. As examples, consider the following path expressions that are taken from the Groupie interface definition.

```
INCREMENT SPECIFICATION ImportList;
...
  fromModule.ImpFrom.father
...

INCREMENT SPECIFICATION UsingType;
...
  (<Module>SELF.my_op().father.father).DefinedNames
...
```

Note how understanding these path expressions is simplified by considering the entity relationship diagrams in Figures 6.5 and 6.6 on Page 130. The first path expression is defined in the context of class `ImportList`. It traverses to the `ImpModule` and then follows an explicit semantic relationship link to a `ModName` to obtain the document by traversing along the abstract syntax father. The reference might then be assigned to the explicit link `ImpFrom` in class `ImportList` in order to establish the semantic relationship. The second path expression is defined in the context of class `UsingType` in order to obtain the increment that declares the type. Therefore, it traverses to the operation increment that includes the using type by method `my_op`, and traverses twice along syntactic fathers in order to obtain the root increment. It then specialises the static type with a cast to become `Module`. Only then is it valid to access the symbol table attribute `DefinedNames` defined in class `Module`. The attribute may then be used further to perform a symbol table lookup to the increment that is associated with the lexical value of the using type increment.

6.5.3 Method Bodies

Bodies of implicit methods are generated by the compiler from specifications given in other sections and deferred methods are refined in subclasses. A tool builder, therefore, only has to implement the bodies of explicit methods. They are defined in the *method section* of class specifications. Path expressions may be used in expressions contained in method bodies for addressing sets of remote increments. The class specification language thus includes primitives for statements as they are known in most object-oriented languages. The static semantics defines the obvious constraints and we do not consider them here. Refer to the detailed definition in the appendix. As an example, consider the explict method body `change_name` exported by class `Module`. The purpose of the method is to propagate the change of a module name to all places where the name is imported so as to avoid the introduction of inter-document consistency constraint violations.

```
INCREMENT SPECIFICATION Module;
 ...
 METHODS
  ...
  METHOD change_name(Str:STRING):BOOLEAN;
  BEGIN
   FOREACH i:ImpModule IN name.ExpTo DO
    i.react_on_change(Str);
   ENDDO;
   RETURN(name.scan(Str))
  END change_name;
 END METHODS;
```

The example displays the use of the predefined operator `FOREACH` with a multi-valued path expression. It declares a cursor `i` that iterates over the set of those import lists that are defined by the multi-valued path expression `name.ExpTo`. During each iteration it informs another import list about the ongoing change. The method `react_on_change`, in turn, performs a change of the identifier in the import list as well. After completion of this change propagation, the change is performed to the module name by applying the implicit method `scan` to the abstract syntax child `name`. The result of scan is returned as result of `change_name`. Methods like `change_name` will then be used in command definitions to implement user commands.

These explicit methods provide tool builders with the flexibility to express arbitrary computations. This is necessary to be able to build process-sensitive tools. In particular, they allow the tool builder to combine implicit methods, deferred methods or methods that are reused from predefined classes, with tool-specific traversals through the abstract syntax graph. The example above is taken from the Groupie module interface editor specification. As already discussed on Page 7 the Groupie approach to consistency management is process dependent. To customise the tool for a different process where change propagations are inappropriate, we simply omit this and other `FOREACH` statements and generate the tool anew.

Admittedly, the level of abstraction provided by explicit methods is still rather low. We, therefore, do not consider these methods the ultimate solution for tool specification. In Chapter 9, we shall see that they are required for the industrial applications within the GOODSTEP project. There we use them for the specification of tool-specific services and gain initial experiences with these services. Starting from these experiences, it might be possible to identify more abstract concepts. Our approach is, therefore, similar to *design patterns* [GHJV93] that are now emerging for object-oriented programming languages. A design pattern describes a number of methods or a set of related classes that solve a particular common problem. In [GHJV93] a number of patterns, like *object factories* or *wrappers* have been deduced from experiences with object-oriented programming gained in several large-scale projects.

6.5.4 Semantic Rules

Attributes and semantic relationships are concepts that can be used for defining data structures for static semantics and inter-document consistency constraints. Changes of attribute values and the creation or deletion of semantic relationships will be defined in tool command definitions by invoking methods. These changes, however, usually require a number of follow-on activities in order to check static semantic constraints for related increments.

As an example, consider the creation of a new type import increment in the Groupie module interface tool specification. A command will invoke a method which creates a new type import increment, scans a given string and inserts the type import increment into an import list increment. Then it must be checked to see whether the type import matches with an exported type in the respective imported module. In that case the semantic relationship `ImpFrom/ExpTo` must be established between the two increments, otherwise the type import must be considered wrong and an error attribute of the type import increment should include a respective error descriptor. By accessing this error attribute, the tool command can decide if the type import is correct or not. If not, it might then define whether to tolerate or to reject the error. If the type import is correct or the tool command tolerates the error, the import now extends the set of declared types and the symbol table attribute `DefinedNames` of the module must be updated. This, in turn, might require checking using type increments such as parameter types or function result types. They could have become correct by the change if their lexical value is identical to that of the type import. Then their error attributes will have to be updated in order to reflect the correction of this error.

If we did not define any other concept than methods, tool builders would have to find valid execution orders to perform the required follow-on actions for all potential attribute and semantic relationship changes. We strongly consider this to be at the wrong level of abstraction. Tool builders require instead a declarative concept for defining the correctness of the various static semantic and inter-document consistency constraints. This concept should, in particular, relieve them from worrying about the order in which evaluations are performed. The new concept should also support our structuring paradigm and be defined in terms of increment classes. In addition, the concept must enable the efficient evaluation of static semantic constraints to be carried out as this has to be done on-line, i.e. during the execution of user commands. It must, therefore, meet the time restrictions required in Section 2.3.3. We now define *semantic rules* that will meet the above requirements.

Each semantic rule consists of a list of statements called *action* that is bound to a *condition*. The condition is specified after the `ON` clause and the action is defined between `ACTION` and `END ACTION` keywords. Temporal predicates may be used to specify conditions, namely `CHANGED` and `DELETED`. A `CHANGED` predicate becomes `TRUE` if its argument has been created or changed since the last execution of the semantic rule. The `DELETED` expression becomes `TRUE` if its argument is about to be removed. Arguments of a `CHANGED` or `DELETED` expression may be attributes or semantic relationships of any other increments. Path expressions are used to determine attributes or semantic relationships of remote increments. A name of an attribute may only occur as the last name in a path expression. Compound conditions can be built by using the `OR` operator. An `EXISTS` operator is used in the usual sense of first order logic to specify that the rule has to be executed as soon as some other condition holds for an element in a multi-valued syntax component or a multi-valued semantic relationship. This predicate, in fact, resolves the situation for semantic rules that intermediate steps in path expressions cannot be multi-valued.

```
INCREMENT SPECIFICATION TypeImport;
...
SEMANTIC RULES
  // Rule 1
  ON CHANGED(father.ImpFrom.DefinedNames) OR CHANGED(value)
     // symbol table of imported module or lexical value of type import changed
  VAR i: Increment;
  ACTION
    i := father.ImpFrom.DefinedNames.increment_at(value);
    IF (i == NIL)                           // Name undefined?
      THEN ImpFrom:= NIL;                  // Type Import not o.k.
      ELSE
      IF (i.IS_OF_CLASS("TypeName"))      // Is Defined Name a TypeName?
        THEN ImpFrom := <TypeName>i;     // o.k: establish semantic relationship
        ELSE ImpFrom := NIL;            // Type Import not o.k.
      ENDIF;
    ENDIF;
  END ACTION;
```

Figure 6.9: Semantic Rule in Class `TypeImport`

As a first example, consider a semantic rule from class `TypeImport` of the Groupie interface editor specification that is displayed in Figure 6.9. The condition of this rule defines that the rule will be executed if either the value of attribute `DefinedNames` in the symbol table of the imported module or the lexical value of the type import itself has been changed.

If the condition of a rule becomes true, the list of GTSL statements given in the action is sequentially executed before attributes and links that are modified by the rule are accessed the next time. The available statements for semantic rule actions are assignments of expressions to attribute values, invocation of methods reading or modifying attribute values, creation of relationships and control flow primitives such as `IF` or `FOREACH` statements. In order to facilitate encapsulation and avoid side effects, semantic rules of a class may only modify attributes or relationships defined or inherited by that class. Note that actions must, therefore, not invoke methods via path expressions that would modify attributes of remote increments. This is excluded by a static semantic constraint

The purpose of the action in the first semantic rule in the above example is to determine the existence of semantic relationship `ImpFrom`. The first statement reads the `DefinedNames` attribute in order to obtain the increment that is associated with the lexical value of the `TypeImport`. If there is no such association, then the semantic relationship should not exist and `NIL` is assigned to `ImpFrom`, otherwise the rule checks whether the dynamic type of the increment really is a type name in order to prevent the import of some other resource, such as an operation that might be exported under the same name. If the association denotes a type name, we can specialise the static type using the cast operator and establish the semantic relationship `ImpFrom/ExpTo` between the import and the corresponding export, otherwise the relationship should not exist, since the increment that is defined in the symbol table is not a type name. A very similar rule is defined for class `OpImport` in Figure 6.10. It establishes the `ImpFrom/ExpTo` relationship between operation imports and the respective exported operation.

The two classes `TypeImport` and `OpImport` share the common characteristic that the increments are considered erroneous whenever the semantic relationship between import and export does not exist. This common behaviour is defined in the common super class `Import` as displayed in

```
INCREMENT SPECIFICATION OpImport;
...
SEMANTIC RULES
  // Rule 1
  ON CHANGED(father.ImpFrom.DefinedNames) OR CHANGED(value)
     // symbol table of imported module or lexical value of op import changed
  VAR i: Increment;
  ACTION
    i := father.ImpFrom.DefinedNames.increment_at(value);
    IF (i == NIL)                            // Name undefined?
      THEN ImpFrom:= NIL;                   // Operation Import not o.k.
      ELSE
      IF (i.IS_OF_CLASS("OpName"))        // Is Defined Name an OpName?
        THEN ImpFrom := <OpName>i;       // o.k: establish semantic relationship
        ELSE ImpFrom := NIL;            // Type Import not o.k.
      ENDIF;
    ENDIF;
  END ACTION;
```

Figure 6.10: Semantic Rule in Class `OpImport`

```
INCREMENT SPECIFICATION Import;
...

SEMANTIC RULES
  // Rule 1
  ON CHANGED(ImpFrom)                          // If link has changed
  ACTION
    IF (ImpFrom == NIL)                        // See whether it (still) exists
      THEN Errors.append_error(#NotAValidExport);// add respective error
      ELSE Errors.clear_error(#NotAValidExport); // delete error
    ENDIF;
  END ACTION;
END SEMANTIC RULES;
```

Figure 6.11: Semantic Rule in Abstract Class `Import`

Figure 6.11. The rule is, therefore, inherited by both subclasses. The value of `Errors` depends on the existence of semantic relationship `ImpFrom/ExpTo`. The condition of the rule in class `Import`, therefore, includes a `CHANGED` expression with the link `ImpFrom` as the argument. To be able to define it this way, we have to define the link `ImpFrom` in class `Import`. It is then covariantly redefined in subclasses to lead to `OpName` or `TypeName`, respectively. The rule's precondition becomes `TRUE`, if for example the first semantic rule has modified the link. In this case, a check is made whether the link exists. If it does not exist then the import is wrong and an error message is added to the `Errors` set, otherwise the import is correct. The corresponding error message is removed because the import might have been incorrect before.

Note that we explicitly define the dependencies between rules and attributes. As with import interfaces, these dependencies might be inferred by some static analysis tool. It is, however, important that the tool builder is aware of the dependencies. As with imports these dependencies should be minimised. The rationale here is to facilitate efficient rule evaluation. Moreover, the explicit declaration of dependencies, as with imports, simplifies the impact analysis if attribute or relationship declarations are changed.

As a further example consider the rules below that modify and access the `DefinedNames` symbol table in an `ADTModule` whenever elements of the export are changed. These changes must be reflected in the symbol table in order to keep imports consistent. The first rule enters a type name into the symbol table whenever the value of the exported type is changed. The second rule uses the `EXISTS` predicate in order to update the symbol table `DefinedNames`, whenever the lexical value of an operation name changes. Then the method `associate` of the predefined non-syntactic class `SymbolTable` is invoked in order to store the operation as an entry with its new name as the key. If necessary the old key is deleted.

```
INCREMENT SPECIFICATION ADTModule
  ...
  // Rule 1
  SEMANTIC RULES
  ON CHANGED (type.value)
  ACTION
    DefinedNames.associate(type.name, type.name.value);
  END;

  // Rule 2
  ON EXISTS op in opl.op_list:CHANGED(op.name.value)
  ACTION
    DefinedNames.associate(op.name, op.name.value);
  END ACTION;
  ...
```

Figure 6.12: Semantic Rule in Class `ADTModule`

The next rule is defined in class `OpName` and uses the symbol table `DefinedNames` of its root increment in order to check for the uniqueness of operation names. If the symbol table contains another increment such as a type name under the same key, then an error message is added to the `Errors` set, otherwise the message is deleted.

```
INCREMENT SPECIFICATION OpName;
  ...
  // Rule OpName::1
  ON CHANGED father.father.father.DefinedNames
  ACTION
    IF father.father.father.DefinedNames.is_duplicate(value) THEN
      Errors.append_error(#NameAlreadyDefined)
    ELSE
      Errors.remove_error(#NameAlreadyDefined)
    ENDIF
  END ACTION;
  ...
```

Figure 6.13: Semantic Rule in Class `OpName`

The dynamic semantics of semantic rules is informally defined as follows. Rule evaluation is, in fact, divided into two phases. The first phase is a propagation phase where attributes and semantic relationships are stamped as *dirty* whenever they are modified. All semantic rules that read dirty attributes or relationships are also stamped as dirty. Then the property of being dirty is transitively propagated to all attributes or relationships that are modified by dirty rules. The propagation phase ends when all affected rules, attributes and relationships have been marked dirty. The second phase evaluates rules and it starts whenever a dirty attribute or relationship is about to be accessed (e.g. during unparsing). Then all rules are executed according to the propagation path defined in the propagation phase. This execution brings the attribute or relationship back into state *clean*. Only then is the attribute or relationship really accessed.

The conditions of semantic rules specify static dependencies between different semantic rules. We do not want to enable computations or even modifications to be defined in these conditions. Such modifications could produce serious side effects and lead to situations where the rules can no longer be understood by the tool builder. In particular, the tool builder had to know the order in which change predicates are evaluated and it was the main motivation for semantic rules to relieve the tool builder from this burden. Path expressions as defined above include steps that may invoke methods and in these methods side effects could be produced. Method invocations would be required to determine recursive path expressions as they occur, for instance, during specification of nested scoping blocks. Upon increment creation, however, recursive path expressions can always be materialised within auxilliary semantic relationships. Then these relationships can be used in semantic rule conditions. We can, therefore, disable method invocations to occur in steps of path expressions of semantic rule conditions.

In short, we have defined a declarative language for the definition of static semantics and inter-document consistency constraints. Note that we have not defined any execution order between the different semantic rules that we have discussed. A valid execution order will be inferred statically from the dependencies between rules, attributes and relationships. Semantic rules enforce well-structured static semantics and inter-document consistency constraint definitions because the language does not enable rule actions to modify attributes or relationships that do not belong to the class. Moreover, subclasses inherit semantic rules from super classes.

6.5.5 Interactions

From the list of requirements identified in Section 6.1, we have not yet addressed the specification of tool commands. A tool builder will have to define the command names that appear in context-sensitive menus that the user can pop up. This includes the definition of preconditions that must be fulfilled for the appearance of the command in the menu. Moreover, the particular dialogues between tool and user, if any, must be defined. The tool builder might want to define different dialogue styles in different tools. In particular, the process model might require a particular way of handling static semantic errors or inter-document consistency constraint violations. The command definition language must be flexible enough to support these different concerns. Moreover, a number of command definitions are very similar among multiple increment classes. A tool builder will appreciate it if commands can be defined once and can then be reused in different classes. To meet these requirements, we add a concept of *interactions* to the class specification language that supports the definition of commands as required.

The definition of an interaction encompasses an internal and an external name, a selection context, a precondition and an action. The external name appears in context sensitive menus. The internal name is used to determine the redefinition of an inherited interaction. The selection context defines which increment must be selected so that the interaction is considered for inclusion in a menu. It is actually included if the precondition that follows the `ON` clause evaluates to `TRUE`. The action is a list of GTSL statements that is executed as soon as the user chooses the interaction from the menu.

We define two further predefined classes that can only be used in interaction bodies. These classes are `TEXT` and `TEXT_SET`. They offer primitives for user dialogue specifications. `TEXT` offers user interaction primitives to cope with strings. It has methods for displaying a string in a message window, editing a single text-line or editing a text that consists of arbitrary lines. `TEXT_SET` offers primitives to handle input and output of sets of strings. Among these are methods for selection windows in which users can select single or multiple entries. As an example, consider in Figure 6.14 the interaction that changes a module's name.

```
INCREMENT SPECIFICATION Module;
  ...
  INTERACTIONS
    INTERACTION ChangeName
    NAME "Change Module Name"
    SELECTED IS name
    ON (name.expanded)
    VAR t:TEXT;
        err:TEXT_SET;
    BEGIN
      t:=NEW TEXT(name.unparse);
      IF (t.LINE_EDIT("Enter New Module Name!")) THEN
         IF (NOT name.change_name(t.CONTENTS())) THEN
            err:=NEW TEXT_SET(name.get_errors());
            err.DISPLAY();
            ABORT
         ENDIF
      ENDIF
    END ChangeName;
```

Figure 6.14: Command Definition to change a Module Name

This interaction is considered to be offered if the current increment is the module's name. It is actually offered if the name identifier has already been expanded. If this is the case and the user has requested a menu, the string `Change Module Name` will become a menu item. If the user chooses this item, the action is executed and the user will be prompted to edit the name of the module in a line edit window. The default character string in this line edit window is the old module name that is computed by sending message `unparse` to abstract syntax child `name`. If the dialogue is completed, the `LINE_EDIT` method returns `TRUE` and the increment modification operation `change_name` (c.f. Page 150) is executed. The operation returns `TRUE` if the identifier is lexically correct, otherwise it returns `FALSE` and an error message is displayed. Afterwards the command is aborted, i.e. the state of the document is restored to its state before the command execution started.

Subclasses inherit interactions from their super classes. The interaction `ChangeName` defined above is, in fact, inherited by the four different subclasses of `Module`. Interactions might be redefined. As an example of this, consider type collection modules in the Groupie interface specification. They define very low-level modules, which are often imported from a very great number of other modules. For these modules, the above interaction might not be appropriate. Suppose that the command for type collection modules should warn the user if the change impacts more than a fixed amount of other modules, say 10. In that case, it would also be inappropriate to tolerate static semantic errors as was the case in the other interaction. Due to the flexibility of the class specification language we can then redefine the interaction in class `TCModule` to take these considerations into account. The redefined interaction is depicted in Figure 6.15.

```
INCREMENT SPECIFICATION TCModule;
  ...
  INTERACTIONS
    INTERACTION ChangeName
    NAME "Change Module Name"
    SELECTED IS name
    ON (name.expanded)
    VAR t,warning:TEXT;
        err:TEXT_SET;
        cont:BOOLEAN;
    BEGIN
      IF name.ExpTo.SIZE>10 THEN
         warning:=NEW TEXT("Module is imported by more than ten other modules!");
         cont:=warning.ASK_TO_CONTINUE("Do you want to continue?");
      ENDIF;
      IF cont THEN
        t:=NEW TEXT(name.unparse);
        IF (t.LINE_EDIT("Enter New Module Name!")) THEN
          name.change_name(t.CONTENTS());
          IF NOT name.Errors.IS_EMPTY()) THEN
            err:=NEW TEXT_SET(name.get_errors());
            err.DISPLAY();
            ABORT
          ENDIF
        ENDIF
      ENDIF
    END ChangeName;
```

Figure 6.15: Redefined Command Definition to change a Module Name

The redefined interaction accesses the semantic relationship `ImpFrom/ExpTo` between module names and their counterpart in import lists. If the number of entries in the implicit link is bigger than the upper bound, a warning will be displayed on the user's screen. If the user ignores the warning, the change operation is performed. Note that this time we access the `Errors` attribute of the name regardless of the result returned by `change_name`. At this state, `Errors` will be dirty for the following reason: method `change_name` modifies the lexical value attribute that, in turn, outdates the set of defined modules. Since the error set of the module name will depend on that attribute it will become dirty as well. The access of the error set attribute that checks whether the set is empty will, therefore, cause semantic rules to fire re-evaluating the static semantic constraints and bringing `Errors` back into clean state. If the set includes any error descriptor, they will be formatted and then the command will be aborted and thus all performed changes will be undone.

Due to multiple inheritance and redefinition, ambiguous situations may occur. These are simpler than for methods because interactions do not have parameters or types and are, therefore, not considered here any further. The formal definition in the appendix includes constraints that exclude name clashes and incorrect repeated inheritance of interactions.

Interactions have transaction properties. They are atomic i.e. they are either performed completely or not at all. They are the unit of concurrency control, i.e. an interaction may be delayed or even aborted in case of concurrency control conflicts with other concurrent interactions. Hence interactions are performed in isolation. Once completed, the effect of an interaction is durable, i.e. all changes that were made during the interaction persist even if the tool is stopped accidentally by a hardware or software failure.

6.5.6 Summary

We have defined an object-oriented language for the behavioural definition of increment classes. Compared to object-oriented programming languages (such as our implementation languages C++ and O_2C), our language considerably raises the level of abstraction at which tool builders specify tools. We have defined unparsing schemes as a declarative concept for defining the external representation of documents. We have included primitives for defining the abstract and lexical syntax of increments. Beyond this higher-level of abstraction, the language provides the flexibility for a tool builder to use a lower-level of abstraction, if required. Implicit methods of increment classes provide an interface to these definitions so that they can be used in methods at a lower-level of abstraction. We have defined semantic rules as a declarative formalism to specify static semantics and inter-document consistency. Compared to programming languages, this particularly relieves the tool builder from worrying about execution orders. Finally, we have added a concept for defining tool commands including their appearance in context-sensitive menus and primitives for user dialogue specification.

The language supports the structuring of specifications particularly well. Components are increment classes that are identified in the ENBNF and entity relationship model. Their inheritance and import/export relationships are explicit. Reuse of definitions is supported because all definitions are inherited by subclasses and can be redefined there if required. The language distinguishes different kinds of classes and different kinds of properties in order to enable domain specific static semantic constraints to be defined. A significant number of potential specification errors are excluded by these constraints. In an object-oriented programming language, even if it is as strongly typed as O_2C, these specification errors cannot be highlighted due to the lack of domain knowledge.

6.6 Related Work

This section reviews the most powerful existing tool generators with respect to the tool specification languages they use and compares the approaches with GTSL. Subsection 6.6.1 presents ordered attribute grammars as they are used by the *Cornell Synthesizer Generator* [RT84]. Section 6.6.2 discusses the *Centaur* System [BCD+88] with respect to its specification languages. The language *PROGRESS* [Sch91a] developed in the *IPSEN* project is dedicated to the specification of general graph structures. As such it has been applied to the specification of abstract syntax graphs as well. PROGRESS is then discussed in Subsection 6.6.3.

6.6.1 Attribute Grammars

Attribute grammars have been suggested in [Knu68] for the specification of the semantics of context-free languages. While they fail to express dynamic semantics appropriately, they have been successfully used for the definition of static semantics of languages. Ordered attribute grammars [Kas80] are an important subclass. For this subclass it is efficiently decidable at compile-time whether the static semantic evaluations terminate. They have been proved expressive enough to define the syntax and static semantics of most programming languages [KHZ82]. Ordered attribute grammars are the basis for the specification language SSL of the Cornell Synthesizer Generator [RT84, RT88, RT89]. SSL enables hybrid syntax-directed tools to be specified.

Abstract Syntax Specification in SSL: The abstract syntax of a language is defined in SSL in terms of productions. The left-hand side of a production is a phylum. The right-hand-side consists of a set of operators delimited by `|`. Phyla represent non-terminal symbols of a grammar and operators represent productions. The arguments of operators are again phyla and represent the right-hand side of a production. As an example, we model the abstract syntax of the Groupie language with SSL. The excerpt given below corresponds to the GTSL fragment on Page 137 in order to simplify the comparison.

```
root           module;
module :         ModulePlaceholder()
               | ADTModule(identifier comment type operation_list import_part identifier)
               | ADOModule(...)
               | FModule(...)
               | TCModule(...);
optional       comment;
comment :        CommentNull()
               | CommentPlaceholder()
               | Comment(COMMENT);
list           operation_list;
operation_list : OpList    ()
               | OpListPair(operation operation_list);
...
```

In the first line, the phylum `module` is declared to be the root phylum of the specification. The next production defines that a module phylum can correspond to the empty word, i.e. still be a place holder or have child phyla according to the four productions that model the four different Groupie module types. Here there are notable differences to GTSL. Firstly, SSL does not define names for child phyla. As we will see this will complicate understanding successive SSL specification fragments. Secondly, common properties of phyla cannot be designated as such. All operators that produce the four module types will have the first two and the last argument in common. In GTSL this can be defined in an abstract increment class from which other classes can inherit. Finally, place holders have to be defined explicitly with an operator. In GTSL these are implicitly defined by the attribute `expanded` that is inherited from `Increment`. The keyword `optional` in the following line defines that comments are optional phyla. This declaration is redundant, since an additional operator producing the empty word is also defined. In GTSL increments are defined as optional by inheriting from the predefined class `OptionalIncrement`. The SSL specification then defines a further operator whose argument is a regular expression definition called `Comment`. This corresponds to the regular expression section of terminal GTSL increments. The next statement declares the phylum `operation_list` as a list phylum. The two operators in the next production then produce operations as abstract syntax children of `operation_list`. In GTSL, this can be more easily defined by inheritance from the predefined class `NonterminalIncrementList`.

Unparsing Schemes in SSL: To define the textual representation of the abstract syntax, SSL offers language constructs for defining unparsing schemes. Besides defining the textual representation, these schemes also determine the increments that are selectable and whether or not they can be freely edited.

```
module : ModulePlaceholder[ ^ :]
       | ADTModule[ @ ::= "DATATYPE MODULE " @ ";" error "%n%n%t"
                              @ "EXPORT INTERFACE %n%n%t"
                                 @ "%n%n%t"
                                    @ "%n%b%b"
                              @ %n%b
                          "END MODULE" @ ".\%n\%n\%n"]
       | ADOModule[...]
       | FModule[...]
       | TCModule[...]    ;
```

Unparsing declarations for each applicable operator of a phylum must be defined in an unparsing scheme. The example above depicts the unparsing scheme for phylum `module`. Each definition for the different operators is given in square brackets after the operator specification. They consist of a left- and right-hand side, which are separated by `:` or `::=`. If the separator is `:`, the increment is not freely editable and it can be edited when declared with `::=`. The property of being freely edited is defined in GTSL in an interaction. This interaction can be defined in a common super class where all subclasses inherit it from. Phyla in unparsing schemes are represented by either `@` or `^`. `@` defines that the phylum can be selected and a phylum represented as `^` cannot be selected. The textual representation of a phylum is then defined by the right-hand side of the unparsing definition. Therefore, strings including optional formatting items (characters with a leading `%`) can be interleaved with the enumeration of child phyla. Here we find the SSL representation harder to understand and to define than the GTSL counterpart, since the child increments are not explicitly named. This may easily lead to specification errors.

Static Semantics and Inter-Document Consistency in SSL: SSL offers a means of defining *static semantics* of languages in terms of *attributes* and *equations*. Attributes are attached to phyla. Equations are attached to operators. Assignment of values to attributes is defined in equations. Each attribute must be declared either as *synthesised* or *inherited*. A synthesised attribute is computed based on attributes of child phyla, whereas an inherited attribute is computed based on attributes of an ancestor phylum. An equation can also invoke a tool-specific function. To define these, SSL offers a C-like notation.

As an example, consider a scoping rule for the Groupie module interface language in Figure 6.16. The rule defines that a module's name must be unique within the module. The specification of this scoping rule in terms of SSL attributes, equations and functions is depicted below. It corresponds to the semantic rules of the Groupie specification on Page 154.

The first three lines define the phylum `name_list` as a list of identifiers. This phylum is then used as an attribute type for the declaration of attribute `elist`. `Elist` is attached as a synthesised attribute to the phyla `module`, `operation_list` and `operation`. Its values are lists of those identifiers that are defined in the scope of a module. The next declaration defines another synthesised attribute for a module phylum which is meant to store an error message. The operator `ModulePlaceholder` initialises the `name_list` attribute of a module to be an empty list. The operator `ADTModule`, in turn, defines the `elist` attribute of an ADT module as concatenation of the module's identifier, the module type identifier and the `elist` attribute of `operation_list`. Similar equations, which are for reasons of brevity not depicted here, have to be defined for all other phyla operators where identifiers are declared. A non-empty error message is assigned to a module's `error` attribute, if the module identifier occurs more than once in the module's `elist` attribute. This computation relies on the SSL

```
list name_list;
name_list       : NameListNil()
                | NameListPair(identifier name_list);
module,
operation_list,
operation       { synthesized name_list elist;};

module          { synthesized STR error;};

module:         : ModulePlaceholder {module.name_list = NameListNil();}
                | ADTModule {
                  module.elist = NameListPair(type,
                                 NameListPair(identifier, operation_list.elist));
                  module.error = MultiplyDefinedErr(identifier,module.elist);
                };

INT Occurrences (identifier i, name_list l) {
/* compute number of occurrences of i in l */
        (i==IdentifierPlaceholder) ? 0 :
        with (l) (NameListPair(mod,nl):
                   (mod == i) ? 1 : 0,
                    default:0
                  ) + Occurrences(i,nl)
}

STR MultiplyDefinedErr(identifier i, name_list l) {
/* returns error message, if i occurs more than once in l */
        (Occurrences(i,l)>1) ? "identifier multiply defined" : "")
}
```

Figure 6.16: Specification of Static Semantics in SSL

function `MultiplyDefinedErr` which, in turn, uses `Occurrences`. The `error` attribute is used in the module's unparsing scheme and specifies how the error message is displayed within the document representation.

One of the strengths of attribute grammars for specifying static semantics is that a tool builder need not bother about execution sequences. This is the same in GTSL. An order for the efficient and incremental evaluation of equations is automatically derived by the Synthesizer Generator from dependencies between equations. In the example above, a module's `error` attribute, for instance, depends on the module's `elist` attribute, which, in turn, depends on the values of module name and type identifier and the `elist` attribute of `operation_list`. Therefore, in a tool generated by the Synthesizer Generator, the first equation would always be evaluated before the second equation. It is shown in [Kas80] that it is possible for ordered attribute grammars to find an evaluation sequence that at most performs one assignment per attribute. In addition, assignments can be ordered in such a way that time-consuming context switches that visit other nodes are minimised. To accelerate the re-evaluation of attributes even further, the Synthesizer Generator applies an incremental attribute evaluation technique which takes into account that only those attributes changed after the last evaluation and the transitive closure of attributes depending on the changed attributes, have to be re-evaluated. As will be shown in Section 8.2.1, GTSL semantic rules can also be evaluated incrementally.

A serious weakness of attribute grammars, however, is that they are based on the syntactic structure of one language. Equations can only be defined using attributes that belong to phyla of the language. Therefore, consistency constraints between phyla of different documents cannot be expressed since they do not have a phylum in common. This could be simulated by defining an artificial root phylum and considering different documents as children of that root phylum. All documents would then collapse into one super-document. In that case, however, concurrent editing of multiple users on different related documents could not be supported for two reasons. Firstly, each tool would open, display and save the super-document since all these operations by definition operate on documents. Then concurrent development would be hampered since users would have to manually merge their changes in the documents. This might be solved by storing the underlying abstract syntax tree in an object database system. Secondly, concurrent development would be hampered because each inter-document consistency constraint would have to be specified by equations modifying root phylum attributes. Then multiple users working on the same set of documents would cause many concurrency control conflicts since all inter-document consistency checks would have to modify the attributes of that root phylum. Hence the root phylum would become a bottleneck. In GTSL this is solved by the implicitly defined common root increment of class `DocumentPool`, semantic relationships and the transaction properties of interactions. The implicitly defined instance of `DocumentPool` serves as a root increment for all documents. Unparsing, however, starts at documents, i.e. at the children of `DocumentPool`. The document pool may be used in path expressions in order to establish semantic relationships between different documents. For that purpose no changes need be carried out at the document pool. If a semantic relationship is once established, it can be used to establish further (more fine-grained) relationships. As an example consider the `ImpFrom/ExpTo` relationships between `OpImport` and `OpName` increments on Page 152. They are based on the `ImpFrom/ExpTo` relationship between `ImportList` and `Module`. To create the relationship between `OpImport` and `OpName` increments, therefore, the document pool need not be accessed at all. Therefore, concurrency control conflicts merely occur if new documents are created or existing documents are deleted concurrently. Evenso conflicts are unlikely because documents are not created or deleted very often. Moreover, the granularity of transactions are commands, and locks are held only during command execution, i.e. for a few hundred milliseconds.

As the direct comparison between the semantic rules on Page 153 and the SSL counterpart further suggests, GTSL semantic rules are more concise. This is because semantic rules are inherited from super classes where common semantic properties of several increment classes can be defined. In addition, GTSL contains a number of predefined classes and attributes such as `SymbolTable`, `ErrorSet` and `errors`. A tool builder may exploit these definitions in his or her static semantics and inter-document consistency specification.

Command Definition in SSL: *Transformations* can be defined for each SSL phylum. Transformations can be used to define tool commands. Either they are offered at the user interface in pop-up menus, or they can be invoked from a command-line interface. Transformations are specified in SSL by means of pattern matching. As an example, consider the transformations below that define commands for modules of the Groupie interface language.

```
transform module
  on "Expand Module"
    ADTModule(a,b,OpList,d,e,f) : ADTModule(a,b,operation,d,e,f),
  on "Expand Module"
    ADTModule(a,CommentNull,c,d,e,f) : ADTModule(a,CommentPlaceholder,c,d,e,f),
  on "Delete Comment"
    ADTModule(a,b,c,d,e,f) : ADTModule(a,CommentNull,c,d,e,f),
```

The string that follows the keyword `on` is the name of the transformation. Each transformation is then specified by means of two patterns delimited by a colon. A transformation name is included in a menu if a match can be found for the selected increment with the left-hand side pattern. If the user selects a transformation, the right-hand side is substituted for the matched pattern. Command `Expand Module` is, therefore, included in the menu if either the operation list is still a place holder, or the comment has been deleted. Admittedly, this pattern matching based command definition is more concise than interactions in GTSL. As a trade-off, however, a further weakness arises.

This weakness is that SSL cannot express the fact that particular static semantic constraints must not be violated. Instead it postulates a laissez-faire strategy. This is because the above transformations only depend on the abstract syntax and a tool builder cannot specify that a transformation has to be undone in case of a particular static semantic error. Tools tolerate any erroneous input and only display errors to the user. As discussed in Section 2.3, there are situations where this is not appropriate. In GTSL the required behaviour can be defined within interactions and `ABORT` statements can be used that undo the effect of command executions that would otherwise cause intolerable errors.

A further weakness is that a significant number of transformation rules will be redundant for different phyla. For those phyla, such as operations or imports that also have comments as child phyla the same transformations have to be defined. In GTSL the respective commands are defined in interactions of super classes and inherited by subclasses. The Groupie specification, for instance, contains a class `Commentable` that serves as a common super class for the various increment classes that have comments as abstract syntax children.

SSL is in no way capable of defining concurrency constraints on tool execution, unlike GTSL interactions which have the semantics of ACID transactions. In addition, the tool builder may use `COMMIT` or `ABORT` statements in order to explicitly determine the success or failure of the interaction. In the case of a failure, all modifications performed within the interaction are undone. Upon success they are immediately persistent and visible to all concurrent users.

Structuring SSL specifications: SSL itself does not provide any structuring concepts. Within an SSL specification, declaration of phyla, operators and functions may be arbitrarily interleaved. Thus, the burden of arranging for an understandable structure is put completely on the tool designer. A designer may structure an SSL specification into several UNIX-files. As the Synthesizer Generator first runs the C-preprocessor on its input, the CPP `#include` directive can be used to compose the different files to obtain the overall input. In contrast, GTSL defines a number of concepts for structuring tool specifications. Import and export interfaces are included for classes. Properties can be hidden from client class specifications. In addition, properties that are common to more than one class can be declared in a common super class and are then inherited in all subclasses.

Reuse: Reuse of SSL specifications is not supported by the language, since SSL has no language constructs for designating reusable components. GTSL has been defined in a way that allows classes to be easily reused. Import statements and inheritance directives display the dependencies that a class has to other classes. If the class is to be reused, all imports and inheritance statements have to be resolved. This can be done either by reusing the imported and inherited class as well or by satisfying the imports and inheritance statements by substitutes.

Validation: SSL has well-defined scoping rules and an elaborated type system. The constraints imposed by them are checked during generation of a tool. The scoping rules and the type system of GTSL are equally well elaborated and formally defined. They are more complex than the SSL static semantic rules due to polymorphism, multiple inheritance, redefinition and import/export of properties and methods.

6.6.2 METAL, TYPOL and PPML

The Centaur system evolved from the Mentor-Project [DGKLM84] that was carried out at INRIA between 1974 and 1986. Mentor, as well as its successor Centaur, are meant to support the prototyping of languages. Therefore, they provide a framework which can be used to define a language in terms of syntax, static and even dynamic semantics. To do this, Centaur offers different languages. Abstract and concrete syntax is defined using the language *METAL* [KLM83]. Unparsing schemes are defined in the *pretty-printing meta language (PPML)*. Static semantics and also dynamic semantics can be defined in a rule-based language which is called *TYPOL* [Des88]. TYPOL is based on structural operational semantics [Plo81].

Abstract Syntax in Metal: Similar to SSL, METAL supports the definition of the abstract syntax of a language in terms of *phyla* and *operators*. Identifiers in lower-case letters denote operators and phyla identifiers are given in upper-case letters. An operator declaration consists of two components delimited by ->. The left-hand side defines the name of the operator and the right-hand side defines a list of argument phyla. A phylum is declared by an identifier followed by ::= and a list of operators, which can be applied to the phylum. For a better comparison with GTSL and SSL, consider the METAL fragment below. It defines an excerpt of the abstract syntax of the Groupie interface language.

```
definition of MIE is
  abstract syntax
  MODULE      ::=  adt_module ado_module f_module tc_module;
  COMMENT     ::=  comment comment_nil;
  OP_LIST     ::=  op_list;
  OPERATION   ::=  function procedure
  adt_module  ->   IDENT COMMENT IDENT OP_LIST IMPORT_PART IDENT;
  ado_module  ->   IDENT COMMENT OP_LIST IMPORT_PART IDENT;
  f_module    ->   IDENT COMMENT OP_LIST IMPORT_PART IDENT;
  tc_module   ->   IDENT COMMENT IDENT_LIST IDENT;
  function    ->   IDENT PAR_LIST IDENT COMMENT;
  procedure   ->   IDENT PAT_LIST COMMENT;
  op_list     ->   OPERATION * ...;
  comment     ->   implemented as STRING;
  comment_nil ->   implemented as SINGLETON;
  ...
```

List phyla declarations (such as OP_LIST) are easier in METAL and GTSL than in SSL. Also place holder definitions are implicit in METAL and GTSL while they have to be explicitly specified in SSL. As in SSL, common properties of operators cannot be specified in METAL as such. Instead they have to be duplicated for each operator. GTSL overcomes this by the inheritance of abstract syntax children.

Concrete Syntax in Metal: The Centaur system distinguishes between the definition of the input syntax and the output representation of an abstract syntax tree. The input syntax is defined in METAL, whereas the output representation is defined in a dedicated language (PPML). The concrete syntax in METAL is defined in terms of *rules*. A rule in METAL resembles a production in context-free grammars, except that it adds a call to a tree building function. This call defines the relationship to the abstract syntax definition. As an example, consider the concrete syntax definition excerpt below.

```
rules
<module>     ::= #DATATYPE #MODULE <ident> #; <comment>
                   <ident><op_list><import_part>
                 #END #MODULE <ident>#.;
                 adt_module(<ident>.1,<comment>,<ident>.2, <op_list>,
                            <import_part>,<ident>.3)
<comment>    ::= %STRING ;
                 comment-atom(%STRING)
...
```

Besides defining the input syntax a METAL specification must determine those increments that can be edited in free textual input mode. Therefore, METAL provides the means to define *entry points* for the multiple entry parser. Entry point specifications are also given in a rule based format. The left-hand side of the rule defines the axiom of the grammar. The right-hand side consists of the name of the entry phylum and the respective non-terminal symbol of the concrete syntax specification. Phyla `MODULE`, `COMMENT` and `OP_LIST` of the example are declared as entries below. These entry point definitions correspond to interaction definitions in GTSL, which define free textual input commands.

```
rules
<module>     ::= [COMMENT] <comment> ;
<module>     ::= [EXPORT_PART] <export_part> ;
<module>     ::= [OP_LIST] <op_list> ;
...
```

In short, the way of defining the abstract and concrete syntax of a language as well as the entry points for a parser is rather lengthy. This is because information about the abstract syntax is redundantly contained in the BNF-like rule definition of the concrete syntax and, similarly, information about the BNF-like rule definition is also contained in the entry point definition. As a consequence, changes to the syntax may cause tedious updates of the METAL definition.

Unparsing Schemes in PPML: Unparsing schemes are defined in Centaur using the pretty-printing meta language (PPML). They are called *prettyprinters*. More than one prettyprinter may be defined for a tool. A tool user can then switch at run-time between the different prettyprinters in order to change the way documents are displayed. A PPML specification consists of a set of rules that map operators, defined in the abstract syntax definition, to a textual output representation. A rule has the form `pattern -> [format]`. A pattern represents an operator with formal parameters. A format definition defines how these parameters are positioned and interleaved with text, such as keywords and symbols. The formatting is based on the notion of a *box*. Each leaf of the abstract syntax tree and each text is considered to form an atomic box. Atomic boxes are glued to compound boxes by square brackets in a formatting specification. In doing so, separator definitions between boxes given in pointed brackets define how a box is aligned. As an example consider an excerpt of a prettyprinter for Groupie interface definitions below:

```
prettyprinter Standard of MIE is
rules
  adt_module(*ident1, *comm, *ident2, *op_list, *import_part, *ident3) ->
    [<v 0,0> [<h>  "DATATYPE MODULE " *ident1 ";"]
     <v 4,1> *comm
     <v 4,1> [<v 0,0> "EXPORT INTERFACE"
              <v 2,1> [<h> "TYPE " *ident2]
              <v 2,1> *op_list
             ]
     <v 4,2> *import_part
     <v 0,1> [<h> "END MODULE " *ident3 "."]
    ]

  op_list(*operation, **op_list) ->
    [<v 0,0> *operation(**oplist)
    ]
  ...
end prettyprinter
```

The first rule defines the unparsing of a complete ADT module. A module is composed of five vertically aligned boxes. The first box contains the module head. It is itself composed of three atomic boxes, which are horizontally aligned. The second box is an atomic box that contains a module's comment. It has a horizontal indentation, to the outer box, of four spaces. Moreover, a blank line is inserted before the comment. Similar definitions define alignments for the type identifier, operation lists, import interface and the module tail. The unparsing scheme definition is very similar to both GTSL and SSL. PPML is more powerful in this respect since it supports the definition of several prettyprinters, which GTSL does not for reasons that have already been discussed.

Static Semantics in TYPOL: A TYPOL specification is based on a METAL abstract syntax definition. This syntax definition is imported with a use directive. A TYPOL specification then consists of *sets* of rules. Each set of rules is a named collection of *inference rules*. These rules define a formal system in which it is possible to prove that a particular proposition holds. The declaration part of a set contains a *judgement* which is a signature definition for the proposition to be proved by the set. Each of the inference rules has two parts called *numerator* and *denominator*. The general form of a rule is:

```
 <nominator>
-------------
<denominator>
```

The denominator is a *sequent* and the nominator is a list of sequents also called *premises*. In natural semantics sequents express the fact that some hypotheses are needed to prove a particular proposition. A sequent, therefore, has two parts which are delimited by the turnstile symbol $\vdash$. The first part of the sequent contains the *hypotheses* and the second part is called *consequent*. The consequent of the denominator sequent is called *subject* of the rule. Sequents are built from lists of expressions which are, in turn, formed from variables and operators defined in the METAL abstract syntax specification. Premises are then formed by a list of sequents separated by the ampersand sign &.

```
program SCOPING_MIE is
set MODULE_OK is
  judgement MODULE |- MODULE;

  NAME_OK(MOD|-ID2) & TYPE_OK(MOD|-ID2) & OPLIST_OK(MOD|-OPL) & IMPORT_OK(MOD|-IM)
  --------------------------------------------------------------------------------
  MOD |- adt_module(ID1,_, ID2, OPL, IM, ID3);
  ...
END MODULE_OK;

set NAME_OK is
  judgement MODULE |- IDENT;

  EQUAL(ID1|-ID3) & NOT_EQUAL(ID1|-ID2) & NOT_IN_OPLIST(OPL|-ID1)
  -------------------------------------------------------------------
  MOD |- adt_module(ID1,_, ID2, OPL, IM, ID3);
  ...
end EXPORT_OK;

set NOT_IN_OPLIST is
  judgement OP_LIST |- IDENT;

  op_list[] |- _;

  NOT_EQUAL(OPID|-ID) & NOT_IN_OPLIST(TAIL|-ID)
  -----------------------------------------------
  op_list[procedure(OPID,_,_).TAIL] |- ID;

  NOT_EQUAL(OPID|-ID) & NOT_IN_OPLIST(TAIL|-ID)
  -----------------------------------------------
  op_list[function(OPID,_,_,_).TAIL] |- ID;
end OP_LIST_OK;
...
end SCOPING_GRE;
```

Figure 6.17: Excerpt of a TYPOL Specification Defining a Scoping Rule

Intuitively, an inference rule states that if all the sequents in the premises hold, the proposition expressed by the denominator sequent holds. The order of the premises in a rule is not important. An example of a TYPOL specification is shown in Figure 6.17. It defines a fragment of the scoping rules for the Groupie interface language.

The first rule set defines the overall correctness of modules. The subject of the first of its rules is determined by the METAL operator `adt_module`. Thus, this rule defines static semantics of modules that are ADT modules. Such a module is correct if the sequents in the nominator part can be proved. These require the ADT module's name, its type, the operation list and the import to be correct. Very similar rules are defined for the other module types, but they are omitted here for reasons of brevity. In order to prove the correctness of the module name, the judgement defined in the rule set `NAME_OK` must hold for the match determined by `MODULE_OK`. Thus, we must be able to deduce the module identifier from the module hypothesis. In order to prove this, the three sequents in the nominator part of the rule must hold. Informally, this means that the identifier in the module's head must be equal to the identifier in the tail. Moreover, this identifier must be different from the exported type name. The third sequent requires that the module identifier must not occur in the operation list. While the rule sets for the first two sequents are trivial, the set to be proved for the third sequent requires a list traversal. This is defined in rule set `NOT_IN_OPLIST`. It consists of three rules. The first is an axiom that always holds. It defines that the empty operation list does not contain any identifiers. The second rule defines that the identifier does not occur in the list if the first element is a procedure, its name is unequal to the identifier and the identifier does not occur in the tail of the list. The third rule defines the same for functions.

TYPOL provides a powerful means for defining scoping rules and the type system of a language. The tool builder can define them in a declarative way and abstract from particular execution dependencies. Resolution of these dependencies is implemented in the TYPOL compiler by mapping a TYPOL program to a Prolog program which exploits backtracking in order to find a valid execution sequences of rules.

Just as deficiencies were identified for SSL in the previous section, TYPOL is not capable of expressing inter-document consistency constraints. Again the reason is that the universe over which expressions can be defined in TYPOL is determined by the abstract syntax specification of one language.

In addition, TYPOL rules are not as concise as they could be. Consider the last two rules in set `NOT_IN_OP_LIST`. They only differ in the operator that is applied to the first list element in order to bind `OPID`. Similarly, the definitions of rules in set `MODULE_OK` only differ regarding the checks of exports since all module types have imports and names. In GTSL, this deficiency is removed, since semantic rules are inherited by subclasses. The rule for module name correctness, for instance, is specified in class `Module` and inherited by all subclasses.

A further drawback of TYPOL is that it can only define correctness conditions for static semantics. It is not possible to define error messages to be displayed if an error is detected. Hence, it is particularly difficult for a user who may be unfamiliar with a language to understand why a sentence is considered wrong. Moreover, in TYPOL, it is not possible to define the means that would prevent the introduction of errors, such as change propagation to dependent increments. GTSL differs from that in that it includes a number of flexible mechanisms that can be used to define error messages and also that the strategy for accepting or rejecting erroneous input can be defined in interactions.

Tool Interactions: Centaur cannot generate tools that support structure-oriented editing. It can only generate editors that facilitate free textual input of those increments whose phyla have been declared as entries. Therefore, Centaur does not have any means for defining the interactions and their preconditions. Moreover, concurrent free textual input of multiple users is not supported at all.

Structuring Metal, TYPOL and PPML specifications: METAL and PPML specifications can be structured into *chapters*, which may be nested. An atomic chapter, which is not nested further, consists of an abstract syntax specification and of concrete syntax rules. Chapters are meant to be used for structuring the overall METAL specification according to the semantic concepts of the language. Besides grouping related component specifications together, chapters do not have any further semantics. A TYPOL program is structured into sets of rules, each of which is a formal system. Moreover, the METAL abstract syntax definition a TYPOL program is based on must be defined by a use directive.

Structuring in GTSL is horizontal, that is different properties of the same increment are specified together in an increment class. This is particularly appropriate since the different properties are highly dependent on each other. Changes to the abstract syntax immediately affect the unparsing scheme as well as semantic rules. Many changes can thus be performed in a GTSL specification without affecting any other class at all. In addition, explicit imports denote the dependencies that have to be considered during a change. Unlike this, structuring in Centaur is vertical in the sense that the same properties of different increments are specified together and only dependencies of these properties are explicit. This structuring principle is inferior to GTSL since many dependencies of other properties are not explicit at all. The TYPOL example rule set `NOT_IN_OP_LIST`, for instance, uses the METAL operators `function` and `procedure` without having to declare their use explicitly.

Reusing Metal, TYPOL and PPML specifications: METAL, TYPOL and PPML specifications do not support reuse. As we have seen above, it is possible to identify different components of these specifications. The relationships they have to each other, however, are not explicit. If a component is to be reused it is hard to identify the other components that must be either reused as well or constructed anew.

Validating Metal, TYPOL and PPML specifications: METAL, TYPOL and PPML are strongly typed languages with well defined scoping rules. They are validated during editing of specifications with dedicated editors that have been generated with Centaur itself. Moreover, Centaur offers a debugger which can be used for validating dynamic semantics of TYPOL programs. As mentioned above, inter-document consistency constraints between METAL, PPML and TYPOL specifications are not validated.

6.6.3 PROGRESS

PROGRESS is a specification language for defining graph grammars. It evolved from the IPSEN [Nag85] project where graph grammars were used for the definition of abstract syntax graph structures [ELS87, ELN^{+}92]. PROGRESS, however, is not explicitly devoted to the specification of syntax graphs, but is intended as a language for the definition of arbitrary graph classes. In [Sch91a], for instance, it is used for the definition of family chart graphs and recently it has been successfully applied to the definition of a CIM environment. As a general purpose language, it can obviously not provide the specific support for tool specification that SSL, the Centaur languages and also GTSL provide. We shall now discuss the extent to which PROGRESS meets our requirements. The comparison with GTSL will highlight the benefits of defining a domain specific language for tool specification. GTSL, however, will almost certainly fail if it is used for specifying problems from other domains.

Abstract Syntax: A PROGRESS specification consists of the definition of *node types*, *node classes*, *attributes*, *edge types*, a *start node type*, *path expressions*, *graph tests*, *graph rewriting rules* and *transactions*. A node type is defined by declaring a type name and determining at least one super node class. Node classes, in turn, define common properties that several node types inherit. Node classes can, themselves, inherit from multiple other node classes. Attributes in PROGRESS are used to store lexical values as well as semantic information. Attribute types, however, cannot be defined in PROGRESS. A programming language has to be used for this purpose. An edge type is defined by determining a name and defining the source and target node classes. PROGRESS is polymorphic. Therefore, edges may start from or lead to nodes whose types have been derived from the respective node class determined in the edge type definition. This corresponds to property polymorphism in GTSL. As an example, consider an excerpt of a PROGRESS specification for the Groupie interface tool depicted in Figure 6.18.

```
node class GRAPH_NODE end;                  node type ADTModule: MODULE              end;
  intrinsic Id: Number := 0;                node type ADOModule: MODULE              end;
end;                                        node type ModName: MOD_NAME              end;
node class IDENTIFIER is a GRAPH_NODE       node type PhOpList: PLACEHOLDER,OP_LIST  end;
  intrinsic Name:STRING:=nil;               node type OpList:  OP_LIST               end;
end;                                        node type PhOp:    PLACEHOLDER,OPERATION end;
node class LIST_HEAD   is a GRAPH_NODE end; node type Function: OPERATION            end;
node class LIST_ELEM   is a GRAPH_NODE end; node type Procedure: OPERATION           end;
node class MODULE      is a GRAPH_NODE end;
node class PLACEHOLDER is a GRAPH_NODE end; edge type ToFirst:  LIST_HEAD --> LIST_ELEM;
node class MOD_NAME    is a IDENTIFIER end; edge type ToLast:   LIST_HEAD --> LIST_ELEM;
node class OPERATION   is a LIST_ELEM end;  edge type ToElem:   LIST_HEAD --> LIST_ELEM;
node class OP_LIST     is a LIST_HEAD end;  edge type ToNext:   LIST_ELEM --> LIST_ELEM;
...                                         ...

node type PhModule: PLACEHOLDER,MODULE end;
```

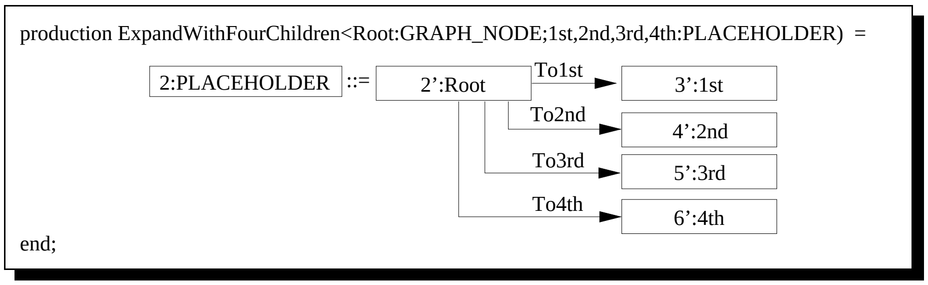
Figure 6.18: Abstract Syntax Graph Structure in PROGRESS

The most general node class is `GRAPH_NODE`. Node class `IDENTIFIER` defines an attribute `Name`, which is used to store the lexical value of the identifier. The attribute is declared to be `intrinsic`, which means that values will be assigned externally.

PROGRESS does not include any primitives for ordered multi-valued edges that are required in syntax graph definitions and are, therefore, included in SSL, METAL and GTSL. In fact, a number of additional edges are needed to implement these list structures. Inheritance may be used to partly overcome this deficiency. The example, therefore, defines two abstract node classes `LIST_HEAD` and `LIST_ELEM` for that purpose. Heads implement a class for the source nodes of a multi-valued edge and `LIST_ELEM` defines the target class. Four edge types are introduced for lists, namely `ToFirst`, `ToLast`, `ToElem` and `ToNext`. Edges of the first two types connect the head of a list with the first or last element respectively. `ToElem` edges connect the head with each element and `ToNext` edges connect each element of a list with its successor element.

Node class `MODULE` defines the common properties of arbitrary module node types. Therefore, the class of node type `ADTModule` and the other kinds of Groupie modules is `MODULE`. Note that we also have to define a node type to model module place holders. This is due to the fact that PROGRESS, as a general purpose language, does not provide implicit mechanisms to handle the concept of place holders since these are rather specific to syntax-directed tools. GTSL defines this concept in the basic predefined class `Increment` and implements it in implicit methods. It thus relieves the tool builder from defining a number of additional node types that are required in PROGRESS.

To fully determine the structure of abstract syntax graphs a number of graph rewriting rules have to be defined. The graph class defined by the PROGRESS specification is then the set of those graphs that can be generated from a start node by applying the rewriting rules. An example of a graph rewriting rule is given below. It defines a generic rule that expands a place holder with four children.


production ExpandWithFourChildren<Root:GRAPH_NODE;1st,2nd,3rd,4th:PLACEHOLDER) =
2:PLACEHOLDER
::=
2':Root
To1st
To2nd
To3rd
To4th
3':1st
4':2nd
5':3rd
6':4th
end;

The rule may then be applied within a transaction that determines actual parameters of the formal node type parameters. A transaction that invokes the above rewriting rule in order to expand the different types of modules is defined below:

```
transaction ExpandModule(Current:Number; type:String) =
ASTIsOfType<MODULE>(Current) &
choose
  when valid type '=' "ADTModule" then
    ExpandWithFourChildren<ADTModule,ModName,TypeName,OpList,ImportInterface>();
else
  when valid type '=' "ADOModule" then
    ExpandWithThreeChildren<ADOModule,ModName,OpList,ImportInterface>();
else
  when valid type '=' "FModule" then
    ExpandWithThreeChildren<FModule,ModName,OpList,ImportInterface>();
else
  when valid type '=' "TCModule" then
    ExpandWithTwoChildren<TCModule,ModName,TypeNameList>();
end;
```

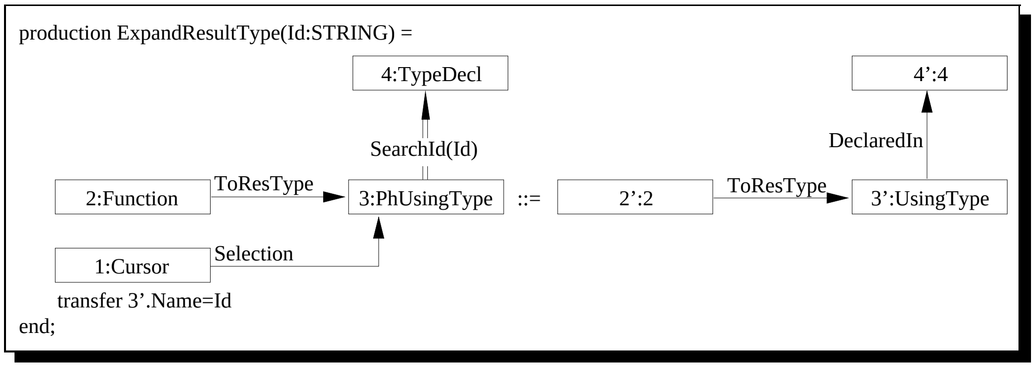
The transaction uses the test `ASTIsOfType` to check whether the node `Current`, which is to be expanded, is a `MODULE` place holder node. Tests are similar to left-hand sides of graph rewriting rules. They return a boolean value that defines whether the test was successful or not. If the selected increment is a place holder of a subtype of `MODULE` then a case switch selects which particular rule to apply based on the argument `type`. Note that this case switch is necessary because the actual node type parameters of the graph rewriting rules need to be determined. In GTSL this is solved in a more elegant way by the implicit methods `expand` and `scan` to establish an abstract syntax tree and `collapse` to delete it. The case switch performed in the transaction above is implicitly performed in GTSL during dynamic binding. An `expand` message sent to an increment is dynamically bound to the right `expand` method, which is the one defined in the class of the increment. Therefore, the test `ASTIsOfType` is not required in GTSL, either.

A further weakness arises if we consider the definition of documents. Besides the edge types that represent abstract syntax relationships, we will also have to define edge types for semantic relationships. These might connect nodes from different documents. PROGRESS, however, does not separate these edge types. Therefore, we cannot identify documents within a project-wide abstract syntax graph. As discussed in Section 3.1, the definition of documents is of primary importance for version and configuration management purposes. Therefore, the specification has to be extended with additional node and edge declarations that specify documents. This is extensively discussed in [Wes91]. In GTSL this specification overhead is not required because documents are implicitly defined due to the distinction of abstract syntax children and semantic relationships. Documents are then determined by root increments, i.e.

instances of the predefined class `Document`. All increments that are reachable from a given root increment via abstract syntax children implicitly belong to the document.

Lexical Syntax, Concrete Syntax and Unparsing Schemes: The problems of defining syntax and unparsing schemes are highly specific to syntax-directed tools. In addition, they cannot reasonably be specified with graph grammars. As a language for graph grammars, PROGRESS does not provide any primitives for unparsing schemes, concrete and lexical syntax and these concerns cannot be defined at all with PROGRESS.

Static Semantics and Inter-Document Consistency Constraints: Graph rewriting rules are not only used for generating abstract syntax trees. They are also suitable for defining static semantics and inter-document consistency. For an eager approach to handling these constraints, which does not tolerate errors, graph rewriting rules may restrict the applicability of the rule to contexts that do not violate static semantics. *Path expressions* can be used to define these context sensitive restrictions on the left-hand side of the rule. The rule is then applicable only if a match can be found in the host graph that includes a path that is conform to the expression. As an example consider the following rule, which defines that using type place holders may only be expanded if the new value matches a declaration.



Path expressions are graphically depicted in the graph rewriting rule as double-lined arrows labelled with their name and the actual parameters of their invocation. Informally, the path expression `SearchId` searches for the node of type `TypeDecl` where the string passed as parameter is declared. As the path expression appears on the left-hand side of the rewriting rule, the rule can only be applied if such a path to a `TypeDecl` node can be found in the host graph. If found, the related identifiers are connected with a *context-sensitive edge*, which reflects the use/declare relation between the two nodes. The formal definition is given below:

```
path SearchId(p:STRING):IDENTIFIER --> IDENTIFIER =
     instance of UsingType & <--ToResType--
   & <--ToElem--
   & <--ToOpList--
   & (   (--ToExpTyp--> & Definition(p))
      or (--ToImpInt-->...))
end;

restriction Definition(Id:STRING) : IDENTIFIER =
  valid self.Name=Id;
end;
```

Note that PROGRESS path expressions are more powerful than their counterpart in GTSL. PROGRESS path expressions can express parallelism which is not required for tool specification and, therefore, neither included in the example nor in GTSL. PROGRESS path expressions are set-oriented, i.e. intermediate steps are always sets of nodes rather than single nodes as in GTSL. GTSL path expressions are, however, safer. Firstly, steps in GTSL path expressions have an explicit cardinality and the type system can prove that the assumptions of a tool builder regarding the cardinality hold. In PROGRESS the tool builder cannot be sure whether a path expression matches multiple target nodes. In the above case, the path expression `SearchId` must be considered wrong if it reveals a set of nodes rather than the single node that declares the identifier. The PROGRESS type system cannot find out about that. As a further contribution to safer path expressions, GTSL offers covariant property redefinition. In PROGRESS it is not possible to redefine edge types in such a way that they connect more specific node types. The above example, therefore, uses an `instance of` operator, which is the equivalent of a GTSL type cast, to be able to traverse along an outgoing `ToResType` edge from a node whose static type is `IDENTIFIER`. In GTSL the use of unsafe type casts can always be avoided with covariant redefinition of abstract syntax children or semantic relationship links, for example.

The applicability of path expressions and restrictions is not confined within an abstract syntax graph of one document. Therefore, inter-document consistency constraints may be expressed in the same manner as static semantics constraints.

While PROGRESS is very appropriate for the definition of eager enforcement of static semantics and inter-document consistency constraints, a strategy that tolerates errors is more difficult to define. The task that has been performed by the single rule in the eager strategy is, for the lazy strategy, split into four rules as given in Figure 6.19. The first rule expands the place holder of a using type without checking for its consistency. The second rule binds a correct using type to its declaration with a semantic edge `DefinedIn`. It also removes an error descriptor from a set of errors that is associated as an attribute to the using type node. The third rule unbinds a using type, whose declaration is no longer available, and removes the semantic edge `DefinedIn`. The last rule inserts an error descriptor into a set of errors for each using type node that is not properly bound to a declaration. Note that this specification exploits the fact that PROGRESS rules are applied non-deterministically by the PROGRESS interpreter.

It is unclear to us how efficiently the PROGRESS interpreter, which is still under construction, can execute lazy specifications like the one above. It will involve a significant complexity to check all types that do not have an outgoing semantic edge against path expression `SearchId`. This concern is reinforced if we consider the complexity that is required for interpretation of path expressions like `SearchId`. In PROGRESS any step in a path expression is an operation on a set with the inherent complexity. In GTSL steps that denote abstract syntax children or links in semantic relationships in path expressions are interpreted within the constant time that is required for dereferencing an instance variable. It would be even more complex to check whether the binding of all bound types is still correct, since it involves negation of a rather complex path expression. This might require significant backtracking. In GTSL this problem is remedied, since the specification of static semantic rules guides an incremental evaluation in the sense that rule predicates refer to incremental changes or deletions made since the last check.

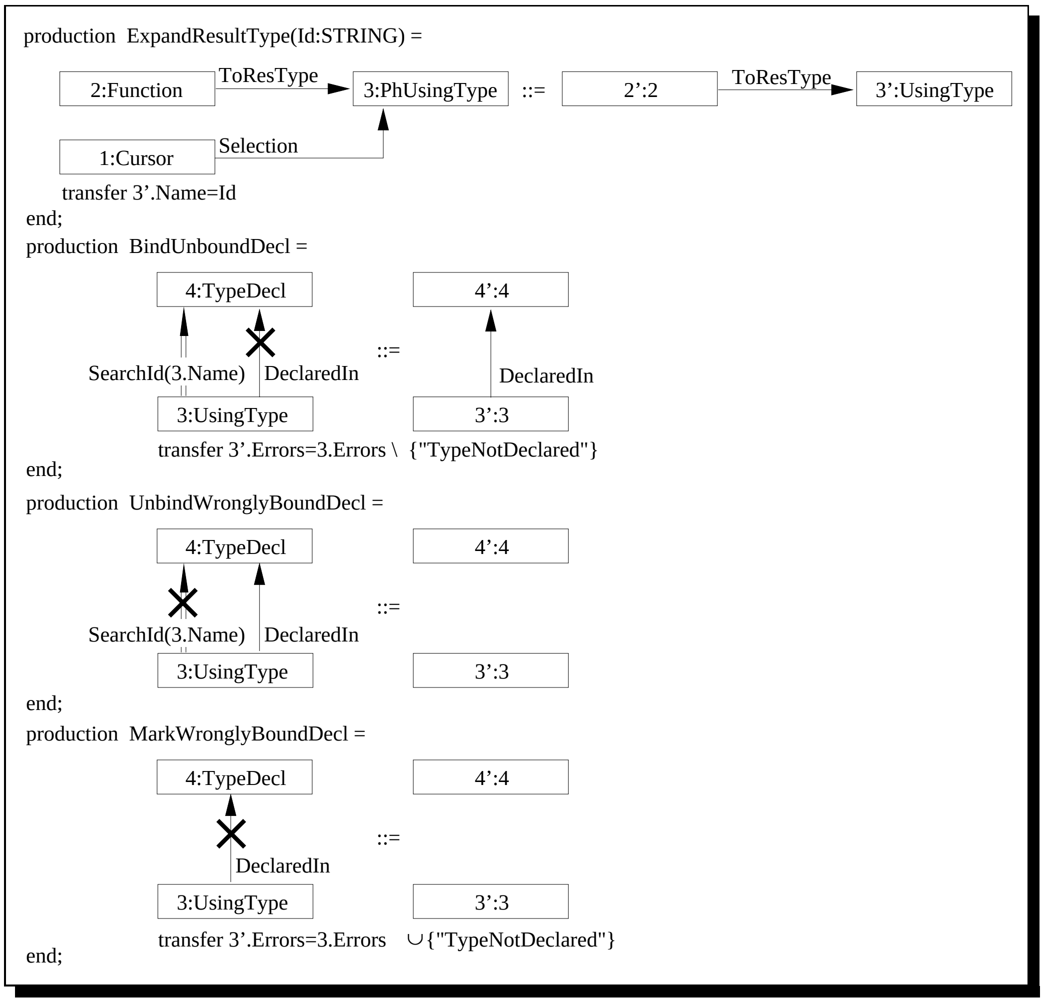


Figure 6.19: Scoping Rule in PROGRESS

Tool Commands: A tool builder can define the preconditions under which a command is offered using the left-hand side of graph rewriting rules. It is, however, not possible to define how a user can choose to apply a particular graph rewriting rule to a host graph. Neither can PROGRESS define particular user dialogues that are required in order to inform a user about some particular error, for example. PROGRESS specifications are instead used to generate an abstract data type module which implements the graph class defined by the PROGRESS specification. Transactions and rewriting rules are generated into operations of this abstract data type and some upper layers, which have to be coded or generated otherwise, might use these operations to implement tool commands, for example.

We have seen that PROGRESS transactions can be used to bind multiple graph tests and rewriting rules together into a larger execution unit. The semantics of a transaction in PROGRESS is then atomicity and durability, i.e. all rules are applied completely or not at all. Once successfully completed, the effect of a transaction is persistent. Compared with transactions in database systems, or interactions in GTSL, a PROGRESS transaction lacks the isolation property. Therefore, PROGRESS transactions cannot be used for specifying concurrent accesses of multiple users to the abstract syntax graph.

Different levels of abstraction: PROGRESS does not offer different levels of abstraction on its specification as GTSL does. Obtaining an overview of the abstract syntax graph structure is rather difficult. A tool builder, therefore, has to consider several related graph rewriting rules and must assume an order in which they are applied. In GTSL, the entity relationship model is devoted to this purpose and represents the structure in a hierarchical manner on a type level of abstraction.

6.7 Summary

In this chapter, we have delineated a number of requirements of tool specification languages. These languages must support the definition of an abstract syntax for target languages to be edited and analysed by the specified tool. Moreover, the concrete syntax and unparsing schemes for target languages must be definable. Any tool specification language must enable static semantics and inter-document consistency constraints to be defined, even between documents of different types. They have to support definition of concurrently executable user commands. These definitions should be carried out at appropriate levels of abstraction. As general specification language requirements, they should enforce structuring of specifications into components to obtain comprehensible, maintainable and reusable specifications.

We have introduced several languages for tool specification on different levels of abstraction. At a very high-level of abstraction, the syntax of the language is defined in terms of an extended and normalised BNF. At a lower level of abstraction, we have suggested an extended entity relationship model based on the OMT notation. It can be used to add structural concerns for static semantics and inter-document consistency based on an abstract syntax tree definition that is generated from an ENBNF. We have then suggested a domain-specific object-oriented language that defines the behaviour of increment classes, which have been identified in the entity relationship model. Unparsing schemes declare the textual representation of documents. Methods define the operations that are exported to other classes in order to allow for increment modifications. Semantic rules define static semantics and inter-document consistency constraints in a declarative way. Interactions define tool commands and their appearance in context-sensitive menus. The language distinguishes between different kinds of increment classes in a domain-specific way and distinguishes between different kinds of properties, i.e. attributes, abstract syntax, and semantic relationship links. These distinctions enable a domain-specific type system to be defined. That type system enables a number of specification errors to be detected. We have then suggested a library of predefined classes that provide solutions to very common problems including command definitions for structure-oriented editing and version and simple configuration management. Inheritance defined for GTSL is then exploited to reuse classes from this library in tool-specific classes.

None of the related tool specification languages fulfils all requirements. The reason is that they have been defined for different purposes. The Synthesizer Generator has been built to generate tools for programming environments. These environments are typically used by single users and are only built for a single language. Thus inter-document consistency constraints, concurrency control mechanisms or version and configuration management strategies for supporting multiple-users have not been considered at all. Consequently, SSL cannot be used to define these concerns. The Centaur environment was intended as an environment for prototyping language definitions. For this purpose, it is again not important to have concurrency control, version and configuration management and inter-document consistency constraints. Nor do user-interactions play an important role here. Therefore, the languages used by Centaur do not address these concerns. PROGRESS is intended as a general purpose language for the definition of graph grammars. As such it does not address the specific requirements that arise during construction of syntax-directed tools, such as definition of lexical and concrete syntax, unparsing schemes and tool commands. However, what all the languages that we reviewed have in common is that their dynamic semantics is formally defined. This is not the case for GTSL which is subject to further work.

Chapter 7

The GENESIS Environment

ENBNFs, entity relationship models, class interfaces and specifications fully determine the tools of an environment. To make the practical use of these languages easier we require a further specification. It will provide the tool builder with an overview of the class hierarchy of a tool specification. A fundamental understanding of the class hierarchy is very important. Firstly, it determines inherited properties, methods, semantic rules and interactions. Secondly, it is important to know whenever an expression is assigned to a property, because assignments are polymorphic and the static type of the expression must be a subtype of the static type of the property. The subtype relationship is, in turn, induced by the inheritance hierarchy. We, therefore, refer to this new language as *inheritance diagrams*. These diagrams are required because the inheritance hierarchy is only determined by the inheritance section of the class interfaces. Even if the tool specification environment offers sophisticated browsing capabilities, it will be too time-consuming to investigate all these sections in order to get an overview of the class hierarchy. Instead, tool builders require a graphical notation that provides this overview at a glance. Inheritance diagrams are defined in Section 7.1.

The different languages specify the same tool at different levels of abstraction. They, therefore, address one of our main requirements. The disadvantage is that there is an inherent redundancy between the contents of the specifications written in the different languages. An immediate consequence of this redundancy is that changes to one definition will affect the consistency of another definition if it contains specification fragments that are redundant to that which was changed. We, therefore, need to define consistency constraints to be able to detect these situations. As examples consider productions of the ENBNF that must be refined by classes in the entity relationship model, which, in turn, have to be refined with an interface and a specification. Keywords on the right-hand side of an ENBNF production determine keywords in unparsing schemes of non-terminal increment classes. Aggregation relationships in an entity relationship diagram must be reflected by abstract syntax children, and reference relationships have to be refined in semantic relationships. With the introduction of inheritance diagrams, the inheritance sections of class interfaces must correspond to the subclass relationship in the graphical notation. We, therefore, define consistency constraints like the above in more detail in this chapter.

Like a programmer who works incrementally and intertwined between design and implementation, a tool builder wants to be able to edit ENBNF, inheritance and entity relationship diagrams, and have the impact of changes propagated into class interfaces and class specifications, for example. The paths for these propagations will be determined by the consistency constraints between the different languages. The need for an integrated environment supporting tool specification arises. In particular, the environment must check and even preserve consistency between the various specifications. This environment is called *GENESIS* (Generation of syntax-directed integrated software development tools).

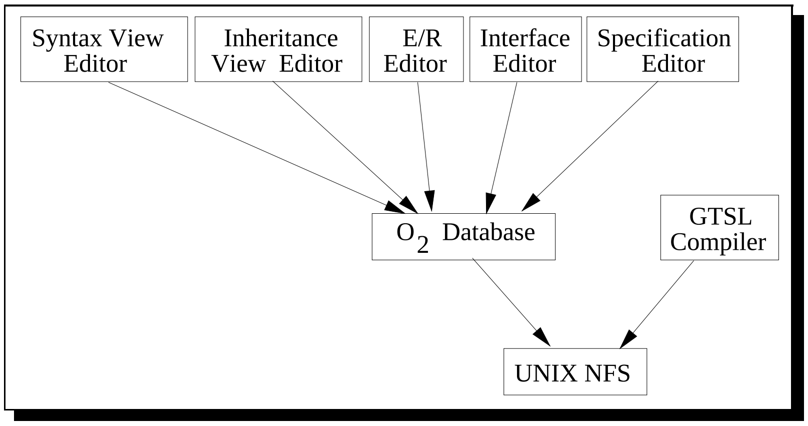


Figure 7.1: The Genesis Environment

The main building blocks of the GENESIS environment are displayed as rectangles in Figure 7.1. Arrows denote usage of services. GENESIS contains integrated editors for the different languages that were identified. These five editors are integrated on top of an O_2 database. The inter-document consistency constraints and the way they are handled by these editors are discussed in Section 7.2. In addition, GENESIS contains a compiler that translates a GTSL specification into an executable tool. Therefore, class interfaces and specifications are dumped from the database to the file-system and are then translated into a tool that is also stored in the file-system. By means of the compiler, GTSL specifications become executable. The implementation of the compiler will be discussed in the next chapter.

7.1 Inheritance Diagrams

In inheritance diagrams, classes are again represented as rectangles. The class names are enclosed in these rectangles. Rectangles that represent increment classes have a solid shape, whereas rectangles that represent non-syntactic classes are depicted by a dashed shape. An arrow that leads from a rectangle c_i to a rectangle c_j denotes that class c_i inherits from c_j. As an example, consider the inheritance diagram of the Groupie module interface tool, displayed in Figure 7.2.

Besides defining the inheritance hierarchy itself, the inheritance diagram also identifies those classes that belong to the specification of a tool. Hence there are as many inheritance diagrams in an environment specification as there are tools that need to be generated. Contrary to that, there is only one entity relationship model for an environment definition. It defines the union of all classes contained in the different inheritance diagrams. Then semantic relationships between different document types are defined in a fine-grained manner as relationships between classes that belong to different tools. The hierarchical structuring mechanisms will be used to separate the classes that belong to different tools into different subdiagrams.

In addition to tool-specific classes, all predefined classes that participate in inheritance rela-

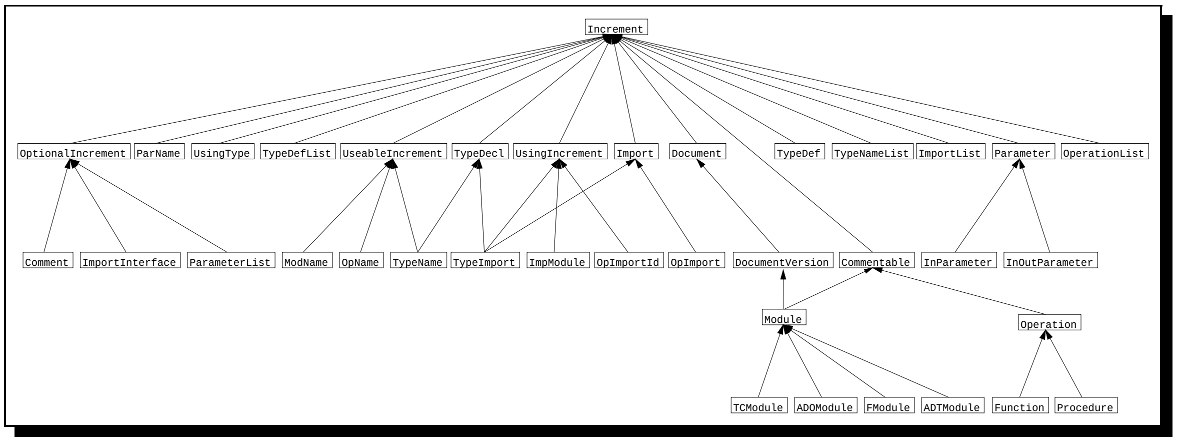


Figure 7.2: Inheritance Diagram of Groupie Interface Editor Specification

tionships with tool-specific classes are included in the inheritance diagram. This allows a tool builder to identify properties, methods, semantic rules and interactions that a tool inherits from predefined classes. Note that not all predefined classes appear in the inheritance diagram, but only those that have tool-specific subclasses. This avoids overloading the diagram with unnecessary information. A tool builder can, nevertheless, define entities in tool-specific classes whose types are predefined classes that were not included in the inheritance diagram. The Groupie interface tool specification, for instance, defines an attribute `DefinedNames` in class `Module` whose type is the predefined class `SymbolTable` although `SymbolTable` is not included in the inheritance diagram.

The obvious static semantic constraints that must hold for the inheritance diagram are the following:

IV1: class names are unique and differ from names of predefined classes,

IV2: only one arrow may be defined between two rectangles,

IV3: the inheritance hierarchy is acyclic, and

IV4: if c_i inherits from c_j then c_i and c_j are of the same kind, i.e. they are either both increment classes or both non-syntactic classes.

7.2 Tools of the Environment

Software development tools are software, too. Tools for developing software development tools should, therefore, meet the requirements that we have identified in Chapter 2 for software development tools. Thus, each tool should support hybrid syntax-directed editing of the respective documents, check and visualise static semantic errors, manage different versions and configurations of the documents and facilitate concurrent editing of multiple tool builders. In particular, the tools should be integrated so that they check and even preserve the various inter-document consistency constraints between the different languages.

We aim at constructing the required tools for the GENESIS environment using the languages that have been defined in this thesis. This will be discussed in Chapter 9. We, therefore, have to define tool requirements. The requirements with respect to syntax and static semantics have already been determined during the introduction of the different languages in the previous chapter. What is still missing is a definition of the inter-document consistency constraints. This definition must also determine the way in which inconsistencies are handled or even avoided. In fact, this definition will be the requirements definition for tool integration in the GENESIS environment.

The general strategy for constraint handling should be to preserve consistency as far as possible without confusing the tool builder(s) involved. There are by definition two different documents involved in an inter-document consistency constraint. Among these, it is possible to identify a determining document and a dependent document. The determining documents are those at the higher level of abstraction where development usually starts. The style to constraint handling will be such that changes made in a determining document are propagated into the dependent document. Consistency constraints will be preserved then. Changes in the dependent document, however, are not propagated to the determining document because it determines the dependent document and not vice versa. We, therefore, guide the tool builder to think about a change at the appropriate level of abstraction, perform the change there and then he or she will have the impact of the change propagated to the affected lower-level documents. In a dependent document, therefore, the style of constraint handling will be lazy in the sense that tools visualise violations, but do not reject commands that cause violations. In that way tool specification is not hampered by enforcing consistency, but guided towards a consistent specification.

7.2.1 ENBNF Editor

ENBNFs are determining documents for the entity relationship model, the inheritance diagram and class interfaces. We first define a number of constraints for consistency between these languages. These constraints will then be handled in such a way that changes are propagated to depending documents. In that way, the tool specification process is again accelerated since these change propagations will generate a significant number of specification fragments from the syntax definition in lower-level documents.

7.2.1.1 Consistency Constraints

We now exploit the normalisation of the ENBNFs to define consistency constraints. They are defined on the basis of the different kinds of productions. For each structure production sp of an ENBNF, there has to be a non-terminal increment class $c(sp)$. This class must be reflected in the entity relationship model. Further constraints on the entity relationship model, the inheritance diagrams and class interfaces will define that the class will be reflected in lower-level documents as well. The name of $c(sp)$ in the different documents must match the name of the symbol on the left-hand side of sp.

For each symbol com that appears in the component list of sp, the entity relationship diagram that includes $c(sp)$ must define an aggregation relationship. The source class must be $c(sp)$ or one of its super classes. The target class of the aggregation relationship must be the class identified by com. Since we cannot ensure this constraint without the existence of that class in the entity relationship diagram, we require that static semantic constraint **SV1** always holds. The constraint that com must also be reflected in the abstract syntax section of the class interface will be defined indirectly via a constraint on the entity relationship diagram and, therefore, need not be considered here.

Each keyword that appears on the right-hand side of a structure production *sp* has to occur as a keyword in the unparsing section of $c(sp)$. For each symbol s on the right-hand side of *sp*, there has to be a child increment of the respective type in the unparsing section of $c(sp)$. The order of keywords and symbols in *sp* must be respected by the order of the equivalent definitions in the unparsing section of $c(sp)$. Note that other definitions such as pretty-printing items may be included in an unparsing section. Since these do not have any impact on the language they have no counterpart in the ENBNF.

For a structure production *sp*, the class $c(sp)$ has to inherit from the predefined increment class `NonterminalIncrement`. This inheritance relationship has to be reflected in the inheritance diagram.

For each regular expression production *rp* given in an ENBNF, there has to be a terminal increment class $c(rp)$. $c(rp)$ must be reflected in the entity relationship model. The name of the symbol on the left-hand side of *rp* has to be equal to the class name of $c(rp)$. The expression defined in the regular expression section in the class interface of $c(rp)$ has to be equal to the regular expression given on the right-hand side of *rp*.

For a regular expression production *rp* the respective class $c(rp)$ has to be a subclass of the predefined increment class `TerminalIncrement`. Therefore, the inheritance diagram definition has to include the required inheritance declarations.

For each list production *lp* of an ENBNF, there has to be a non-terminal increment class $c(lp)$ in the entity relationship model. The name of $c(lp)$ and the name of the symbol on the left-hand side of *lp* must be equal.

For each list production *lp*, the entity relationship model must define an ordered multi-valued aggregation relationship. The source class must be $c(lp)$ or one of its super classes. The target class has to be the class identified by the name of the symbol given in braces. Again a constraint defined for the entity relationship model will define that the abstract syntax child is also reflected in the interface definition of class $c(lp)$

The multi-valued abstract syntax child in $c(lp)$ has to be referenced from the unparsing scheme. If a delimiter definition is given in round brackets in *lp*, the keywords contained in the definition have to be inserted into the GTSL `DELIMITED BY` clause that follows a multi-valued abstract syntax child in the unparsing section.

The class $c(lp)$ that originates in a list production in general has to be a subclass of the predefined GTSL class `IncrementList`. It is defined as a subclass of `TerminalIncrementList`, if the symbol on the right-hand side of *lp* is declared in a regular expression production. It is defined as a subclass of `NonterminalIncrementList`, if the symbol on the right-hand side is declared in a structure production. The class hierarchy has to be defined accordingly in the inheritance diagram.

For each optional production *op* a class $c(op)$ has to be defined. The name of the symbol on the left-hand side of *op* must match the class name of $c(op)$. The class $c(op)$ has to be a subclass of the predefined increment class `OptionalIncrement`. A respective inheritance declaration, therefore, has to be included in the inheritance diagram. An optional production *op* is either an optional regular expression or an optional structure production. Class $c(op)$ is terminal if *op* is an optional regular expression production and then the constraints for regular expression productions are applied accordingly, otherwise it is non-terminal. The constraints for structure productions must hold for *op* if it has a structure on the right-hand side.

For each alternative production ap, there has to be an abstract increment class $c(ap)$ in the entity relationship model. The name of the symbol on the left-hand side of ap has to be equal to the name of class $c(ap)$.

For each symbol a that appears as an alternative on the right-hand side of ap, a class $c(a)$ is available. This is ensured by the static semantic constraints on the ENBNF and previous constraints. These classes $c(a)$ have to be subclasses of $c(ap)$. This must be reflected in the entity relationship model. A further constraint for the entity relationship model will ensure that the inheritance relationship is also reflected in the inheritance diagram. In general, $c(ap)$ inherits from the predefined increment class `Increment`. It may inherit from a more specific class depending on the subclasses $c(a)$. If they are all subclasses of `NonterminalIncrement`, $c(ap)$ also inherits from `NonterminalIncrement`. If they are all subclasses from `TerminalIncrement`, $c(ap)$ has to be a subclass of `TerminalIncrement` as well. These inheritance relationships need not be reflected in the entity relationship model, but must be defined in the inheritance diagram.

7.2.1.2 Constraint Handling

The ENBNF definition determines the entity relationship model, the inheritance diagram and the class interface definition. Therefore, changes of the syntax will be propagated to these depending views. The inheritance diagram and class specifications are further determined by the entity relationship model and class interfaces, respectively. Therefore, changes will be propagated further to these, if required. We demonstrate what these propagations can achieve by means of an example taken from the Groupie specification. The ENBNF fragment below is taken from the Groupie module interface language definition and will now be used for illustration purposes.

```
Module        ::= ADTModule | FModule | ADOModule | TCModule .
ADTModule     ::= "DATATYPE" "MODULE" ModName ";"
                    Comment
                    "EXPORT" "INTERFACE" "TYPE" TypeName ";" OperationList
                    "END" "EXPORT" "INTERFACE" ImportInterface
                  "END" "MODULE" ModName "." .
OperationList ::= {Operation} .
Operation     ::= Function | Procedure .
Function      ::= <ComponentList> .
Procedure     ::= "PROCEDURE" OpName ParameterList ";" Comment .
ParameterList ::= | "(" {Parameter}(";") ")" .
```

Upon creation of a new ENBNF definition, a new inheritance diagram is created as well. In addition, a new subsystem in the environment's entity relationship model is created. During insertion of a new structure production into the list of ENBNF productions, the name of the left-hand side symbol is determined by the tool builder. After that the production is inserted into the list (like the production with `Function` on the left-hand side above). The ENBNF editor has also triggered change propagations to the entity relationship model. It will further propagate the change to the class interface definitions. The subsystem that corresponds to the ENBNF now includes a new class `Function`.

If an element is inserted into the component list of a structure production, a number of subsequent changes are performed in the unparsing sections and abstract syntax section of the class interface that originated in the production. If the component element is a keyword (like `"PROCEDURE"` above), it is inserted as plain text into the unparsing scheme. If the new component is a symbol (like `OpName` in production `Procedure`), a new aggregation relationship is inserted into the entity relationship model and that, in turn, will trigger the creation of a new child in the abstract syntax section of the interface (of class `Procedure`). In addition, a place holder for a child is inserted into the unparsing scheme at the respective position. If the new element is a list (like {`Parameter`} in production `ParameterList`) a multi-valued aggregation relationship is inserted into the entity relationship model, which will further ensure that an abstract syntax child with a list type is inserted into the abstract syntax section of the class interface. The base type is expanded to the name of the symbol given in braces. Like a symbol, a place holder is inserted into the unparsing section in the right position. Changes to the delimiter parts of the list component are mapped onto respective changes in the `DELIMITED BY` clause of the unparsing scheme. As an example consider the GTSL class interface fragments below, which have (transitively) been generated from the two productions `Procedure` and `ParameterList`.

```
NONTERMINAL INCREMENT INTERFACE Procedure; NONTERMINAL INCREMENT INTERFACE ParameterList;
INHERIT <SuperClassList>;                  INHERIT OptionalIncrement;
EXPORT INTERFACE                           EXPORT INTERFACE
  ABSTRACT SYNTAX                            ABSTRACT SYNTAX
    <Child>:OpName;                            <Child>:LIST OF Parameter;
    <Child>:ParameterList;                   END ABSTRACT SYNTAX;
    <Child>:Comment
  END ABSTRACT SYNTAX;                       UNPARSING SCHEME
                                               "(", <Child> DELIMITED BY ";" END, ")"
  UNPARSING SCHEME                           END UNPARSING SCHEME;
   "PROCEDURE", <Child>, <Child>, ";",     ...
      <Child>
  END UNPARSING SCHEME;...
```

We have only considered those changes above that, due to increment creation, were propagated to the entity relationship model and indirectly into class interface definitions. We also have to consider the inheritance diagram and define propagations for increment creation operations. Moreover, we have to consider operations that change or delete increments and define change propagations. This is done in a very straight-forward manner and, therefore, not considered here further.

7.2.2 Entity Relationship View Editor

The entity relationship model determines inheritance diagrams and class interfaces in the sense that any change to a class in the entity relationship model must be reflected in the class' interface definition and might also have to be reflected in the inheritance diagram.

7.2.2.1 Consistency Constraints

Each diagram er that refines a subsystem of the top-level entity relationship diagram is associated with an inheritance diagram. For each class erc contained in er or its nested subsystems, there must be a tool specific increment class $c(erc)$ in the inheritance diagram. The names of erc and $c(erc)$ have to be equal. For each inheritance relationship in the entity relationship model that defines erc_i as subclass of erc_j, $c(erc_i)$ has to be a subclass of $c(erc_j)$ in the inheritance diagram. Note that we do not require the reverse, i.e. there may be inheritance relationships that are not reflected in the entity relationship model so as to decrease the complexity. Only those relationships should be reflected that contribute to the inheritance of relationships and thus impact navigation paths. The inheritance diagram serves as a complete overview of the inheritance hierarchy.

For each aggregation relationship ars that defines erc_j as a component of erc_i, there must be an abstract syntax child e in class $c(erc_i)$ The child name must be equal to the name of ars in the entity relationship model. With regard to the type of e, we have to distinguish single-valued and multi-valued relationships. If ars is single-valued, the type identifier of e must be equal to the name of $c(erc_j)$. If ars is multi-valued, the type of e must have been constructed with the list type constructor applied to $c(erc_j)$. Vice versa, for each abstract syntax child e of class $c(erc_i)$ there must be an aggregation relationship ars starting from erc_i. If e is single-valued, ars must be single-valued as well and lead to the class in the entity relationship model that corresponds to $type(e)$. If e is multi-valued, ars must be multi-valued, ordered and lead to the class that corresponds to the base type of $type(e)$.

For each reference relationship rrs that refers in the entity relationship model from class erc_i to erc_j there has to be a semantic relationship between $c(erc_i)$ and $c(erc_j)$. The first name n_1 of rrs determines the explicit link e of the corresponding semantic relationship. The name of e must be n_1. The type of e depends on the cardinality of rrs. If rrs is single-valued, the type of e must be $c(erc_j)$, otherwise the type is a set type with $c(erc_j)$ as base type. If a second name n_2 is given for rrs, we assume that the tool builder needs the implicit link for the relationship and require that there is an implicit link i defined in the semantic relationship section of $c(erc_j)$ whose name equals n_2. The type of i must be $c(erc_i)$. Vice versa, for each explicit link defined for class erc_i, there must be a reference relationship to the class in the entity relationship model that corresponds to the type or base type of the link. The name of the explicit link must be equal to the first name of the reference relationship. If the explicit link has a corresponding implicit link, the name of the implicit link must match the second name of the relationship.

7.2.2.2 Constraint Handling

The entity relationship editor supports the creation and deletion of new subsystems and the various kinds of reference relationships. It supports the creation of abstract classes in order to define common properties of several subclasses. It then supports the introduction and deletion of inheritance relationships. The definition of properties that are common to all subclasses of an abstract class may be moved to the abstract class. We consider this as property generalisation. The reverse command, i.e. property specialisation, transfers the property definition to all subclasses that are leaves in the inheritance hierarchy. Moreover, subsystem and relationship names can be expanded and changed. The editor supports navigation through the decomposition hierarchy in terms of zoom-in/zoom-out commands. These navigation commands are offered for subsystems. The graphical layout of the diagram can be modified and, in particular, classes can be moved to lower- or upper-level subsystems. These operations will automatically insert ports as required by the static semantic constraints.

An initial diagram that represents a tool in the entity relationship model of an environment is created together with the ENBNF definition of the tool. Change propagations of the ENBNF editor ensure that the entity relationship model is kept consistent with the ENBNF definition. Note that terminal and non-terminal classes cannot be created or deleted in the entity relationship editor. Also aggregation relationships cannot be created or deleted, they can only be generalised or specialised, which does not impact the abstract syntax.

The entity relationship editor performs a number of change propagations to preserve inter-document consistency constraints to inheritance diagrams and class interfaces. The tool builder can change names of aggregation relationships in the entity relationship editor. These changes are propagated to the abstract syntax sections of the target class of the changed relationship in order to change the child name accordingly. In addition, the tool builder can create new reference relationships and modify or delete existing ones. These commands require propagations to semantic relationship sections of class interfaces. If a new reference relationship is created, a new explicit link is inserted into the semantic relationship section of the class where the reference relationship starts. Expanding or changing the first name of the reference relationship expands or changes the name of the explicit link. Expanding the second name creates a new implicit link in the semantic relationship section of the target class of the reference relationship. Deleting a reference relationship triggers deletion of the corresponding explicit and implicit links. Finally, the tool builder can create, change or delete attributes of a class. These commands trigger the changes to be made in the attribute section of respective class interfaces.

The introduction of new inheritance relationships are propagated from the entity relationship model to inheritance diagrams. Similarly deletion of inheritance relationships in the entity relationship model triggers a propagation that deletes the relationship from the inheritance diagram as well.

We have defined above an editing command that allows for generalisation of properties. The generalisation command can move the definition of a property p that is defined for all subclasses of an abstract class erc from the subclasses $erc_1, \ldots, erc_n$ to erc. Without any further action, this would reveal inconsistencies in class interfaces. Therefore, the declarations of p in $c(erc_1), \ldots, c(erc_n)$ are removed and inserted into $c(erc)$. In particular, this command simplifies the actions that are required for the generalisation of properties. Likewise, specialisation is handled in such a way that definitions are moved from the abstract class to its non-terminal and terminal subclasses.

Upon creation of a new abstract class, this class is inserted into the inheritance diagram during a change propagation. Creation or deletion of inheritance relationships causes change propagations that insert or delete the inheritance hierarchy from the inheritance diagram.

7.2.3 Inheritance Diagram Editor

An initial inheritance hierarchy for increment classes is established by change propagations from the entity relationship editor. This inheritance hierarchy might be edited by further introducing new abstract increment classes and redefining the inheritance relationships of generated increment classes. Creation of further non-terminal or terminal increment classes or deletion of classes that were contained in the initial inheritance hierarchy is not supported. The entity relationship model does not include non-syntactic classes that are used to define attribute types, since these do not contribute to the definition of navigation paths. Non-syntactic classes can, therefore, be created with the inheritance diagram editor as subclasses of predefined non-syntactic classes. Since these editing commands are offered, further inter-document consistency constraints have to be defined.

7.2.3.1 Consistency Constraints

Each class c in the inheritance diagram that is not predefined must be refined by a class $c(c)$ with an interface and a specification. If the class is an abstract increment class (it has subclasses), it must have an abstract increment class interface. If it does not have subclasses and does not define or inherit outgoing aggregation relationships, $c(c)$ must be a terminal class. If it does not have subclasses, but defines or inherits outgoing aggregation relationships, it must be defined by a non-syntactic class. If it is a subclass of a predefined non-syntactic class, it must be refined by a non-syntactic class.

The inheritance relationships defined in the inheritance diagram must be reflected in the inheritance sections of the class interface definitions. Therefore, for each inheritance relationship that defines class c_i as a subclass of c_j, there must be an entry in the super class list of the inheritance section of $c(c_i)$ that defines the name of $c(c_j)$.

7.2.3.2 Constraint Handling

For an abstract increment class of a non-syntactic class c that is created in the inheritance diagram, a class interface definition $c(c)$ has to be defined. The interface will be an abstract increment interface if the class in the inheritance diagram was an increment class and it will be a non-syntactic interface otherwise. The creation of this interface will further cause a propagation by the class interface editor that also creates a class specification document. Likewise, if a class ac is deleted, the class interface $c(ac)$ will also be deleted. This deletion will, as a follow-on propagation, also delete the respective class specification.

Creation or deletion of an inheritance relationship that defines c_i as subclass of c_j will change the inheritance section of $c(c_i)$. If a new relationship is created in the inheritance diagram, the name of c_j will be added in the inheritance section of c_i. In the case of a deletion, the name of c_j will be deleted from the inheritance section of c_i.

7.2.4 Class Interface and Specification Editors

Class interfaces define the external behaviour of a class. This external definition is then refined in the internal class specification. While the inter-document constraints we discussed above were always between different types, we now have inter- as well as intra-type consistency constraints. The import interfaces and inheritance sections in class interfaces impose intra-type inter-document consistency constraints on other class interface definitions. Inter-type consistency constraints require, for instance, each property or method imported by a class specification to be exported by the respective interface or the explicit methods defined in the interface of a class to be specified in the method section of the specification.

For reasons of brevity, we cannot address the constraints in full detail here. They are formally defined in Appendix A. We, therefore, restrict ourselves to discussing the way the interface editor propagates changes to other class interfaces in order to deal with intra-type constraints and to other specifications in order to deal with inter-type constraints.

The existence of a class specification depends on the existence of the interface of a class. The class interface editor, therefore, propagates the creation of a class to the class specification editor in order to create a class there as well. When a class interface is deleted, the class specification must also be deleted.

Initial class names are determined with the entity relationship and inheritance editors. If names are changed there, they will propagate the change to the interface view. The interface editor will, in turn, propagate the change further to

- all inheritance sections of subclasses,
- all import interfaces of class interface definitions that imported the class, and
- all import interfaces of class specification that imported the class.

The interfaces, in turn, will propagate the change further to property types, local variable types, method parameter and result types where the class might be used as a type name.

If the entity relationship editor propagates the change of a property name to the interface editor, the change must be propagated further to all interfaces of subclasses that inherit the property. In addition, all specifications of subclasses that use the name in semantic rules, methods or interactions must be informed so as to change the property name there as well. The name may in addition have been imported in client class specification. Then the name change must be propagated to those imports from where it is further propagated to the places where it is used.

Any explicit method of a class interface must have a body defined in the specification of the class. Upon creation of a method in the interface a place holder is created in the specification as well. Any change to an explicit method such as creation, change or deletion of a parameter, changes to the result type are propagated to the respective method in the class specification.

7.3 Summary

We have defined a number of inter-document consistency constraints between the different document types that were identified for the specification of tools. The constraints are defined between a determining and a dependent document type. They define paths for propagating changes from a determining into a dependent document. The different paths are summarised in Figure 7.3.

Each rectangle represents a document type and an arrow represents a propagation path from a determining document type to a dependent type. The annotations of arrows denote the subject of a propagation. Classes and aggregation relationships are created by the ENBNF editor in the entity relationship model. Furthermore, it creates the inheritance relationships between increment classes and predefined classes in the inheritance diagram. Finally, it directly creates regular expressions and unparsing scheme definitions in the class interface. The entity relationship editor, in turn, propagates the definition of classes to the inheritance diagram editor. Moreover, it creates properties, i.e. attributes, abstract syntax children and semantic relationships in the respective class interface. The inheritance diagram editor generates class interfaces and also determines their inheritance sections. Finally, changes to property and method definitions are propagated from the interface editor to the respective class specifications. In this way a tool builder can incrementally define a tool specification at different levels of abstraction; the tool builder can choose the appropriate level of abstraction for the particular concern that has to be defined. While working on a document at a higher level of abstraction, fragments for tool

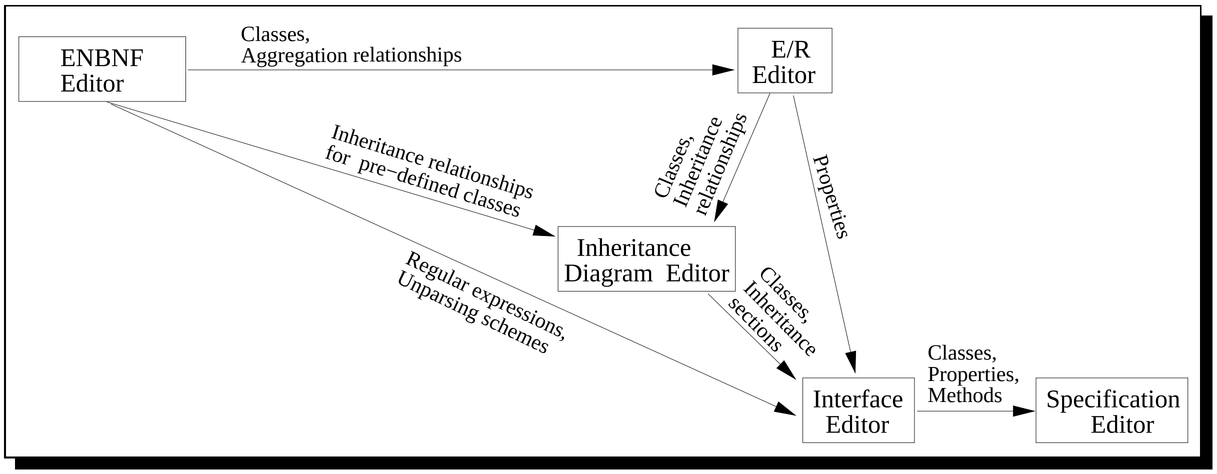

ENBNF Editor
Classes, Aggregation relationships
E/R Editor
Inheritance relationships for pre-defined classes
Classes, Inheritance relationships
Properties
Inheritance Diagram Editor
Regular expressions, Unparsing schemes
Classes, Inheritance sections
Interface Editor
Classes, Properties, Methods
Specification Editor

Figure 7.3: Propagation Paths

specifications are created, modified or deleted by the environment in lower-level documents.

It is our strong belief that the definition of our tool specification environment, as exemplified here, can be generalised to environments for many software processes. Any software process model will include documents at different levels of abstraction. High-level documents are, for instance, data flow diagrams, petri nets or informal requirements definitions. At lower levels, documents such as architecture diagrams and component interface definitions will be defined and at even lower levels programming languages will be defined that are used to implement components. These documents will be written in a formal language of some sort. They will always include redundant information. To cope with this redundancy, propagation paths should be defined from the higher levels of abstraction into the lower levels. At lower levels, violations of inter-document consistencies should be visualised in order to guide users so as to enable them to produce statically correct software. The implementation of these inter-document consistency constraints and the propagation paths at the appropriate level of abstraction, in turn, will be supported by the GOODSTEP tool specification languages and the GENESIS environment.

Chapter 8

The GTSL Compiler

In this chapter, we discuss the design and implementation of the compiler that is used to translate class interface and specification definitions into executable tools. A number of interesting problems arise during construction of this compiler. Most of them are related to the question of how the various tool-specific architecture components, which have been identified in Chapter 5, can be derived from a GTSL specification. In the next chapter, we then evaluate GTSL by constructing a number of tools. In particular, we will discuss the specification of tools for the GENESIS environment that were constructed with a *bootstrap* approach.

A principle in the construction of the GTSL compiler was to support independent compilation of class interface and specification definitions. Only this approach enables incremental and cooperative development of tools to be effected. If all class definitions were stored in a single file, only one developer could edit or compile it at the same time and cooperation would be seriously hampered. Each class interface and specification is, therefore, compiled independently. As there are obvious dependencies between interfaces and specifications, it is necessary to write a specification that controls the order in which different sources are compiled. This could be done in terms of a make-file, but we consider this as too low a level of abstraction. Instead, the compiler should use information, defined in the inheritance and entity relationship diagrams about classes and relationships, to determine a valid compilation order. Here we run into a dilemma because we need tools of the GENESIS environment to be able to construct the tools.

We solve the dilemma by introducing an intermediate tool specification document. We refer to this document as *tool configuration*. It will determine the different increment and non-syntactic classes, their inheritance relationships, the root increment class and the error messages that are to be associated with error descriptors. Tool configurations thus serve the same purpose as system files that are required to compile Eiffel programs. When the other tools are finished, the configuration document will be generated from the ENBNF definition, inheritance and entity relationship diagram. For the time being, however, the tool builder will have to write it manually. The complete syntax and static semantics is defined in Appendix A. An example of a configuration definition is given below. It displays an excerpt of the Groupie module interface editor configuration.

```
CONFIGURATION MIE
  CONSISTS OF
    INCREMENT CLASSES
      Commentable INHERIT Increment;
      Module INHERIT DocumentVersion, Commentable;
      ADTModule INHERIT Module;
      ADOModule INHERIT Module;
      ...
    END INCREMENT CLASSES;
    ROOT INCREMENT IS Module
    ADDITIONAL ERRORS
      #NameAlreadyDefined : "The given name has already been defined";
      #NotATypeName: "The Import is not an exported type";
      #NotAnOpName: "The Import is not an exported operation";
      ...
    END ADDITIONAL ERRORS
END CONFIGURATION MIE.
```

As shown in Figure 8.1 the GTSL compiler consists of five components: A tool configuration compiler (`conf`), a class interface compiler (`int`), a class specification compiler (`spec`), a linker (`link`) and a controller (`genesis`) that serves as a user interface for the previous components. The compilers store symbol tables and generated code in the file-system.

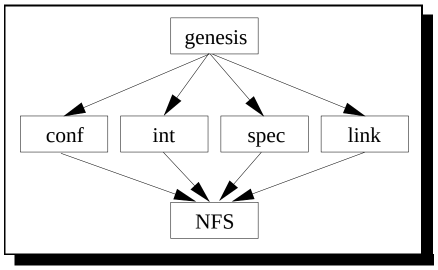


Figure 8.1: Components of the GTSL Compiler

The main purpose of the tool configuration compiler is to generate various build definitions. `Conf` generates a make-file from a tool specification so that `make` [Fel79] can be used for controlling any further invocations of the GTSL compiler. In particular, this make-file ensures that the interface of a class is translated before its specification and that a class interface is only compiled when all interfaces of super classes have already been compiled. In addition, `conf` generates make files for compiling the O_2 database schema with the O_2C compiler, exporting the schema to C++ in order to create the `ToolAPI` subsystem, compiling all tool-specific components of the tool architecture and linking the result with the library of reusable tool components.

The purpose of the `int` compiler is to compile interface definitions of single GTSL classes. The main result of such a compilation is a symbol table that is stored in the file-system. That table is loaded during compilations of dependent class interfaces (e.g. interfaces of subclasses to check for correct inheritance) or specifications (e.g. to check whether imported entities or methods are exported from the other class). Moreover, symbol tables serve as the source of information during code generation, which is performed in the specification compiler.

The `spec` compiler checks class specifications against their static semantics and validates inter-document consistency constraints to class interfaces. It then generates code fragments for the various tool-specific architecture components. The layout computation subsystem is derived from the unparsing schemes. The required information is obtained from the class' symbol table because unparsing schemes have already been compiled by the interface compiler. Parts for the command execution subsystem are generated from interactions. Components of a rule interpreter that incrementally evaluates semantic rules are generated from semantic rules. The O_2 class interface definitions stem from property and method definitions given in the class interface. Implicit methods are generated from the various interface sections, and bodies of explicit methods are generated from the method sections given in class specifications. The code, however, must remain in fragments because the generation of the semantic rule interpreter requires all property accesses to be known. This is only the case after all increment specifications have been compiled. Then the linker binds the fragments together and thus completes code generation.

Each of the three compilers is fully-fledged, that is it has a scanner, a parser, a semantic analyser and a code generator. Scanner, parser and semantic analyser are called *front end* of the compiler in the following paragraphs. The code generator is called *back end*. For the construction of these compilers, we use the compiler construction tool kit Eli [GHL+92]. We have chosen Eli rather than lex and yacc, or flex and bison because Eli not only supports the generation of scanners and parsers, but also the generation of static analysers and code generators. As the generation of scanners and parsers is done by Eli based on standard techniques, we do not address it any further in this thesis. The next subsection about the front end, therefore, focuses on the generation of static analysers for the three compilers. The second subsection then addresses how the different tool-specific components are generated. During that, we focus on an algorithm for the incremental evaluation of GTSL semantic rules.

8.1 Front End

One could argue that there is no need to check static semantics in the GTSL compiler. The GTSL specifications to be compiled are error free since they will be edited with the GENESIS environment and its tools check for static semantics and inter-document consistency whenever a tool builder performs a change. There are, however, a number of reasons that made us implement static semantic analysis in the compiler. Firstly, the specifications for the GENESIS tools have to be edited manually with conventional text editors, because we follow a bootstrap approach and GENESIS is not yet available. GTSL specifications have a considerable complexity. To ensure their correctness manually is very difficult, if not impossible. If the GTSL compiler checks static semantics and inter-document consistency this will give us considerable support. Secondly, during the specification of the GENESIS tools, we gained experience on the use of GTSL. This included experience on the appropriateness of static semantic constraints. In fact, some of these insights made us change the initial language definition, and the definition given in Appendix A is the result of an iterative language definition process. Without a compiler to implement those changes it would be even harder to ensure the consistency of GENESIS tool specifications after a change in the language. Finally, the code generation that has to be implemented in the compiler requires data that must partly be gathered during static semantic analysis.

The different static semantic properties are defined formally in Appendix A and can be classified into *scoping* and *typing* rules. The scoping rules are concerned with uniqueness of identifiers and whether a using application of an identifier matches some declaration. They, therefore, implement the mapping *scope* and the various mappings *name* that are defined in Appendix A. The typing rules are concerned with the properties of identifiers and thus implement the various mappings *type* and *kind* given in the Appendix. On the basis of these implementations, correctness of property and method redefinition, multiple inheritance and the polymorphism rule are checked. For reasons of brevity, we cannot discuss the implementation of all static semantic constraints, but simply present some archetypical examples from these two categories. These examples then demonstrate how we exploit various basic mechanisms for the implementation of static semantic analysis that are offered by Eli.

The basis for defining static semantic analysis is the availability of ordered attribute grammars [Kas80]. We have already introduced them in Subsection 6.6.1, when we discussed the Synthesizer Specification Language. Eli offers a dedicated language for defining attributes, attaching them to symbols and defining semantic functions based on a number of predefined operators. In addition to this language, Eli offers a number of higher-level concepts such as an identifier table, a module for generating error messages, a mechanism for definition tables and a property definition language. It also defines an interface to C so that we can define dedicated operators in cases where the predefined ones prove to be insufficient. We now discuss how we use these mechanisms in order to implement static semantic checks of GTSL.

Whenever a terminal symbol is scanned, the scanner can invoke a processor that stores the value of the symbol in an *identifier table* and returns the address of the value. This address is also called a *symbol*. It can be used during static semantic analysis to obtain the value. Therefore, symbols are usually stored as attributes of syntax tree nodes. The example below is taken from the tool configuration compiler and illustrates this:

```
ATTR Sym: int;

RULE rule_009 : incr_class_id ::= IDENTIFIER
COMPUTE
  incr_class_id.Sym = IDENTIFIER.Sym;
END;
```

The first line declares that all nodes in the syntax tree have an attribute `Sym` of type `int`. The next four lines declare a rule that determines how the value of this attribute is computed for the name of an increment class. The head of the rule defines a rule name and declares the production after the colon. Between the keywords `COMPUTE` and `END` a number of semantic functions can be invoked and their results can be assigned to attributes. Here the identity function is used and the value of the `Sym` attribute of node `IDENTIFIER`, which, in turn, was set by the scanner, is assigned to attribute `Sym` of node `incr_class_id`. Attribute `Sym` is then used to implement scoping rules based on Eli's mechanism for definition tables.

A *definition table* consists of a tree of scopes. The root scope is created by the operator `NewEnv`. The result of a call to this operator is usually stored in a node attribute. A sub-scope may be added by the operator `NewScope`, which takes an existing scope as argument and returns a new sub-scope. A number of operators are available to update and query scopes. Operator `DefineIdn` declares an identifier that is identified by a `Sym` attribute in a scope. `KeyInScope` performs a lookup and checks whether or not an identifier has been defined in a scope. `KeyInEnv` not only queries the current scope, but also all its ancestor scopes. The example below demonstrates how class identifiers are entered into a scope and checked for uniqueness.

```
ATTR ActEnv : Environment;

RULE rule_001 : configuration ::=  'CONFIGURATION' conf_id 'CONSISTS' 'OF'
        conf_opt_import_part
        conf_inc_part
        conf_opt_att_part
        conf_root_part
        conf_opt_export_part
        conf_opt_additional_errors
        'END' 'CONFIGURATION' doc_using_id '.'
COMPUTE
        ActEnv = NewEnv();
END;

SYMBOL incr_class_id : Done : Void;
SYMBOL incr_class_id : Key : DefTableKey;
RULE rule_009 : incr_class_id ::= IDENTIFIER
COMPUTE
  incr_class_id.ActEnv = INCLUDING configuration.ActEnv;
  incr_class_id.Done=IF(EQ(KeyInScope(incr_class_id.ActEnv, incr_class_id.Sym), NoKey),
                        DefineIdn(incr_class_id.ActEnv, incr_class_id.Sym),
                        message(ERROR, "Identifier already defined",
                                0, COORDREF));
  incr_class_id.Key=KeyInScope(incr_class_id.ActEnv, class_id.Sym)
                      DEPENDS_ON incr_class_id.Done;
END;
```

The computation defined in `rule_001` creates a new root scope and stores it in attribute `ActEnv` of the root node of the configuration syntax tree. In `rule_009`, the `ActEnv` attribute for an increment class identifier is determined by fetching the `ActEnv` attribute of the configuration. This is defined by the `INCLUDING` declaration that traverses the syntax tree to the root until it obtains a node of type `configuration`. The next function performs a lookup in scope `ActEnv` with the symbol of the identifier and checks whether that lookup returns the invalid key `NoKey`, in which case the symbol has not yet been defined. If so, it defines the symbol using operator `DefineIdn`, otherwise it uses operator `message` to print an error message. The first argument of `message` is the severeness of the error, ranging from warnings to fatal errors. The second argument is the error message and the last argument is an operator that computes the error coordinates in the input file. Note that operator `DefineIdn` takes the symbol of the class identifier as an argument. Thus execution of the function depends on the assignment of the symbol that was explained above. The attribute evaluator that is generated by Eli ensures that these dependencies are respected without further explicit measures. It orders the evaluation in such a way that first the assignment is performed and then the attribute is used. Upon completion of the second function it assigns some value to the attribute `Done`. Only then is the third function executed because it is declared dependent on attribute `Done`. That function performs a lookup of the key and assigns it to attribute `Key` for later use during property definition.

Eli supports the definition of properties for entries of a definition table. These properties are then used during implementation of typing rules. Eli includes a property definition language (PDL). A property has a name and a type. Eli generates for each property an update and a query operator. The name of the update operator is the name of the property prefixed with `Set`. It takes a key of a definition table entry as the first argument. If the property has been determined for this key, its value is replaced by the second argument. Otherwise, the third argument is taken as the new value. The query operator's name is the name of the property prefixed by `Get`. The query operator has a key as first argument and a default property value as second argument. If the property has been determined for the key it returns its property value, otherwise it returns the default. PDL is exploited in the GTSL compiler implementation for the definition of the various properties of classes, methods, attributes, syntax children and links that are given in Appendix A. As an example consider the property of being an increment or non-syntactic class used in the configuration compiler.

```
typedef enum {
        IsNonSyntacticClass,
        IsIncrementClass,
        IsDocName,
        IsRoot,
        IsError,
        Undefined} TypeOfDef;

Kind : TypeOfDef;
```

The first part declares an enumeration type `TypeOfDef`. It enumerates the different roles that a declaration in the tool configuration can play. Then an Eli property `Kind` is defined. The type of `Kind` is declared as `TypeOfDef`. This property is then used, for instance, to implement Condition A.6, given in the appendix on Page 264. It ensures that the root class in a configuration is an increment class:

```
RULE rule_009 :
  incr_class_id ::= IDENTIFIER
COMPUTE
  SetKind(incr_class_id.Key, IsIncrement, IsIncrement);
END;

RULE rule_015 :
  root_part ::=  'ROOT' 'INCREMENT' 'IS' root_id
COMPUTE
        root_id.Key=KeyInScope(INCLUDING configuration.ActEnv, root_id.Sym)
        IF (NE(GetKind(root_id.Key, Undefined), IsIncrement),
            message(ERROR, "Root-identifier must be an increment",
                    0, COORDREF));
END;
```

Rule `rule_009` determines the value of property `Kind` for the definition table entry of an increment class identifier to be `IsIncrement`. Rule `rule_015` then queries the definition table with `KeyInScope` to obtain a definition table key of an identifier that declares the value associated with `root_id.Sym`. The result can be a key that corresponds to an arbitrary declaration of an identifier. To ensure that it corresponds to an increment class, the key is used in a further query that obtains the value of property `Kind`. If it is not equal to `IsIncrement`, an error message is released.

By using the mechanisms presented above most GTSL static semantic constraints can be implemented. The implementation of some constraints, however, requires operators that are not available in Eli. As an example, consider the polymorphism rule or covariant method redefinition. These require an operator that checks whether some identifier denotes a class that is a subclass of a class denoted by some other identifier. As this operator is highly specific, it is obviously not available in Eli. It has, therefore, to be implemented in a programming language, i.e. C or C++ using Eli's C-Interface. This interface, in particular, offers all operators that are available in Eli's semantic function language as C-functions and provides for access to the identifier table. The example below demonstrates how an operator implemented in C is used in a semantic function to check the polymorphism rule:

```
RULE rule_083 :
  assignment ::=  ass_id2 ASSIGN attribute_creation
COMPUTE
  IF(NE(SubtypeCheck(attribute_creation.actual_type_sym,
                     ass_id2.actual_type_sym), TRUE),
     message(ERROR, "types are not compatible", 0,COORDREF)),
END;
```

The example is taken from the specification compiler and implements the check of the polymorphism rule for the creation of a new instance of an attribute. The operator `SubtypeCheck` returns `TRUE` if the symbol passed as first argument is the name of a subclass of the class identified by the symbol passed as second argument. If this is not the case, the assignment is not correct according to the polymorphism rule. In that case an error message is released. `SubtypeCheck`, in turn, is implemented in C based on the symbol table that is written by the configuration compiler. This symbol table contains all information about the class hierarchy and is loaded by the specification compiler upon start-up.

While it is fully acceptable to implement specific operators such as `SubtypeCheck` in a programming language, Eli could have provided better support for the general problem of implementing independent compilation of source files. Persistence of definition tables is required for independent compilation so as to perform inter-document consistency checks. The definitions that have been made in one class have to be stored persistently in order to check for the correctness of their use during compilation of other classes. Eli does not provide built-in facilities for that. Upon successful compilation of a class we had, therefore, to iterate over the definition table and dump relevant entries into a file stored in the file-system. This file then had to be loaded when processing an import statement and its definitions had to be entered into the definition table of the other class. Thus we were forced to implement inter-document consistency checks at a rather low level of abstraction. A further problem, which makes it difficult to change the language definition, is that both parser generator and static analyser generator require the context free grammar definition as input. Unfortunately, the input format is different and the compiler builder has to maintain redundant grammar definitions. We would find it more appropriate if the parser generator derived the grammar from the definition given in the input for the static analysis generator, for example. We see the trade-off that defining an attribute grammar is more complex than defining a context-free grammar and this overhead is only bearable if a static semantic analyser is generated.

8.2 Back End

The purpose of the back end of any GTSL compiler is to generate code in order to contribute to the generation of the tool-specific architecture components. The most important and complex component is the database schema. It implements the structure and behaviour of the abstract syntax graphs that represent documents. Since we use object databases, there is no large conceptual gap between properties and methods that have been defined in GTSL and instance variables and methods to be generated in the target schema definition language O_2C. O_2C instance variables and methods can basically be generated from their GTSL counterpart in a one-by-one fashion. A large conceptual gap that we have to bridge, however, exists between the specification of static semantics and inter-document consistency constraints in semantic rules and their implementation in terms of O_2C instance variables and methods. The first subsection, therefore, presents an incremental evaluation algorithm for semantic rules and discusses how the input for this algorithm is derived from GTSL semantic rules. The second subsection briefly discusses the different techniques that are used to generate the tool-specific architecture components.

8.2.1 Incremental Evaluation of Semantic Rules

Semantic rules are very similar to database triggers. It is, therefore, worthwhile to see whether triggers can be used for the implementation of semantic rules. Triggers are among the extensions that were developed in the GOODSTEP project for the O_2 ODBS [CCS94]. A trigger in O_2 consists of an event, a condition and an action. Among the available events are creation or deletion of objects or attribute changes. A condition is an OQL query and an action is an O_2C method body. To implement a semantic rule, we would then translate the rule condition into an event and a condition of a trigger and translate the body of the rule into an O_2C method body. An example is given below. This trigger implements the second semantic rule of class `ADTModule` that was discussed on Page 154.

```
create rule ADTModule_Rule1
coupling (deferred)
on update OpName->value with delta_OpNames
if ( select opns into OP
     from opns in delta_OpNames )
do {
 o2 ADTModule __self;
 for (opn in OP) do {
  __self=((o2 ADTModule) opn->father->father->father);
  __self->DefinedNames->associate(op->name,op->name->value);
 }
}
```

It defines that whenever the `value` attribute of an `OpName` is modified during a transaction, the `OpName` is inserted into a set of delta objects. The deferred coupling mode defines that the trigger is fired upon successful completion of a transaction. Before finally committing the transaction, the `do` part is executed. It inverts the path expression in the `ON` part of the semantic rule for each element of the set in order to obtain the module to which the operation name belongs. It then invokes an `associate` operation and thus inserts the value of the operation name into the module's symbol table.

The approach looks very promising at first glance. There are, however, several problems that force us to reject it. The first problem is that with deferred coupling, we cannot access the results of the trigger execution within a transaction. This is required, for instance, to provide the user of a tool with an error message if some static semantic constraint is violated. To achieve this, the semantic rule must have been executed within the transaction that implements the command. After that, the user might decide to undo the command and then the transaction must be aborted. Therefore, deferred coupling cannot be used. The trigger mechanism of O_2 offers a second coupling mode. With immediate coupling, the trigger is fired, as soon as the event occurs. Immediate coupling is weak, if we consider efficient tool execution. As an example, consider parsing a document. If all triggers that depend on a symbol table fire immediately whenever, for example, a new identifier is entered during parsing, they fire much too often and the tools' performance will suffer from this inefficiency. They should only fire after the last identifier has been inserted. The reason for this inefficiency is that the trigger execution subsystem in O_2 is not aware of the dependencies between different semantic rules. What is required is a combination of immediate and deferred coupling, which in addition exploits knowledge on the dependencies between the different semantic rules. This knowledge can be acquired during static semantic analysis of rules.

We, therefore, develop an algorithm for *incremental evaluation of semantic rules.* It is a distributed adaption of the two-phase, lazy attribute evaluation algorithm for graphs presented in [Hud87]. The two phases are a *propagation phase* where dependent rules and attributes are informed about a change and marked as *dirty*, and an *evaluation phase* where dirty rules are executed and brought back into a clean state. The distinction between these two phases contributes to the overall efficiency of the semantic rule evaluation. They are efficient because only a small excerpt of the documents that have been produced in a project are displayed in some user's editor. Most of them reside only in the database. Only semantic rules contained in these displayed documents have to be reevaluated immediately after they have been marked as dirty. Dirty rules of increments that are not displayed only need to be reevaluated when the increment is accessed the next time. It could well be that the rule receives further propagations before it is displayed. Thus we save time for two reasons. Firstly, subsequent propagations can terminate at the dirty increment because all transitively dependent rules were marked dirty the first time. Secondly, the rule is only evaluated when really required and evaluations between successive propagations are omitted.

The original algorithm has been adjusted for attribute evaluation in the PROGRESS system [Sch91b]. PROGRESS uses a global scheduling graph storing all information about static dependencies between attributes and semantic rules. Instead of a global scheduling graph we use *distributed* scheduling information in combination with *message passing* in a locally controlled algorithm. The main benefit of our approach is that conflicting updates of global scheduling information by concurrent changes of the ASG are avoided.

Semantic rules have to consider those properties that carry semantic information, namely, attributes and links of semantic relationships. For the rest of this chapter, we refer to attributes or links as *semantic variables.* To keep track of the dependencies between different semantic rules and to allow for incremental evaluation, we have to consider and maintain *scheduling information.* Thus, the main idea of the algorithm described below is to distribute the information about static dependencies by assigning local *static* scheduling information to each semantic variable. Furthermore, *dynamic* scheduling information is maintained by each increment to describe its state of evaluation at run-time. Locally controlled variable evaluation is synchronised by exchanging messages between affected increments.

The local static scheduling information is inferred by the `spec` compiler as a set of *propagation information items.* Each item consists of a *path expression* and a unique *propagation identifier.* The path expression determines the set of increments that contain rules and semantic variables that have to be stamped dirty. These are exactly those rules that have declared a dependency in their `ON` condition to the variable that is becoming dirty. The path expression for a semantic variable v is defined by inverting the path expression of the condition of the semantic rule, which contains the semantic variable v as the last name[1]. The propagation identifier in the propagation information identifies the particular semantic rule which has to be stamped dirty.

[1]For a detailed definition of the generation of path expressions for propagation informations we refer to [Jah94].

We assume that each semantic rule is identified by a unique propagation identifier. These identifiers are computed during static semantic analysis. If a semantic rule's condition contains an `OR` expression it may be identified by two or more propagation identifiers, i.e. a set of propagation identifiers may exist for one semantic rule sr which we call $pid(sr)$.

The inversion of path expressions required for this algorithm is only possible since we were able to exclude method invocations from preconditions of semantic rules. We would not have been able to invert path expressions like those in object-oriented programming languages because they may include method invocations. These methods may compute non-injective functions and, therefore, cannot be inverted in general. Instead, we again exploit domain-specific restrictions here to be able to provide a simpler and more efficient implementation.

Let $PATH$ be the set of all possible path expressions. Furthermore, we define V_c as the set of all semantic variables that are declared in an increment class c. The static scheduling information SI_v^{st} that is assigned to a semantic variable $v \in V_c$ is then defined as

$$SI_v^{st} \subset PATH \times I\!N$$

In order to define the dynamic scheduling information we define SR_c as the set of semantic rules which are declared in class c or are inherited from a super class. Furthermore, let I_t denote the set of all existing increments at time t. The dynamic scheduling information of an increment of class c includes the state marks of each semantic variable ($statv$) and each semantic rule ($statsr$) defined in c. Furthermore, a set ($prems$) is maintained for each semantic rule sr of a class c containing all increments that have sent a change propagation identifying sr after the most recent execution of sr. We define the dynamic scheduling information of an increment $i \in I_t$ of class c at time t as a three-tuple:

$$\begin{aligned} SI_i^{dy}(t) = \langle statv, statsr, prems\rangle \text{ with}& \\ statv : V_c \rightarrow \{ \text{'clean', 'dirty'} \}& \\ statsr : SR_c \rightarrow \{ \text{'clean', 'executing', 'dirty'} \}& \\ prems : SR_c \rightarrow \mathcal{P}(I_t)& \end{aligned}$$

Each increment has a predefined method `propagate`. In the implementation it will be inherited from the predefined class `Increment`. During the propagation phase this method is invoked to stamp semantic variables and semantic rules as dirty according to the propagation information of a semantic variable in the invoking class.

For the definition of the algorithm performed by method `propagate` in Figure 8.2 the following definitions are required:

- Let C_s be the set of all increment classes in a specification s.
- We define a function ($readvars_c$) that returns the set of semantic variables for a semantic rule that are read by the rule, i.e. the set of semantic variables that appear in the condition of the rule.

  $$\begin{aligned} &readvars_c : SR_c \rightarrow \mathcal{P}(\textstyle\bigcup_{d \in C_s} V_d) \\ &\quad v \in readvars_c(sr) \Leftrightarrow v \text{ is read by } sr \in SR_c \end{aligned}$$
- $modvars_c$ is a function that returns the set of semantic variables for a rule in class c, which are modified by the rule, i.e. the set of semantic variables in the action of the rule that appear on the left side of an assignment or that are modified by a method call.

$modvars_c : SR_c \rightarrow \mathcal{P}(V_c)$
$v \in modvars_c(sr) \Leftrightarrow v$ is modified by sr

- Furthermore, we define a function *reachable* which computes a set of increments for each $i \in I_t$ based on a path expression, i.e. the evaluation of a path expression starting at i at time t.

  $reachable_{i,t} : PATH_s \rightarrow \mathcal{P}(I_t)$
  $i_0 \in reachable_{i,t}(p) \Leftrightarrow p$ denotes an existing path between i and i_0 at time t

```
c::propagate( pid, sender )
begin
    if $\exists sr \in SR_c : pid \in pids(sr)$ then
        $prems(sr) := prems(sr) \cup \{sender\}$;
        if $statsr(sr) =$ 'clean' then
            $statsr(sr) :=$ 'dirty';
            foreach $v \in modvars_c(sr)$ do
                if $statv(v) =$ 'clean' then
                    $statv(v) :=$ 'dirty';
                    foreach $(path, pid') \in SI_v^{st}$ do
                        foreach $i \in reachable_{self,t}(path)$ do
                            $i$.propagate($pid'$,self);
                        repeat
                    repeat
                fi
            repeat
        fi
    fi
end
```

Figure 8.2: Algorithm of Method `propagate`

In order to execute a propagation message for an increment of class c, an action is only required if there is a semantic rule sr in class c that is dependent of the variable whose change is being propagated, i.e. it is included in the set $pids(sr)$. Then the increment that sent the propagation is stored in *prems* for use during the evaluation phase. If the affected rule sr has already been stamped as dirty the propagation terminates, otherwise the rule is stamped as dirty. The propagation also terminates if sr does not modify any variable, otherwise the modified variables are stamped as dirty and a propagation message is sent to all increments that have semantic rules with a dependency declared to any of these modified variables.

As an example of a propagation scenario consider Figure 8.3. It displays an excerpt of an ASG including two subgraphs of Groupie interface modules. The semantic rules from which the propagations have been derived were discussed on Pages 152–154. Each rectangle represents an increment. Attributes are represented as ovals. Semantic relationships are depicted as dotted arrows. Dashed arrows indicate the sending of *propagate* messages. In the example the name of the exported operation `Close` in module `Window` is changed to `CloseWindow`. The operation is imported in module `WindowStack` and the semantic rules define that the import has to be marked as erroneous as soon as it is accessed.

The propagation starts at the changed increment of class `OpName`. The static scheduling information SI_{value}^{st} for `OpName` indicates that the modification must be propagated to the `ADTModule` increment. SI_{value}^{st} is inferred from the precondition of Rule 2 in class `ADTModule` (c.f. Page 154). The precondition defines that symbol table `DefinedNames` depends on the modified attribute

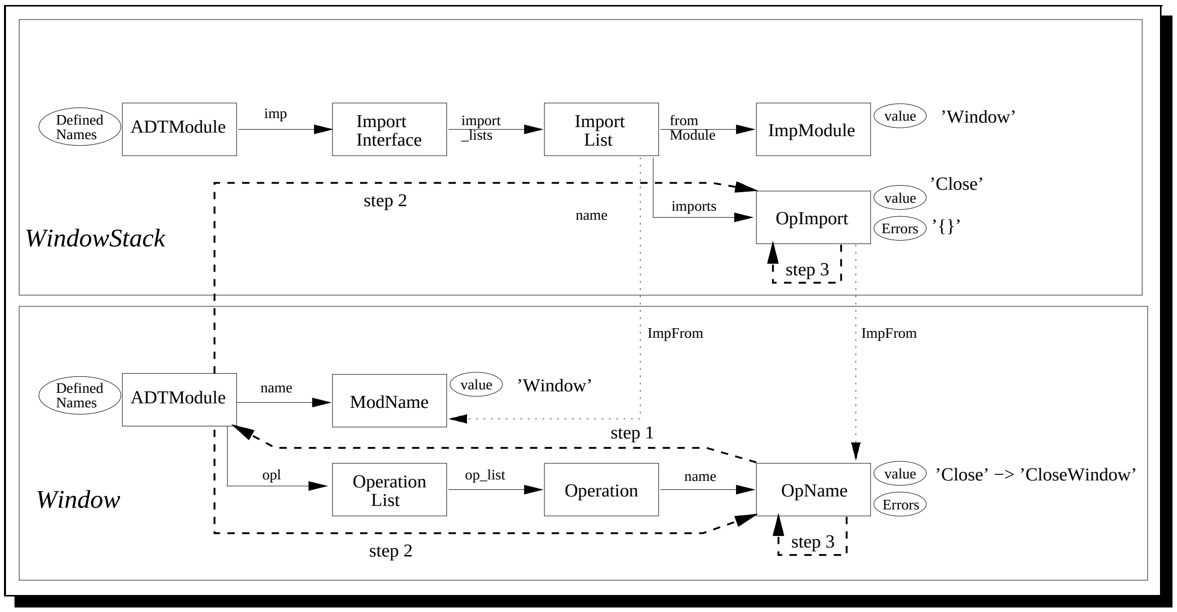


Figure 8.3: Propagation Phase

`value`. When the `ADTModule` increment receives the propagation message its dynamic scheduling information has to be updated: (1) Rule 2 is stamped dirty, (2) the sender of the propagation is included in the set of premises (*prems*) associated with the affected rule and (3) attribute `DefinedNames`, which is modified by this dirty semantic rule, is also stamped as dirty. Then the static scheduling information of class `DefinedNames` is considered in order to propagate the possible modification further. The static scheduling information for `DefinedNames` includes two propagation items that are relevant for this example. The first item is inferred from Rule 1 of class `OpName` (c.f. Page 154). It controls the value of attribute `Errors` in class `OpName`. The second propagation item addresses all increments in import lists that are connected via semantic relationship `ImpFrom` to the `ADTModule`. The propagation item is inferred from Rule 1 in class `OpImport` (c.f. Page 153). The second propagation step, therefore, causes the following updates to the dynamic scheduling information. Rule 1 of the `OpImport` increment and Rule 1 of the `OpName` increment are marked dirty. The `ADTModule` of `Window` is included in the set of premises for these two increments and attribute `Errors` in the `OpName` and link `ImpFrom` in the `OpImport` increments are stamped dirty because they might be modified by the rules that have become dirty. In a third propagation step, the `Errors` attribute of class `OpImport` is marked dirty together with Rule 1 of class `Import` that the `OpImport` increment inherited. The static propagation information is inferred from Rule 1 in class `Import` that is inherited by `OpImport`. Then the propagation phase is complete.

Incremental semantic variable evaluation begins whenever a variable v of an increment of a class c is dirty and about to be accessed. In this case method `evaluate`, a further predefined method that any class inherits from class `Increment`, is executed. This ensures that the semantic checks of a rule are performed on the basis of up-to-date information. The underlying algorithm is presented in Figure 8.4.

In order to evaluate a dirty semantic variable v all dirty semantic rules have to be executed that modify v. When a semantic rule sr is about to be executed the dynamic scheduling information is examined to find out about dirty-state semantic variables that are read by sr. These semantic variables are evaluated before the body of sr is executed. Therefore, an

```
c::evaluate( $v$ )
begin
    if $statv(v)$ = 'dirty' then
        foreach $sr \in SR_c : v \in modvars_c(sr) \wedge statsr(sr) =$ 'dirty' do
            $statsr(sr) :=$ 'executing';
            foreach $node \in prems(sr)$ do
                foreach $var \in readvars_c(sr)$ do
                    $node$.evaluate($var$);
                repeat
            repeat
            execute(sr);
            $statsr(sr) :=$ 'clean';
            $prems(sr) := \{\}$;
        repeat
    fi
    if $\exists sr \in SR_c : v \in modvars_c(sr) \wedge statsr(sr) =$ 'executing' then
        error('Cyclic dependency detected');
    else
        $statv(v) :=$ 'clean';
    fi
end
```

Figure 8.4: Algorithm of Method `evaluate`

evaluation request is passed to all increments in $prems(sr)$.

Semantic rules that are about to be executed change state from *dirty* to *executing* in order to be able to detect occurrences of cyclic dependencies between semantic variables at evaluation time[2]. A cyclic dependency is found if an increment of class c receives the request to evaluate some semantic variable v that is modified by a semantic rule $sr \in SR_c$ and sr has already changed its state to executing.

In Figure 8.5 the example of Figure 8.3 is continued assuming that attribute `Errors` of the `OpImport` increment is about to be accessed. This is, for instance, the case during unparsing when the `LayoutComputation` class decides whether to underline the representation of the `OpImport` increment. In this case the body of Rule 2 is executed for the increment of class `ADTModule` in document `Window`, because this increment is included in *prems* of the `OpImport` increment. Before Rule 2 in class `ADTModule` can actually be be executed, attribute `value` of the `OpName` increment has to be in clean state. This condition is fulfilled because there is no semantic rule in class `OpName` that modifies attribute `value`. Therefore, Rule 2 is executed for the `ADTModule` and the increment and attribute `DefinedNames` changes to clean state. Then Rule 1 of class `OpImport` is executed and it will delete the semantic relationship `ImpFrom`. After that `ImpFrom` as well as the rule change to clean state. Then Rule 1 of class `Import` can be executed and it will include the error descriptor `#NotAnOpName` into the `Errors` set. This, in turn, will cause the operation import to be displayed as erroneous at the user interface.

For simplification purposes we only discussed the incremental execution of semantic rules with `CHANGED` expressions in their conditions. The presented algorithm and the scheduling information has to be slightly extended for the execution of semantic rules with `DELETED` expressions in their conditions. These semantic rules have to be executed immediately after the propagation, otherwise the semantic rules could no longer access the deleted increments.

[2]As argued in [Sch91a] static cyclic dependencies of semantic variables in abstract syntax graphs occur so frequently that it is unreasonable to reject them at compile time.

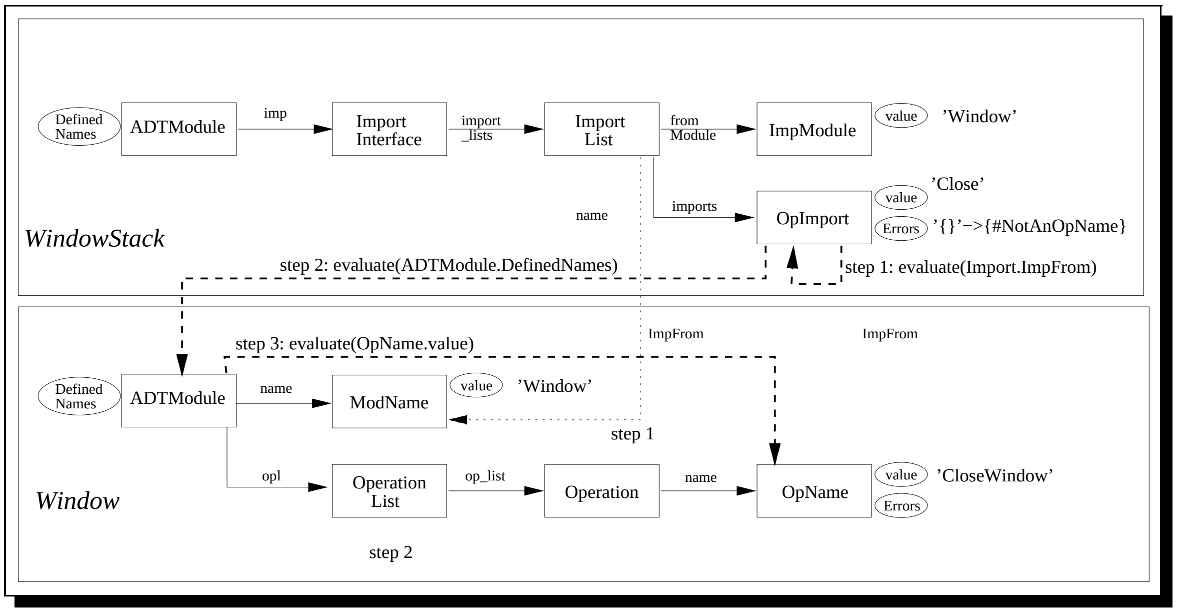


Figure 8.5: Evaluation Phase

Furthermore, information about the nodes on the path in the ASG between an increment that sends a propagation message and the receiving increment may be added to the propagation message. When a semantic rule is executed this information can be used to bind the declared identifier in `EXISTS` expressions. This enables incremental computation of complex semantic variables that depend on a large number of other variables. A symbol table, for instance, can be updated incrementally when an identifier has changed. Without the binding the entire symbol table would have to be computed anew for each single modified identifier.

8.2.2 Code Generation

In Chapter 5, we identified four main architecture components as tool-specific. These were the database schema, the programming interface to the schema, the layout computation class and the command execution subsystem. These components have to be generated from a GTSL tool specification in order to achieve full tool generation. We now sketch how this generation is implemented in the code generation phases of the various GTSL compilers.

Database Schema: The database schema consists of a set of predefined and a set of tool-specific classes. The predefined classes have already been discussed in Subsection 5.9.1. The tool-specific classes are derived from GTSL classes and inherit from the predefined classes. For each GTSL class c a class $code(c)$ is generated in the database schema. Class name and inheritance hierarchy are identical. This is possible since O_2 supports multiple inheritance. For each property of a class, an instance variable is defined in the respective schema class. GTSL types are translated into O_2C types in a straight-forward manner. For each method of a GTSL class c, regardless of its kind, an O_2C method is defined in $code(c)$. Parameter names are equivalent and types of parameter and results are generated in a straight-forward manner. Note again that this is only possible since O_2 supports the Cardelli approach to covariant redefinition in the same way as GTSL does. Signatures of methods that do not have a counterpart in GTSL are generated for the following purposes: creation and deletion of semantic relationships, access operations for instance variables that stem from exported properties, bodies of semantic rules and static scheduling data for semantic rule evaluation. The interfaces of these methods can only be defined in the linker since, in order to obtain the required information, all semantic rules must have been analysed.

For the code generation of bodies for explicit GTSL methods, we use syntax-directed translation. It is well supported by the program text generator (PTG) module of Eli. An Eli PTG specification defines a number of patterns that are very similar to unparsing schemes in syntax-directed editors. A PTG operator that may be used in semantic functions is generated for each pattern. The result of a PTG operator invocation is usually assigned to a code attribute of an abstract syntax tree node. The code attribute, in turn, may be used as an argument for other PTG operators. In this way, code is generated in a bottom-up manner and finally a dedicated PTG operator is applied to dump the contents of a code attribute to a file in the file-system. As an example consider the generation of code for GTSL `FOREACH` statements:

```
ATTR Code : PTGNode;

String          : string []
Tupel           : $ $
ForEach         : "{ " $ " " $ ";\n"
                  for (" $ " in " $ ")\n{\n"
                         $ "\n}\n"
                  "}"

RULE rule_087 :
  action_iteration ::=
        'FOREACH' action_foreach_id ':' type_declaration 'IN' expression 'DO'
        action_statement_list 'ENDDO'
COMPUTE
   action_iteration.Code=
      PTGForEach(PTGTupel(PTGString("o2 "),type_declaration.Code),
                          action_foreach_id.Code,
                          action_foreach_id.Code,
                          expression.Code,
                          action_statement_list.Code);
END;
```

The first line defines that each node in the abstract syntax tree has an attribute `Code` of type `PTGNode`. This attribute stores the code that is generated for its node. The next part in this example defines three patterns. Pattern `String` converts a character string argument into a `PTGNode` containing the string. Pattern `Tuple` defines that the result is obtained by concatenating two argument values (each `$` sign represents an argument). The third pattern defines how the code for a `FOREACH` statement is composed of five argument values. First, the "{" starts a new block so that the declaration of the cursor variable does not interfere with outer declarations. The value of the next two arguments define name and type of an O_2 local variable. Then an O_2C `for` statement is generated. The next argument is again the name of the cursor variable. It is delimited by the O_2C keyword `in` from the code of a multi-valued expression that shall be iterated. Then the code generated from a GTSL statement list is inserted. Finally the block is closed by a "}". The semantic function defined in `rule_087` shows how the operators, which are generated from the pattern definition, are applied in order to synthesise the code for a `FOREACH` statement from the code attributes of child nodes.

In the same style, patterns and semantic functions are included for each production in the subset of the class specification grammar that defines statement lists. The code attribute for statement lists is, then, not only used for the generation of explicit methods, but also for the bodies of those O_2C methods that execute semantic rule bodies.

This technique for syntax-directed code generation cannot reasonably be applied for the generation of code for implicit method bodies, such as `init` or `parse`, or for methods implementing static scheduling data. Firstly, the information required during code generation of these method bodies is spread over different sections of the interface and specification of a class. These are compiled independently. Therefore, attribute grammars, as supported by Eli, fail since there is not necessarily a syntactic relationship between the different components that could be used as attribute propagation path. Moreover, the computation of the bodies' code cannot be expressed appropriately in the language that is available for semantic functions. In particular, this language does not include primitives for loops or iterations that are required during generation of this code. We, therefore, designed a module, which supports code generation for implicit method bodies based on the data that has been collected in terms of properties of definition table entries. The module exports an operator, written in C, for each possible implicit method, which, in turn, generates the code for the method's body.

The code for the `init` method is derived from the initialisation section of the class specification documents. In addition to these specified initialisations, the `init` methods assign the reference to the father increment that is passed as an argument to the constructor to an instance variable. Furthermore, the newly created object is inserted into the O_2 version unit that is associated with the document that contains the increment.

The bodies of the implicit methods `scan` try to match the string that is passed as a parameter with the regular expression that has been provided in the class interface of the respective terminal increment class. The implementation of the pattern matching is based on the algorithms described in [AHU74]. We actually reuse the implementation of a public-domain package for extended regular expressions called `regexp`, which is available via anonymous `ftp` from the University of Toronto.

The bodies of the `parse` methods implement a *recursive descent parser*. We implement the same strategy as suggested for the Eiffel parser library [Mey89]. By defining language specific subclasses that redefine deferred `parse` methods, the Eiffel parser library can be specialised to implement parsers for arbitrary context-free languages and is not confined to lower level languages in the Chomsky hierarchy, such as LALR(1). We decided in favour of expressiveness rather than efficiency here. Efficient parsing is not so important in syntax-directed editors. In the structure-oriented mode of editing, the user decides which particular productions to choose. A parser is only required after free textual input in order to ensure syntactic correctness of the input. Free textual input is usually performed with small increments such as statements, parameters and expressions. The overhead of a context-free parser can be neglected for these small increments. In addition, the time required to create persistent objects, which represent syntax graph nodes, dominates the overall time required for parsing the text. The equivalent of methods declared in the Eiffel parser library are, in our case, implemented as methods of classes contained in the predefined schema. Unlike an Eiffel parser, however, the language specific methods that are deferred in the Eiffel library need not be hand coded in our case, but are generated from the abstract syntax and unparsing scheme specifications. The code generation for tool-specific classes, therefore, generates the implementation of those deferred methods of the Eiffel library and implement the particular syntax. The generated parser then is a *multiple-entry parser*, since the target to be parsed is defined by the object for which a `parse` method is executed. For a detailed description of the way in which `parse` methods are generated we refer to [Bud92].

Tool API Subsystem: Once the generated database schema has been established, i.e. once it has been compiled by the O_2C compiler and committed to the database, O_2 itself can generate the `ToolAPI` subsystem. As discussed in Section 5.8, the C++ interface of O_2 supports the export of an O_2C class, including a subset of the available methods, to C++. To actually have O_2 exporting a subset of the methods defined for a class, we only have to generate an export command (c.f. Page 103) that defines all methods to be exported. These are the methods that implement methods exported in GTSL class interface definitions and, in addition, the methods that implement accesses to readable GTSL properties. For generation of the export command, we again use syntax-directed translation with a number of dedicated PTG patterns.

Command Execution Subsystem: As discussed in Section 5.4, each tool command is implemented by an interaction class. The code for class definitions and implementations of interaction classes is generated using syntax-directed code generation techniques in the specification compiler. The `GetName` method is generated from the GTSL interaction name, the `IsAvailable` method is derived from the expression that follows an interaction's `ON` clause. Finally, the `Execute` method is generated from the interaction body. For its generation a number of PTG patterns are defined. Although these patterns transform syntactic elements of statement lists, they differ from the patterns for statement lists that we discussed during the code generation of explicit method bodies. The reason is that now the target language for code generation is C++ rather than O_2C. The tool builder, however, does not need to be aware of these different target languages and can use the same GTSL statements in interaction, explict method or semantic rule bodies.

Interaction collections that implement menus are generated in a straight-forward manner by the specification compiler as well. It creates an interaction collection class for each GTSL increment class. The constructor of the interaction collection class creates an object of each interaction class that has been derived from a GTSL interaction of that class. The constructor then invokes the `IsAvailable` method and inserts the interaction object into its collection only if `IsAvailable` returns `TRUE`, otherwise the interaction object is deleted. The implementation of method `GetCommandName` is then generated to invoke the `GetName` method of each interaction object included in the collection. The `Execute` method of the collection has a parameter i that addresses the i^{th} element of the collection. `Execute` from the interaction collection then invokes the `Execute` method of the interaction object that is stored at position i in the collection.

Inheritance of GTSL interactions is implemented in the interaction collection as well. As discussed in Section 5.4, there are inheritance relationships among interaction collections. The inheritance hierarchy of interaction collections is the same as the hierarchy of GTSL increment classes. Each constructor of an interaction collection executes the constructor of its super classes before executing its own body. These super class constructors, in turn, create and eventually insert interaction objects implementing interactions of GTSL super classes into the collection. Thus the interaction collection includes interaction objects that correspond to interactions of different GTSL classes. Redefinition of interactions is resolved by replacing an interaction object that is already in the collection with a new one if they have the same name.

Layout Computation Class: As discussed in Section 5.3, the `LayoutComputation` class exports a method `unparseIncrement`. It checks for the dynamic type of increment that is passed as a parameter and invokes the respective hidden unparse operation for that type.

Since all classes need to be known for that, the `unparseIncrement` method cannot be generated by the specification compiler, but is created by the linker.

The bodies of hidden unparse methods are generated by the specification compiler. These hidden operations perform a depth-first traversal of the syntax tree starting from the increment that was passed as an argument to `unparseIncrement`. Each operation interprets the unparsing information stored in the symbol table of each class and generates invocations of method `InsertIncrementPart`, which is exported from the user interface subsystem for pretty-printing items and keywords. In order to insert text segments from a child increment into the edit window, the hidden unparse operation generated for that type is invoked.

8.3 Summary

We have discussed the importance of having static semantic checks in the front ends of the various GTSL compilers and how these checks can be generated from an ordered attribute grammar definition using the techniques that are implemented in Eli. Some weaknesses of Eli have been identified in the process. In general, however, Eli proved to be very useful for the *"construction of application generators"* as claimed in [Kas94]. We then discussed code generation that is performed in the back end of the compilers. For most tool-specific architecture components generation is straight-forward. The implementation of semantic rules, however, is not as obvious. The reason is the conceptual gap between declarative specification of rules in GTSL and the basic mechanisms that can be used for their implementation. We have discussed why semantic rules cannot be implemented with triggers and instead suggested an incremental evaluation algorithm. Finally, we sketched how the tool-specific architecture components are generated by the back ends of the various GTSL compilers. We are now in the position to generate tools for the GENESIS environment from GTSL specifications.

Chapter 9

Evaluation

GTSL and the GTSL compiler have been successfully used on a number of occasions. The Groupie module interface editor, which we used as a running example throughout this thesis, has been generated from a GTSL specification. In [dSR95] a tool for the design of Beta class definitions [LMN93] was specified in GTSL and integrated a-posteriori with a graphical editor for the OMT notation. Moreover, we used GTSL to specify and generate a subset of the tools for the GENESIS environment. Finally, GTSL was used to specify and generate several tools for C++ class library development and maintenance.

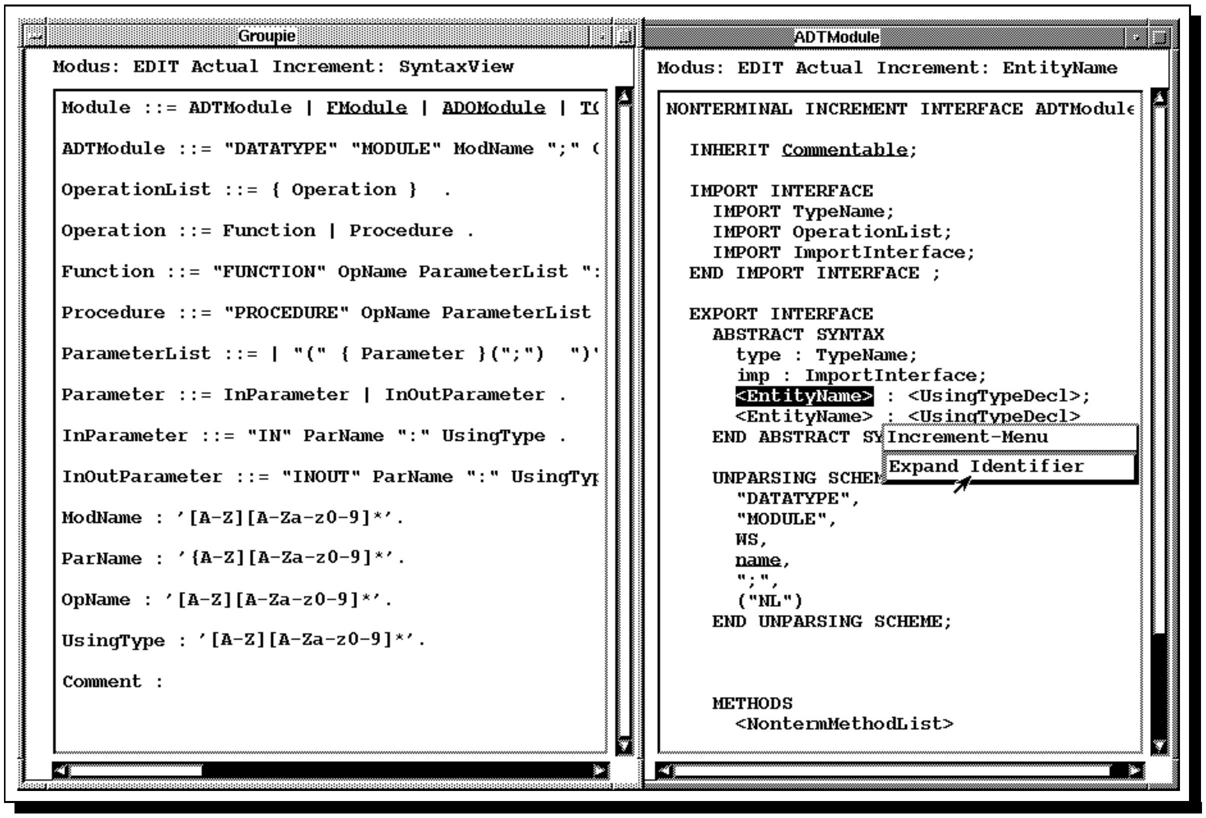
The purpose of this chapter is to highlight the strengths and weaknesses of our tool construction approach. The evaluation will be carried out with respect to three dimensions. Firstly, we will assess whether tools generated from GTSL specifications meet the functional requirements that we demanded in Chapter 2. Secondly, we want to see whether or not GTSL can express the various concerns that are tool-specific. In particular, we evaluate the way for modelling different strategies of inter-document consistency constraint handling. We also set out to quantify the effort that is required for the GTSL specification of tools and compare it with the case where tools are implemented in an object-oriented programming language. As we especially focussed on efficiency within this thesis we, finally, also evaluate the run-time performance of generated tools. The evaluation is based on two complementary evaluation scenarios, tools for C++ class library development and tools of the GENESIS environment.

This chapter will further be structured as follows. We outline the evaluation scenarios in the next section and discuss why they are appropriate and to what degree they are complementary. Then we discuss, from a user's point of view, how well the functional tool requirements are met by tools that were generated from GTSL specifications. After that, in Section 9.3, we evaluate the appropriateness of GTSL for tool specification purposes. We discuss this from a qualitative as well as from a quantitative perspective. We then discuss performance measurement results of the execution times of archetypical tool commands that we obtained during the execution of the various generated tools in Section 9.4.

9.1 Evaluation Scenarios

9.1.1 The GENESIS Environment

The first scenario is the GENESIS environment whose requirements have been delineated in Chapter 7. For the evaluation we generate an ENBNF editor and a class interface editor in such a way that they are integrated a-priori and that they obey consistency constraints that have been discussed in Chapter 7.



```
Modus: EDIT Actual Increment: SyntaxView

Module ::= ADTModule | FModule | ADOModule | T(
ADTModule ::= "DATATYPE" "MODULE" ModName ";" (
OperationList ::= { Operation }  .
Operation ::= Function | Procedure .
Function ::= "FUNCTION" OpName ParameterList ":
Procedure ::= "PROCEDURE" OpName ParameterList
ParameterList ::= | "(" { Parameter }(";")  ")'
Parameter ::= InParameter | InOutParameter .
InParameter ::= "IN" ParName ":" UsingType .
InOutParameter ::= "INOUT" ParName ":" UsingTyp
ModName : '[A-Z][A-Za-z0-9]*'.
ParName : '{A-Z][A-Za-z0-9]*'.
OpName : '[A-Z][A-Za-z0-9]*'.
UsingType : '[A-Z][A-Za-z0-9]*'.
Comment :
```

```
Modus: EDIT Actual Increment: EntityName

NONTERMINAL INCREMENT INTERFACE ADTModule

  INHERIT Commentable;

  IMPORT INTERFACE
    IMPORT TypeName;
    IMPORT OperationList;
    IMPORT ImportInterface;
  END IMPORT INTERFACE ;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      type : TypeName;
      imp : ImportInterface;
      <EntityName> : <UsingTypeDecl>;
      <EntityName> : <UsingTypeDecl>
    END ABSTRACT SY

    UNPARSING SCHEM
      "DATATYPE",
      "MODULE",
      WS,
      name,
      ";",
      ("NL")
    END UNPARSING SCHEME;



    METHODS
      <NontermMethodList>
```

Figure 9.1: Tools of the GENESIS Environment

We evaluate whether GTSL can express the various inter-document consistency constraints between ENBNFs and class interfaces. In particular, we have required that consistency is preserved as far as possible in terms of change propagations. We, therefore, evaluate how the handling of these constraints is supported by the tools and how this is specified in GTSL.

9.1.2 The British Airways SEE

The requirements for the second scenario have been provided by British Airways, an industrial partner of the GOODSTEP project [GOO94]. British Airways does most of its software development in-house, with an increasing number of projects using object-oriented techniques. A department of the IT infrastructure division, with seven developers at present, is supposed to design, implement and document class libraries for the purpose of corporate reuse. The second evaluation scenario, which we refer to as *BA SEE* in the following, is an environment that supports this class library development and maintenance process.

Design documents of class libraries are defined in the Booch notation [Boo91]. A Booch class diagram is structured into *categories* that serve structuring purposes, like subsystems in our entity relationship notation. A category may contain a number of classes and nested categories. The most recent definition of the Booch notation is strongly tight towards C++, which is the programming language used for any object-oriented development at British Airways. Booch distinguishes between three different kinds of classes. These are *template classes*, *instantiation classes* and *plain classes*. In addition, Booch diagrams identify different kinds of relationships between classes, namely inheritance, has- and use-relationships. They have different C++ specific *adornments*. Inheritance may, for instance, be public, protected or private.

The language used for class definitions at British Airways is, in fact, only a subset of the C++ programming language. The subset is determined by British Airways corporate programming guidelines. The guidelines exclude C++ statements, such as ellipses, inlines and friends, whose use would be contrary to accepted software engineering principles. Apart from a class definition, further declarations related to the class, like enumerations, type definitions or constants can be defined. Moreover, the dependencies of a class definition document to other class definitions are defined in terms of `#include` statements and forward declarations.

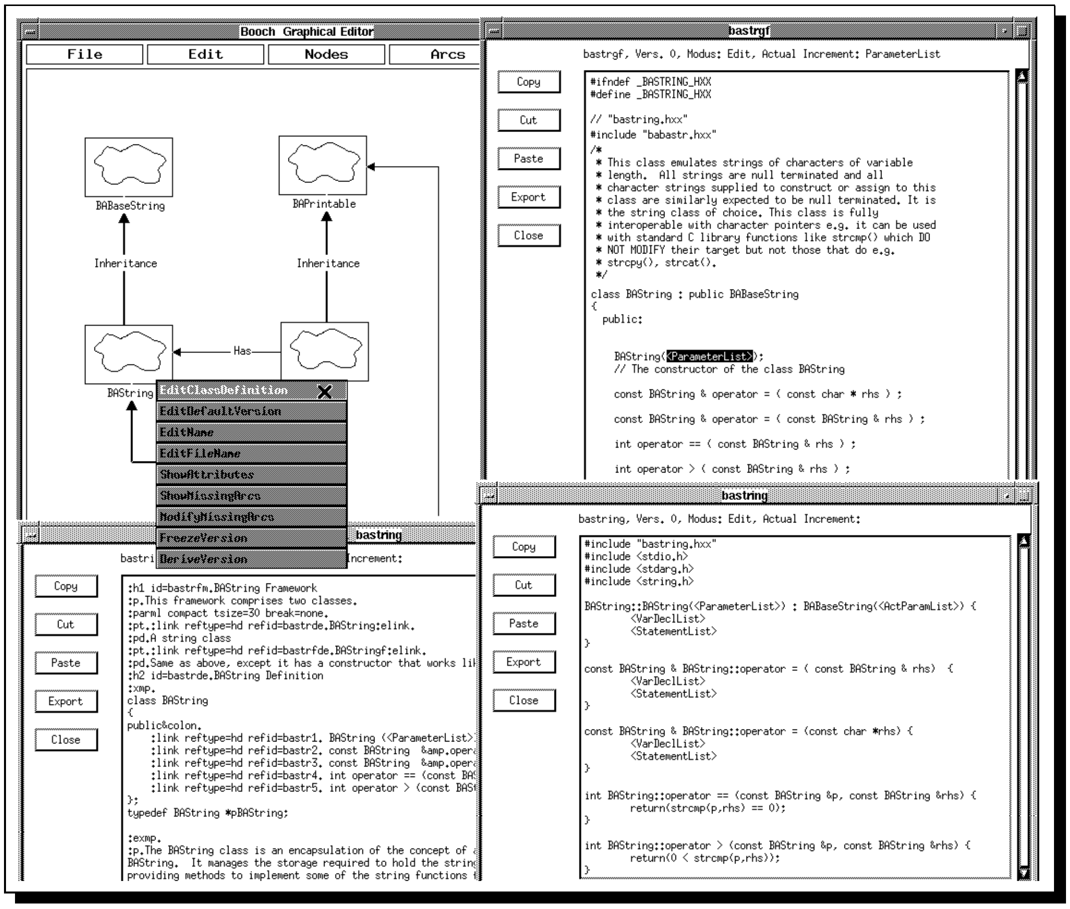
An implementation has to be provided for each defined class. Therefore, a further document type includes C++ method implementations. Each method that has been defined in a class interface must be implemented in the respective class implementation document.

Since class libraries are developed for corporate reuse, they must be accompanied by documentation that enables customers to reuse classes from the library. Apart from the design that identifies the different dependencies of classes, customers require a technical documentation of the functionality provided by the public methods of a class. This technical documentation is defined in the *information processing facility (IPF)*, a mark-up language for online-hypertexts defined by IBM. A hypertext, which can be compiled from an IPF document, is then delivered to customers to provide an on-line help facility that supports use of the library. For each class of the library an IPF document must be provided. Besides the signatures of public methods of the class, it also includes a description of each method and examples illustrating how to use the methods.

The two scenarios are complementary for several reasons. The Booch diagram editor is a graphical tool that has been provided by another GOODSTEP partner, whereas the three other tools have been generated from GTSL specifications. Preserving inter-document consistency constraints between Booch diagrams and the other three document types can thus only be achieved a-posteriori with tool-specific services. In contrast, the two tools of the GENESIS environment are integrated a-priori. This integration between ENBNF documents and GTSL class interface definitions cannot be achieved with different views of the same real abstract syntax graph. ENBNFs and class definitions are, from a structural point of view, too different. Instead, integration is defined in terms of inter-document reference edges between ENBNF and class interface abstract syntax graphs. The three textual documents of the BA SEE, however, are very similar from a structural point of view. There is no reason for tolerating consistency constraint violations between these documents because the same user is responsible for any of these documents that have to be defined for a class. Moreover, they are always versioned together. Therefore, tools for these documents can be integrated a-priori based on a common conceptual database schema. Class definitions, implementations and documentations are then considered as different views that are defined on top of this conceptual schema.

9.2 Satisfaction of Functional Requirements

Figure 9.1 on Page 208 displays the user interface of the *ENBNF tool* and the *GTSL class interface tool*. Figure 9.2 displays the user interfaces of tools contained in the BA SEE. A *Booch editor* is available for defining Booch diagrams. A *class definition tool* is used to edit, analyse and browse through C++ class definitions, a *method implementation tool* is used for defining method implementations and an *IPF tool* serves documentation purposes.



bastrgf, Vers. 0, Modus: Edit, Actual Increment: ParameterList

```
#ifndef _BASTRING_HXX
#define _BASTRING_HXX

// "bastring.hxx"
#include "babastr.hxx"
/*
 * This class emulates strings of characters of variable
 * length.  All strings are null terminated and all
 * character strings supplied to construct or assign to this
 * class are similarly expected to be null terminated. It is
 * the string class of choice. This class is fully
 * interoperable with character pointers e.g. it can be used
 * with standard C library functions like strcmp() which DO
 * NOT MODIFY their target but not those that do e.g.
 * strcpy(), strcat().
 */
class BAString : public BABaseString
{
  public:


    BAString(<ParameterList>);
    // The constructor of the class BAString

    const BAString & operator = ( const char * rhs ) ;

    const BAString & operator = ( const BAString & rhs ) ;

    int operator == ( const BAString & rhs ) ;

    int operator > ( const BAString & rhs ) ;
```

bastri Increment:

```
:h1 id=bastrfm.BAString Framework
:p.This framework comprises two classes.
:parml compact tsize=30 break=none.
:pt.:link reftype=hd refid=bastrde.BAString:elink.
:pd.A string class
:pt.:link reftype=hd refid=bastrfde.BAStringf:elink.
:pd.Same as above, except it has a constructor that works lil
:h2 id=bastrde.BAString Definition
:xmp.
class BAString
{
public&colon.
    :link reftype=hd refid=bastr1. BAString (<ParameterList>)
    :link reftype=hd refid=bastr2. const BAString  &amp.opera
    :link reftype=hd refid=bastr3. const BAString  &amp.opera
    :link reftype=hd refid=bastr4. int operator == (const BAS
    :link reftype=hd refid=bastr5. int operator > (const BAS
};
typedef BAString *pBAString;

:exmp.
:p.The BAString class is an encapsulation of the concept of
BAString.  It manages the storage required to hold the strin
providing methods to implement some of the string functions
```

bastring, Vers. 0, Modus: Edit, Actual Increment:

```
#include "bastring.hxx"
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

BAString::BAString(<ParameterList>) : BABaseString(<ActParamList>) {
        <VarDeclList>
        <StatementList>
}

const BAString & BAString::operator = ( const BAString & rhs)  {
        <VarDeclList>
        <StatementList>
}

const BAString & BAString::operator = (const char *rhs) {
        <VarDeclList>
        <StatementList>
}

int BAString::operator == (const BAString &p, const BAString &rhs) {
        return(strcmp(p,rhs) == 0);
}

int BAString::operator > (const BAString &p, const BAString &rhs) {
        return(0 < strcmp(p,rhs));
}
```

Figure 9.2: Tools Contained in the BA SEE

Inter-Document Consistency Constraints: Numerous inter-document consistency constraints exist between documents managed by the BA SEE. These are checked and preserved by the tools contained in the environment. Each class in the Booch diagram must be refined in terms of a class definition, an implementation and an IPF documentation. Upon creation of a class, the Booch editor creates these documents as well. If a class name is changed, the Booch editor consistently changes the class names in the corresponding documents. Moreover, the relationships that are defined in the Booch diagram must be reflected in the class definition and implementation documents. The Booch editor creates, for instance, the corresponding C++ declarations for an inheritance relationship between two classes in the Booch diagram. If an adornment of a relationship is changed, the respective declaration is consistently changed in the class definition. For each use-relationship between classes in the Booch diagram, there must be an `#include` statement in the respective class definition and for each has-relationship there must be a respective data member declaration. Adornments of has-relationships must match names of data members. Similarly, the integration of the three textual tools causes the insertion of the respective counterparts into the corresponding implementation and IPF documents, upon creation of a method. Furthermore, the matching of method signatures of the class definition and their corresponding definitions in the implementation and IPF documents is ensured.

The GENESIS environment preserves the inter-document consistency constraints that exist between ENBNFs and GTSL class interface definitions by means of change propagations. If a tool builder, for instance, creates a new production, a respective class definition is automatically created. Moreover, inheritance relationships are introduced depending on the kind of production. If symbols are inserted on the right-hand side of an ENBNF structure production, respective abstract syntax children are inserted into the abstract syntax section of the non-terminal class interface that is associated with the production. Likewise, the regular expression section of a terminal increment class interface is automatically generated from the right-hand side of a terminal production.

Versions and Configurations: Version and configuration management of class definitions in the BA SEE is required because different customers reuse different configurations of a class library. The reuse department must be able to restore any of these configurations, for instance to fix a bug. The Booch diagram determines a configuration of a class library as follows. A version label is associated with each class of the Booch diagram. The Booch editor then provides a command to derive a new version of a class that, in turn, increases the version label and derives new versions of the class definition, implementation and IPF documents. Moreover, the Booch editor provides a command to open that document version of the class interface, implementation or IPF document identified by the version label. The version labels, therefore, determine version selection rules and different versions of the Booch diagram then define different configurations.

Process Sensitivity: The class library development and maintenance is done under control of a software process defined in the SLANG language [BFG93b] and enacted by the SPADE process engine [BFGG92]. For a description of the process model we refer to [Rod95]. The Booch editor is a process-sensitive tool and is integrated in the process model based on the SPADE communication interface (SCI). Three different *roles* are identified in the process model. A *librarian* is the contact point for all library customers. He schedules bug reports and change requests to particular developers and decides when to release a new library configuration. An *implementor* modifies the library and edits Booch diagram, class definition, implementation and IPF documents. Finally, a *tester* is responsible for the correctness of library updates. The granularity of documents that is considered by the process model is a library since these are the units of task assignment. The process definition for accessing and modifying the other document types are implemented by the other tools and their integration with the Booch tool. Whenever a user starts the Booch tool the user has to declare his or her role. Then the process model displays the libraries that the user can currently access and the user can load one of the libraries, in terms of a Booch diagram, into the Booch editor. The user may then use the Booch editor to browse to versions of class definition, implementation and IPF documents. It is not necessary to control accesses to these document types from the process model because they do not affect the work of other users. A user can then change the state of the library configuration in the process model. A tester may define that the library update has been successful and that the library can now be released by the librarian.

Concurrent Editing: Multiple users may concurrently update the same document version. The British Airways process model excludes multiple users concurrently opening the same library. This is enforced by SPADE during the enactment of the process definition. Nevertheless, there is a need to support concurrent document version updates because classes of a library might use classes contained in other libraries. Thus one user might edit a class definition that uses a declaration of a class contained in another library. If this class is concurrently edited by another user, who then changes the used declaration, the impact of this change needs to be displayed concurrently. Therefore, the first user will receive a message saying that a used declaration has been concurrently updated and then the impact of this update is concurrently displayed. In the GENESIS environment there is no process control at all and, therefore, there is no restriction of concurrent accesses to the same document version. The required concurrency control in both environments is implemented on the basis of the transaction mechanism that is used during command execution.

Syntax-directed Editing: The corporate programming guidelines for C++ have been defined by the same department that is in charge of library development and maintenance. Interestingly not even the developers who released the guidelines obeyed them. This problem has been remedied by the introduction of syntax-directed editing facilities, which enable only those C++ definitions that comply with the programming guidelines to be edited.

Unlike C++, the languages for tool specification purposes are not widely known. By means of structure-oriented editing, users of the GENESIS environment are, therefore, significantly supported in the efficient definition of a tool specification without having to go through tedious edit-compile cycles. The users of the BA SEE, who have already acquired fundamental C++ programming skills, however, tend to use the free textual input facilities that are available in the various BA SEE editors.

For the BA SEE, a great number of static semantic constraints defined for C++ class definitions have been implemented in terms of GTSL semantic rules. The class definition tool, therefore, checks violations of static semantics incrementally and shows them to the user by underlining erroneous increments. The static semantic analysis is done in a way transparent to users, i.e. users need not explicitly start a check command nor keep track of the increments that have been checked already. Moreover, the semantic analysis is done incrementally, i.e. only those increments that have been changed since the last static semantic analysis are re-analysed.

Analysis and Browsing: A number of cross-reference analysis commands are available in the C++ class definition editor of the BA SEE. These are required because class definitions also have to be understood by those developers who have not developed them. The analysis commands allow a user, for instance, to identify all the class definitions where a particular class or type declaration is used. The tool also offers browsing commands, for instance, to display the declaration of a used class definition. In addition, the tool supports the removal of errors. If an identifier is declared twice, the original declaration of the identifier can be visited by means of a browsing command.

Persistence and Integrity Any command of the syntax-directed tools for class definition, implementation and documentation of the BA SEE is performed as an ODBS transaction. This means that the effect of a command is immediately persistent and the potential loss of effort in the case of a hardware or software failure is restricted to the last command execution that has not been completed. However, integrity may be violated due to the fact that the Booch editor does not exploit ODBSs to achieve persistence and integrity. Therefore, all changes made since a Booch diagram has been saved will be lost in case of a failure. Then the integrity between the saved Booch diagram and the class definitions, implementations and documentations might be violated.

Compilation Support: The BA SEE also supports the compilation of a library. The Booch editor generates a make-file from a Booch diagram. According to the programming guidelines, each class has to be compiled separately. The make-file, therefore, enumerates the different classes as compilation units. In addition, the Booch editor then exploits the various relationships in the Booch diagram to determine dependency rules for the make-file. The Booch editor also offers a command to dump the contents of a class library into a file-system. This dump is incremental in the sense that only modified documents are dumped and, therefore, the compilation of the library, which is based on the file-system time stamps is also incremental. Likewise the GENESIS class definition editor is able to dump all GTSL class definitions into the file-system in order to enable the GTSL compiler to access them.

9.3 Tool Specification

In this section, we demonstrate how GTSL is exploited to construct the ENBNF and class interface tools for the GENESIS environment. For each of the two tools, we discuss the ENBNF the entity relationship model and the inheritance diagram. Then we present excerpts from class interface and specification definitions that demonstrate the specification of static semantic and inter-document consistency checks and even their preservation by means of change propagations. While doing so, we indicate how the solutions can be generalised for the construction of other tools. We then outline those strategies for the definition of inter-document consistency constraint handling in the BA SEE that are different from the GENESIS environment. Finally, we measure the complexity of the tool specifications and compare it with the size of the generated code.

9.3.1 ENBNF Editor

The purpose of the ENBNF editor is to support a tool builder in defining the syntax of a language in terms of a normalised ENBNF, as defined in Section 6.2. In particular, the tool will support hybrid syntax-directed editing of these ENBNFs and check for their static semantic correctness. In addition, the editor will generate initial GTSL class interface definitions and propagate the tool builder's changes into these class definitions. To define this propagation, the ENBNF editor specification must import classes from the interface editor specification. We, therefore, postpone the discussion of these change propagations to Subsection 9.3.3 and discuss the ENBNF and the class interface editor first.

9.3.1.1 ENBNF of ENBNF

Figure 9.3 defines the ENBNF of the ENBNF language. The definition has been derived by normalising the language definition given on Page 122. The extensions defined for ENBNFs

```
ENBNF          ::= ProductionList .
ProductionList ::= {Production} .
Production     ::= Alternative | Structure | StructureOpt | Regular | RegularOpt .
Alternative    ::= DefiningSymbol "::=" SymbolList .
Structure      ::= DefiningSymbol "::=" ComponentList .
StructureOpt   ::= DefiningSymbol "::=" "|" ComponentList .
Regular        ::= DefiningSymbol ":" RegExp .
RegularOpt     ::= DefiningSymbol ":" "|" RegExp .
SymbolList     ::= {UsingSymbol}("|") .
ComponentList  ::= {Component} .
Component      ::= ListProd | Keyword | RegExp | UsingSymbol .
ListProd       ::= "{" UsingSymbol "}" Delimiter .
Delimiter      ::= | "(" KeywordList ")" .
KeywordList    ::= {Keyword} .
Keyword        : '["][^"]*["]' .
RegExp         : '[']['^']*[']' .
DefiningSymbol : '[a-zA-Z_][a-zA-Z_0-9]*' .
UsingSymbol    : '[a-zA-Z_][a-zA-Z_0-9]*' .
```

Figure 9.3: ENBNF of ENBNF

enable the language to be defined in a more concise way. It, therefore, contains fewer productions. In particular, list structures such as `SymbolList` and alternatives can be defined more appropriately.

A further difference arises since we anticipate the specification of static semantics. We know that for each ENBNF production, a GTSL class will be derived. Moreover, we know that static semantics is defined in terms of semantic rules that are attached to class specifications. We, therefore, take the constraint SV2 (c.f. Page 123) into account and distinguish between using symbols that occur on the right-hand sides of productions and defining symbols that occur on the left-hand side.

The above observation applies in general. It is always the terminal symbols, which are the smallest unit in the grammar, that carry semantics. Therefore, we must declare different terminal symbols in the grammar, although they might have the same lexical syntax, if they play different roles in static semantics. This ensures that the different symbols are translated into different GTSL terminal increment classes. The different semantic properties are then defined by the semantic rules of these different increment classes.

9.3.1.2 Inheritance Diagram of ENBNF Editor

The inheritance diagram of the ENBNF editor is displayed in Figure 9.4. It identifies the different GTSL classes that are derived from the ENBNF and their inheritance relationships to predefined classes. Predefined class names are typeset in italics. There are three classes which have not been derived from ENBNF productions, but are added here. These are `ScopingBlock`, `NameInST` and `UsingNameInST`. These classes are abstract GTSL classes that will implement static semantic constraints SV1 and SV2. We could also have defined these constraints in their subclasses, i.e. in `ENBNF`, `DefiningSymbol` and `UsingSymbol`. The constraint for uniqueness of identifiers in a particular scope and the constraint that applied identifiers must be declared occur in almost any typed language. We, therefore, define these constraints in abstract classes in such a way that these classes can be reused in other tool specifications.

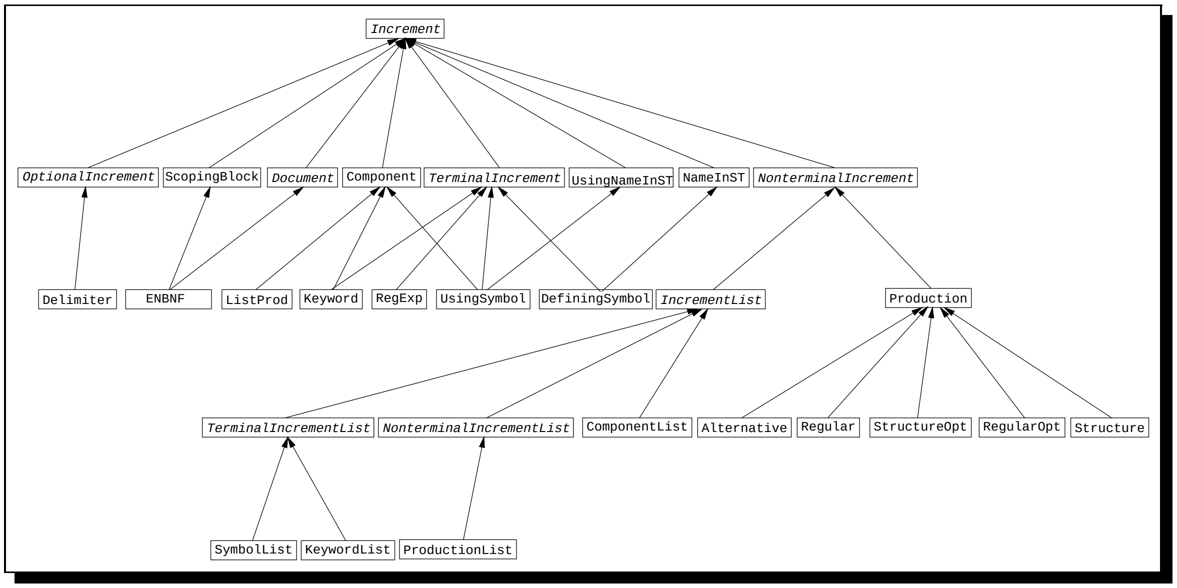


Figure 9.4: Class Hierarchy of ENBNF Editor

9.3.1.3 Entity Relationship Model of ENBNF Editor

Figure 9.5 defines the top-level entity relationship diagram of the ENBNF editor specification. The diagram visualises abstract syntax children as aggregation relationships. The names of these relationships are defined by the tool builder here in the entity relationship diagram. In addition, three semantic relationships that are the baseline for specification of static semantics are defined.

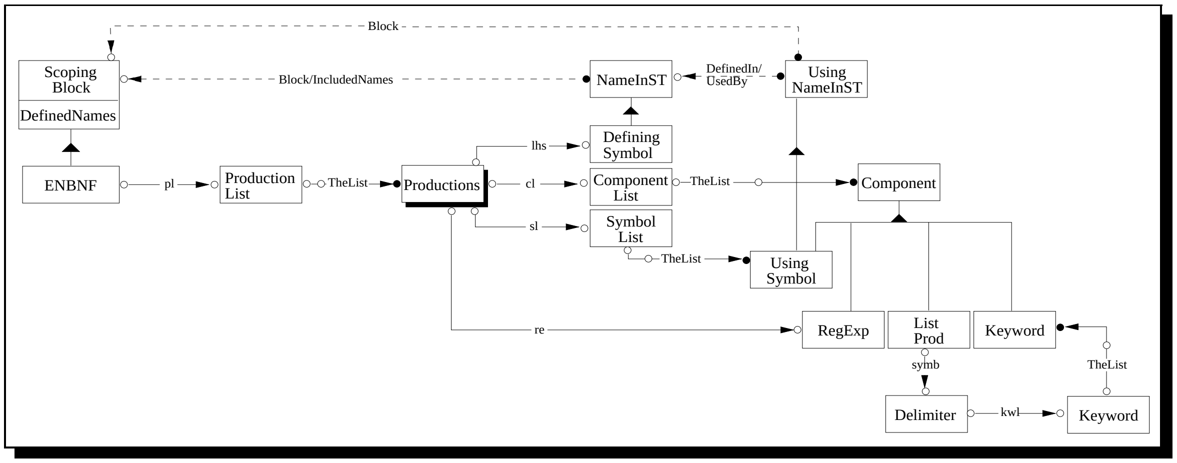


Figure 9.5: Entity Relationship View of Syntax View

The first semantic relationship connects defining occurrences of names with the increment that defines the scoping block. This increment has an attribute `DefinedNames` which is an instance of the non-syntactic class `DuplicateSymbolTable`. It is used to store all declaring occurrences of names. The explicit link of the first semantic relationship is defined in `NameInST` and is called `Block`. The corresponding implicit link is defined in `ScopingBlock` and is called `IncludedNames`. This link refers to the set of those declaring names that are defined within the scope. The second relationship connects using names with the scoping block. This relationship is required in semantic rules of class `UsingNameInST` in order to define name lookups. The last semantic relationship connects using names with those increments that declare the name. The explicit link `DefinedIn` is defined in `UsingNameInST` and used in semantic rules of class `UsingNameInSt`. The corresponding implicit link `UsedBy` is used in `NameInST` to allow for the propagation of changes of the name to all using names. The classes `ENBNF`, `DefiningSymbol` and `UsingSymbol` then inherit these semantic relationships.

The entity relationship diagram in Figure 9.5 contains a subsystem `Productions`. This subsystem is refined by the entity relationship diagram in Figure 9.6. It defines five increment classes that represent the different kinds of production in our normalised EBNF as subclasses of class `Production`. The subclass relationship has been derived from the alternative production `Production` in the ENBNF. Note that we have simplified the abstract syntax relationship that was derived from the ENBNF. Any of the subclasses of `Production` have an abstract syntax child of class `DefiningSymbol`. We have simplified that by replacing the five abstract syntax children in each production increment with an abstract syntax child `lhs` in class `Production`. The subclasses now inherit the abstract syntax child from `Production`.

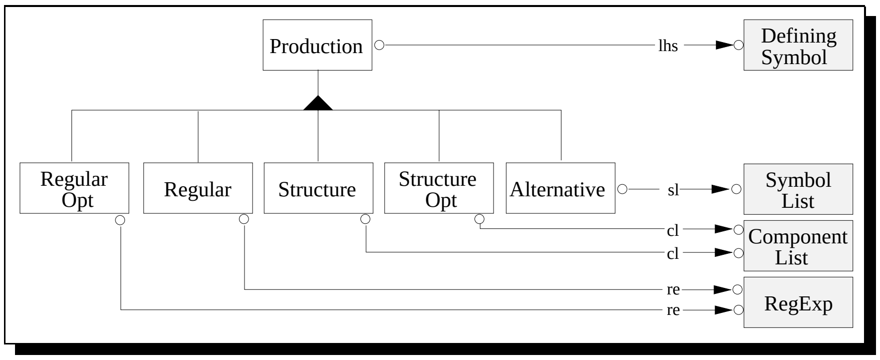


Figure 9.6: Refinement of Subsystem `Productions`

This strategy should be applied in general. The productions that define symbols of an alternative often have particular child increments in common. In that case these children should be defined in the super class that has been derived from the alternative production rather than in all classes derived from the alternative symbols. This concern is reinforced when we consider methods, interactions and semantic rules. They often depend on a single child increment. If the child is defined in the super class methods, semantic rules and interactions can also be defined there and reused in all subclasses.

9.3.1.4 Static Semantics

We now use the attribute and the semantic relationships that have been defined in the entity relationship diagram to define the static semantics of the ENBNF. A new symbol table is created for each new scoping block. This is defined in the initialisation section of class `ScopingBlock`. For a detailed definition of the initialisation section we refer to the Appendix A.3.2, Page 280. Its purpose is to permit user-defined initialisations during increment construction. Note that class `ENBNF` inherits this initialisation. The symbol table has to be updated whenever a new name is defined in the scope, an existing name changes or a name is deleted from the scope. This is defined in a semantic rule of class `ScopingBlock` from which `ENBNF` inherits.

```
INCREMENT SPECIFICATION ScopingBlock;
INITIALIZATION
  DefinedNames := NEW DuplicateSymbolTable;
END INITIALIZATION;

SEMANTIC RULES
  ON EXISTS(name : NameInST IN SELF.IncludedNames):
     CHANGED(name.value) OR DELETED(name)
  ACTION
     SELF.DefinedNames.associate(name,name.value);
  END ACTION;
END SEMANTIC RULES;
END INCREMENT SPECIFICATION ScopingBlock.
```

If a new name increment is created, the semantic relationship between name and scoping block increments must be established. This is defined in the initialisation section of class `NameInST`. As defined below, the result of executing method `envelopingScope` is assigned to the explicit link `Block`. The method traverses to father increments until it reaches an increment of class `ScopingBlock` or one of its subclasses. Assigning this increment to link `Block` implicitly inserts the newly created object into the implicit link `IncludedNames` of class `ScopingBlock`. This, in turn, implies that the newly created increment is now reachable via the path `SELF.IncludedNames` and then the semantic rule above is fired. This rule then changes the attribute `DefinedNames` and then the semantic rule of class `NameInST` given below fires.

```
INCREMENT SPECIFICATION NameInST;
INITIALIZATION
  Block := SELF.envelopingScope();
END INITIALIZATION;

SEMANTIC RULES
  ON CHANGED(SELF.Block.DefinedNames)
  ACTION
    IF(SELF.Block.DefinedNames.is_duplicate_incr(SELF)) THEN
      SELF.Errors.append_error(SELF.ErrorId());
    ELSE
      SELF.Errors.clear_error(SELF.ErrorId());
    ENDIF
  END ACTION;
END SEMANTIC RULES;


METHODS
  METHOD envelopingScope():ScopingBlock;
  VAR i: Increment;
  BEGIN
    i:=SELF.father;
    WHILE ((i!= NIL) AND (NOT i.IS_KIND_OF("ScopingBlock"))) DO
      i := i.father;
    ENDDO;
    RETURN(<ScopingBlock>i);
  END envelopingScope;
END INCREMENT SPECIFICATION NameInST.
```

The semantic rule investigates whether the name is duplicate or not by invoking the method `is_duplicate_incr` from the symbol table attribute of the scoping block. If it returns `TRUE`, the name is not unique and an error descriptor is inserted into attribute `Errors`, which is inherited from class `Increment`, otherwise the error descriptor is deleted from the error set.

Note that the error descriptor to be inserted into the set is tool-specific and cannot be defined in this abstract class. We, therefore, invoke a deferred method `ErrorId` for computing the error descriptor rather than inserting a constant error descriptor into the set. By exploiting late binding, we have managed to postpone the decision as to which error descriptor to use to the definition in some tool-specific subclass of `NameInST`.

Class `UsingNameInST` defines the common properties for using names. Therefore, we have defined an explicit link `Block` of a semantic relationship to the increment of a subclass of `ScopingBlock`. This relationship is established upon creation of a using name in the same way as it was established for defining names.

The relationship is exploited in a semantic rule in order to lookup names in the symbol table of the scoping block. The semantic rule below fires whenever either the symbol table or the lexical value of the using name is changed. The rule then first performs a lookup in the symbol table by invoking method `increment_at` with the symbol table `DefinedNames`. This method returns the undefined value `NIL` if the name is not included, otherwise it returns a reference to the increment that declares the name. If the reference is defined, the semantic rule investigates the type of the increment using the GTSL operator `KIND_OF`. Again the conformity is tool-specific. Therefore, we again define a deferred method and then determine the type we are looking for in a subclass. In addition, we must check for conformance of `NameInST`, since otherwise we would not be allowed to specialise the found symbol using the type cast `<NameInST>`. If the increment that was returned by the lookup is conform to both types, the semantic rule establishes a semantic relationship between the using and defining increments by assigning the increment to the explicit link `DefinedIn`. Note again that this assignment also modifies the implicit link `UsedBy` in class `NameInST`. It now also includes a reference to the increment for which the semantic rule was fired. The semantic rule then removes the error descriptor that again is determined by a deferred method. In all other cases the semantic relationship is deleted and the error descriptor is added to the set of errors.

```
INCREMENT SPECIFICATION UsingNameInST;
...
SEMANTIC RULES
  ON CHANGED(Block.DefinedNames) OR CHANGED(value)
  VAR inc: Increment;
  ACTION
    inc:=SELF.Block.DefinedNames.increment_at(SELF.value);
    IF inc != NIL THEN
      IF( inc.IS_KIND_OF("NameInST") AND
          inc.IS_KIND_OF(SELF.DeclClassName())) THEN
        DefinedIn := <NameInST>inc;
        Errors.clear_error(SELF.ErrorId());
      ELSE
        DefinedIn:=NIL;
        Errors.append_error(SELF.ErrorId());
      ENDIF;
    ELSE
      DefinedIn:=NIL;
      Errors.append_error(SELF.ErrorId());
    ENDIF;
  END ACTION;
END SEMANTIC RULES;
END INCREMENT SPECIFICATION UsingNameInST.
```

We required the static semantic constraints in the ENBNF not to be violated. Without additional means, the ENBNF editor would not help the tool builder to achieve consistency as much as it could. In particular upon a change to a defining symbol, the using symbols that were consistent before would become inconsistent. We overcome this deficiency by propagating the change from the defining symbol to all using symbols. Since this is tool-specific, we define it in a method of `DefiningSymbol` rather than in `NameInST`. The method section first defines method `ErrorId` that determines the error descriptor by redefining the deferred method of its super class. Then method `change_symbol` defines the change propagation. It iterates over all elements of the implicit link `UsedBy` that is inherited from class `NameInST` and thus refers to using symbols. It then invokes the method `react_on_change` for each element of the implicit link. Since the base type of the implicit link is `UsingNameInST` rather than `UsingSymbol` and `react_on_change` is only exported by `UsingSymbol`, we have to cast the type here.

```
INCREMENT SPECIFICATION DefiningSymbol;
METHODS
  METHOD ErrorId: ERROR;
    RETURN (#SymbolAlreadyDef);
  END ErrorId;

  METHOD change_symbol(new_symbol:STRING):BOOLEAN;
  BEGIN
    FOREACH sym:UsingNameInST IN SELF.UsedBy DO
      (<UsingSymbol>sym).react_on_change(new_symbol)
    ENDDO;
    RETURN(SELF.scan(new_symbol));
  END
END METHODS;
```

The interactions that define the tool commands are completely inherited from the predefined classes `TerminalIncrement`, `NonterminalIncrement`, `IncrementList`, `TerminalIncrementList` and `NonTerminalIncrementList`. In particular, `TerminalIncrement` defines an interaction for changing a terminal increment that has already been expanded. To ensure that the method `change_symbol` is invoked so as to arrange for change propagation, we must redefine the respective interaction and invoke the method `change_symbol` defined above. The tool-specific interaction `ChangeTerminal` defines this behaviour.

```
  INTERACTIONS
    INTERACTION ChangeTerminal;
    NAME "Change Symbol"
    SELECTED IS SELF
    ON ( NOT SELF.is_phylum() )
    VAR new_name:TEXT;
        errors:TEXT_SET;
    BEGIN
      new_name:= NEW TEXT(SELF.unparse());
      IF new_name.LINE_EDIT("Change symbol:") THEN
        IF NOT SELF.change_symbol(new_name.CONTENTS()) THEN
          errors:=NEW TEXT_SET(SELF.get_set_of_errors());
          errors.DISPLAY();
          ABORT
        ENDIF
      ENDIF
    END ChangeTheIdentifier;
  END INTERACTIONS
END INCREMENT SPECIFICATION DefiningSymbol;
```

9.3.2 Class Interface Editor

In the last subsection, we defined the ENBNF editor, which is a syntax-directed tool for a rather simple language and we demonstrated the appropriateness of GTSL for defining syntax-directed editing, static semantic checks, and even automatic correction of static semantic violations in terms of change propagations. In this subsection, we sketch the use of GTSL on a much higher scale for the definition of an interface editor for GTSL classes. The sublanguage for GTSL class interface definitions is far more complex than the ENBNF language. As we are able to construct a tool for this complex language with GTSL, we provide evidence that the approach developed in this thesis scales up.

9.3.2.1 ENBNF of Class Interface

Figure 9.7 displays the ENBNF of the class interface editor. It has been derived in a straightforward manner from the GTSL grammar that is given in Appendix A. It defines the context-free syntax of the language in terms of 69 ENBNF productions. We have again defined symbols that anticipate the later definition of static semantics. In particular, we have defined various terminal symbols for identifiers. The terminal GTSL classes that are derived from these terminal symbols will define most of the static semantics and inter-document consistency constraints for GTSL class interfaces.

9.3.2.2 Inheritance Diagram of Class Interface Editor

We have derived a GTSL class for each production of the ENBNF. The inheritance relationship among these classes as well as their relationship to predefined GTSL classes are displayed in Figure 9.8. The picture only presents an excerpt because the overall hierarchy is too complex to fit on a page. We have, therefore, depicted classes `NonterminalIncrement` and `TerminalIncrement` with an underlying shadow. This indicates that there is a hierarchy of classes below each of them. The terminal increment classes are important for static semantics and inter-document consistency definition. They are, therefore, displayed in Figure 9.9[1].

We have added abstract classes for the specification of properties that are common to multiple classes. For the purpose of defining the scoping rules of GTSL class interfaces, for instance, we have reused the three abstract classes `ScopingBlock`, `NameInST` and `UsingNameInST`, which were introduced in the last subsection. Types in GTSL class interfaces can be declared by three different means: the class name (`ClassName`) can be used as a type, any super class (`SuperClass`) can be used as a type and any imported class (`ImportClass`) can be used as a type. We have, therefore, introduced an abstract class `ClassDecl` that serves as super class for the above three increment classes. Moreover, we have added abstract classes to serve as super classes for the different kind of method definitions that can appear in GTSL class interfaces. In this way we are able to exploit inheritance in GTSL to define the common properties of GTSL methods in a single class.

[1]For the non-terminal classes, we refer to Figure C.1 in Appendix C.2.

```
Interface          ::= AbstractInterface | NonterminalInterface | TerminalInterface | NSCInterface .
AbstractInterface  ::= "ABSTRACT" "INCREMENT" "INTERFACE" ClassName ";"
                        InheritSection ImportInterface AbstractExpInt
                       "END" "ABSTRACT" "INCREMENT" "INTERFACE" ClassName "." .
NonterminalInterface::="NONTERMINAL" "INCREMENT" "INTERFACE" ClassName ";"
                        InheritSection ImportInterface NonterminalExpInt
                       "END" "NONTERMINAL" "INCREMENT" "INTERFACE" ClassName "." .
TerminalInterface  ::= "TERMINAL" "INCREMENT" "INTERFACE" ClassName ";"
                        InheritSection ImportInterface TerminalExpInt
                       "END" "TERMINAL" "INCREMENT" "INTERFACE" ClassName "." .
NSCInterface       ::= "INTERFACE" ClassName ";"
                        InheritSection ImportInterface NSCExpInterface
                       "END" "INTERFACE" ClassName "." .
InheritSection     ::= "INHERIT" SuperClassList .
SuperClassList     ::= {SuperClass}(",") ";" .
ImportInterface    ::= | "IMPORT" "INTERFACE" ImportList "END" "IMPORT" "INTERFACE" ";" .
ImportList         ::= {Import} .
Import             ::= "IMPORT" ImportClass ";" .
AbstractExpInt     ::= "EXPORT" "INTERFACE"
                        AbstractSyntax Attributes SemanticRelations MethodSection
                       "END" "EXPORT" "INTERFACE" ";" .
NonterminalExpInt  ::= "EXPORT" "INTERFACE"
                        AbstractSyntax UnparsingScheme Attributes SemanticRelations NonterminalMethodSection
                       "END" "EXPORT" "INTERFACE" ";" .
TerminalExpInt     ::= "EXPORT" "INTERFACE"
                        RegularExpression Attributes SemanticRelations TerminalMethodSection
                       "END" "EXPORT" "INTERFACE" ";" .
NSCExpInterface    ::= "EXPORT" "INTERFACE"
                        Construction MethodSection
                       "END" "EXPORT" "INTERFACE" ";" .
Construction       ::= | "CONSTRUCTION" AttributeList "END" "CONSTRUCTION" ";" .
AbstractSyntax     ::= "ABSTRACT" "SYNTAX" ChildIncrementList "END" "ABSTRACT" "SYNTAX" ";" .
ChildIncrementList ::= {ChildIncrement}(";") .
ChildIncrement     ::= EntityName ":" UsingTypeDecl .
UnparsingScheme    ::= "UNPARSING" "SCHEME" UnparsingItemList "END" "UNPARSING" "SCHEME" ";" .
UnparsingItemList  ::= {UnparsingItem}(",") .
UnparsingItem      ::= PrettyPrinting | RegDef | Format | Component .
Component          ::= UsingEntity Delimiter .
Delimiter          ::= | "DELIMITED" "BY" DelimiterItemList "END" .
DelimiterItemList  ::= {DelimiterItem}(",") .
DelimiterItem      ::= PrettyPrinting | RegDef | Format .
RegularExpression  ::= "REGULAR" "EXPRESSION" RegExp "END" "REGULAR" "EXPRESSION" ";" .
Attributes         ::= | "ATTRIBUTES" AttributeList "END" "ATTRIBUTES" ";" .
AttributeList      ::= {AttributeDefinition}(";") .
AttributeDefinition::= AttributeCategory EntityName ":" UsingTypeDecl .
AttributeCategory  ::= | "HIDDEN" .
SemanticRelations  ::= | "SEMANTIC" "RELATIONSHIPS" SemanticRelList "END" "SEMANTIC" "RELATIONSHIPS" ";" .
SemanticRelList    ::= {SemanticRel}(";") .
SemanticRel        ::= ExplicitLink | ImplicitLink .
ExplicitLink       ::= EntityName ":" LinkType .
ImplicitLink       ::= "IMPLICIT" EntityName ":" "SET" "OF" UsingClass "." UsingEntity .
LinkType           ::= UsingType | UsingSetType .
MethodSection      ::= "METHODS" MethodList  "END" "METHODS" ";" .
MethodList         ::= {Method} .
Method             ::= DeferredMethod | HiddenMethod | ExplicitMethod .
NonterminalMethodSection::="METHODS" NontermMethodList  "END" "METHODS" ";" .
NontermMethodList  ::= {NonterminalMethod} .
NonterminalMethod  ::= NontermImpMethod | HiddenMethod | ExplicitMethod .
NontermImpMethod   ::= "IMPLICIT" "METHOD" NontermImpMethName "(" ParameterList ")" ResultType ";" .
TerminalMethodSection:"METHODS" TerminalMethodList  "END" "METHODS" ";" .
TerminalMethodList ::= {TerminalMethod} .
TerminalMethod     ::= TerminalImpMethod | HiddenMethod | ExplicitMethod .
DeferredMethod     ::= "DEFERRED" "METHOD" MethName "(" ParameterList ")" ResultType ";" .
TerminalImpMethod  ::= "IMPLICIT" "METHOD" TerminalImpMethName "(" ParameterList ")" ResultType ";" .
HiddenMethod       ::= "HIDDEN" "METHOD" MethName "(" ParameterList ")" ResultType ";" .
ExplicitMethod     ::= "METHOD" MethName "(" ParameterList ")" ResultType ";" .
ParameterList      ::= | {Parameter}(";") .
Parameter          ::= ParName ":" UsingTypeDecl .
ResultType         ::= | ":" UsingTypeDecl .
UsingTypeDecl      ::= MultiValue UsingType .
UsingType          : '[A-Za-z][A-Za-z0-9$_]*' .
MultiValue         : | '("LIST"|"SET"|"BAG"|"DICTIONARY")[ ]+"OF"[ ]+'
Format             : '\"((\\\")|([^"]))*\"' .
RegExp             : '"{"[^}]*"}"' .
PrettyPrinting     : '(\(\"[^\\^\"\*\+]+"\))|"(NL)"' .
ClassName          : '[A-Za-z][A-Za-z0-9$_]*' .
SuperClass         : '[A-Za-z][A-Za-z0-9$_]*' .
ImportClass        : '[A-Za-z][A-Za-z0-9$_]*' .
EntityName         : '[A-Za-z][A-Za-z0-9$_]*' .
ParName            : '[A-Za-z][A-Za-z0-9$_]*' .
MethName           : '[A-Za-z][A-Za-z0-9$_]*' .
TerminalImpMethName: '[A-Za-z][A-Za-z0-9$_]*' .
NontermImpMethName : '[A-Za-z][A-Za-z0-9$_]*' .
UsingEntity        : '[A-Za-z][A-Za-z0-9$_]*' .
UsingClass         : '[A-Za-z][A-Za-z0-9$_]*' .
```

Figure 9.7: ENBNF of the Class Interface Editor

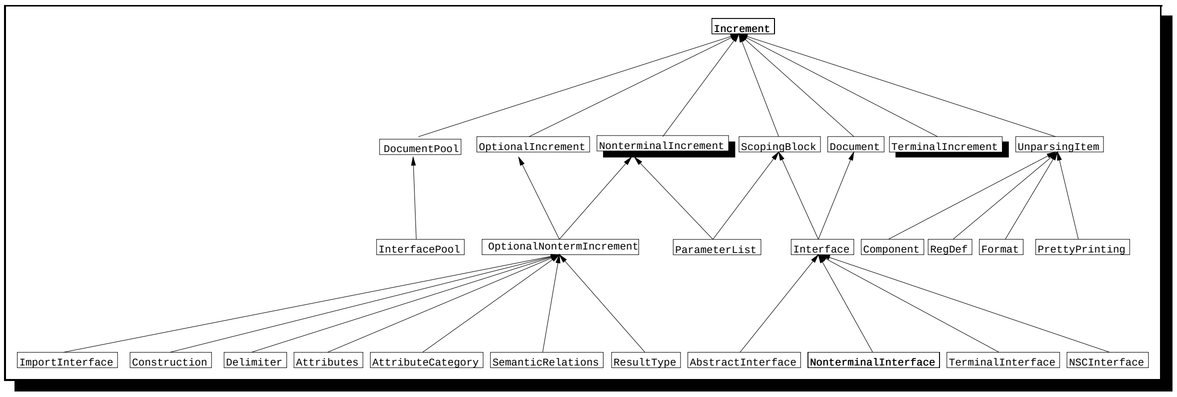


Figure 9.8: Top-Level Diagram of Inheritance Diagram

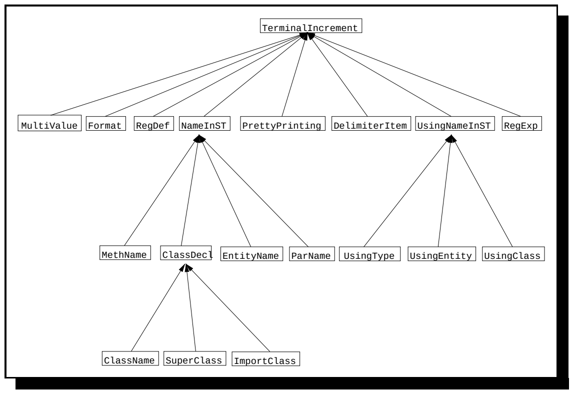


Figure 9.9: Inheritance Hierarchy of Terminal Increment Classes

9.3.2.3 Entity Relationship Model of Class Interface Editor

In order to define the abstract syntax children and semantic relationships we now consider the entity relationship model of the class interface editor specification[2]. The entity relationship model consists of a hierarchy of seven entity relationship diagrams. The top-level diagram defines the classes and relationships for class names, inheritance sections and import interfaces. Moreover, it includes two subsystems `Interfaces` and `ExportInterfaces`. Subsystem `Interfaces` defines the four different root increment classes that can appear in GTSL. Subsystem `ExportInterfaces` defines the different sections that can appear in export interfaces of GTSL class interface definitions. It contains further subsystems. Subsystem `RelationDefinitions` defines the structure of semantic relationships, `UnparsingItems` defines the structure of unparsing schemes and finally `UsingDecls` determines the various type definitions that can appear as property types, method result types and parameter types. Figure 9.10 displays the top-level diagram.

The top-level diagram defines that the newly introduced abstract increment class `ClassDecl` is a subclass of the reused class `NameInST`. `ClassDecl` thus inherits the semantic relationship `Block/IncludedNames` with `ScopingBlock` from class `NameInST`. Subclasses of `ScopingBlock` are defined in subsystems `Interfaces` and `ExportInterfaces`. A further semantic relationship is `InheritFrom/PassOnTo`, which is defined between class `SuperClass` and `ClassName`. This relationship stores the inheritance hierarchy of GTSL class definitions. Furthermore, the

[2]Again, we only sketch an excerpt here and refer to Appendix C.2 for the full definition.

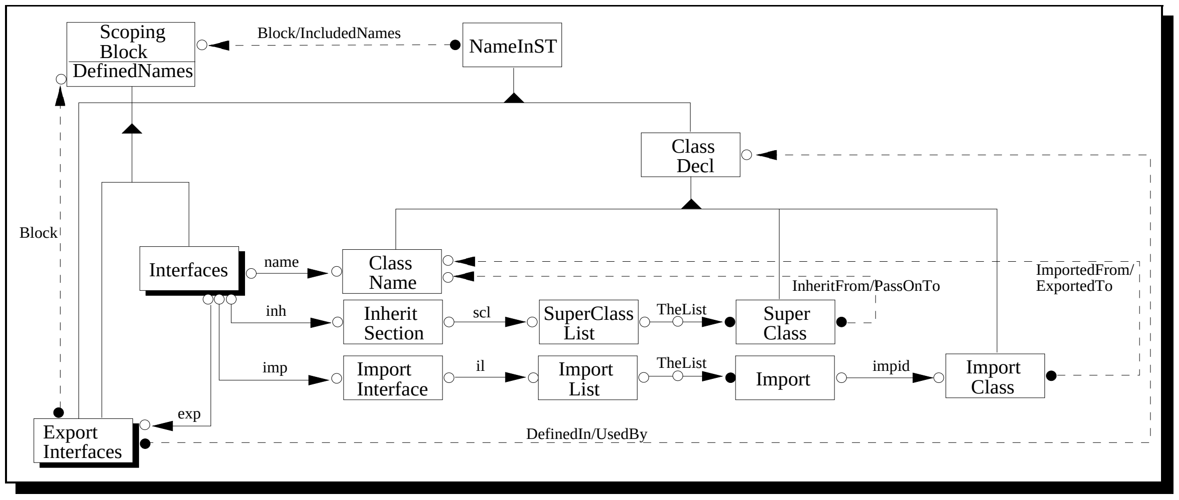


Figure 9.10: Top-Level Diagram of Entity Relationship Model

semantic relationship `ImportedFrom/ExportedTo`, which is defined between class `ClassImport` and `SuperClass`, stores the import/export relationships between classes. Figure 9.11 then depicts the refinement of subsystem `ExportInterfaces`.

This diagram contains as ports all classes from the top-level diagram that are related to the subsystem. In addition, it contains four ports that stem from the brother diagram `Interfaces` and define how the relationship `exp` from the top-level diagram is refined. Moreover, the diagram defines that subsystem `MethodDefinitions` contains a class that inherits from `ScopingBlock`. This class is `ParameterList`, which defines the nesting of scopes in GTSL class interfaces.

The main purpose of the diagram in Figure 9.11 is to define the composition of the different export interfaces of GTSL classes. It defines, for instance, that a terminal export interface (`TermExpInt`) has abstract syntax children for the attribute section (`Attributes`), the regular expression section (`RegularExpression`) and the semantic relationship section (`SemanticRels`), but does not have an abstract syntax section or an unparsing scheme. Moreover, it defines that attributes, semantic relationships and abstract syntax children have an abstract syntax component that stores the name (`EntityName`) of the respective declaration. In addition, all of them have a component `type` that is defined in the subsystem `UsingDecls`. This subsystem also contains a class that has a semantic relationship `DefinedIn/UsedBy` with the abstract increment class `ClassDecl`. Moreover, it contains a class having a semantic relationship with class `ScopingBlock`. The refinement of subsystem `UsingDecls` is depicted in Figure 9.12.

On the left-hand side, the diagram depicts the different classes as ports that have an abstract syntax child `type` of class `UsingTypeDecl`. A `TypeDecl` can be single- or multi-valued. This is determined by class `MultiValue`. Moreover, `UsingTypeDecl` has a child of class `UsingType`. This is a terminal increment that inherits from the abstract class `UsingNameInST`. It can access the symbol table that stores the declared types via the inherited relationship `Block`. It can then instantiate the semantic relationship `DefinedIn/UsedBy` to an instance of a subclass of `ClassDecl`. Two further classes are defined as subclasses of `UsingNameInST`. `UsingClass` is an abstract syntax child of an implicit link and also has a semantic relationship with a class declaration. Moreover, `UsingEntity` is an abstract syntax child of `Component`, which, in turn, is a child of an unparsing item list. `UsingEntity`, therefore, has a semantic relationship with `EntityName` to keep track of the declaration of the property.

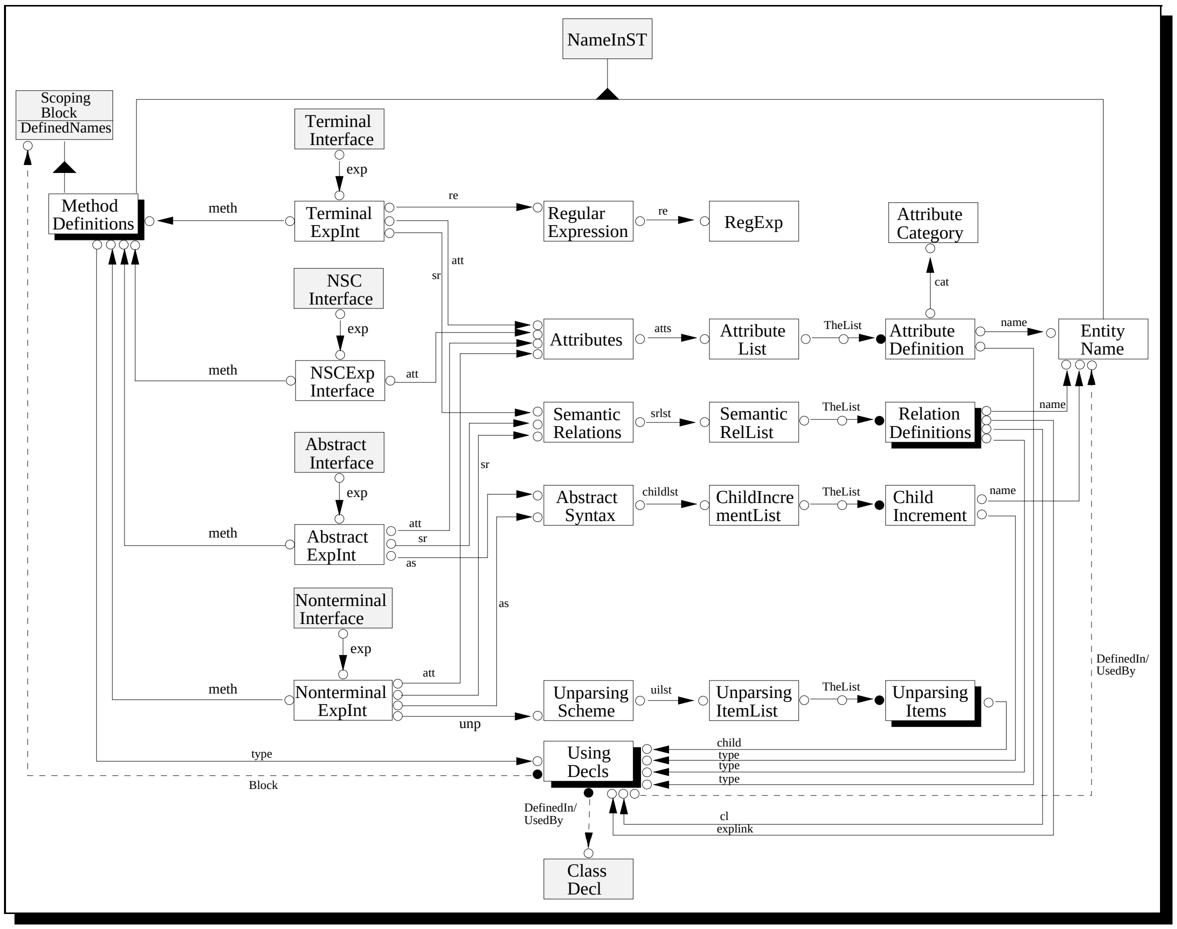


Figure 9.11: Refinement of Subsystem `ExportInterfaces`

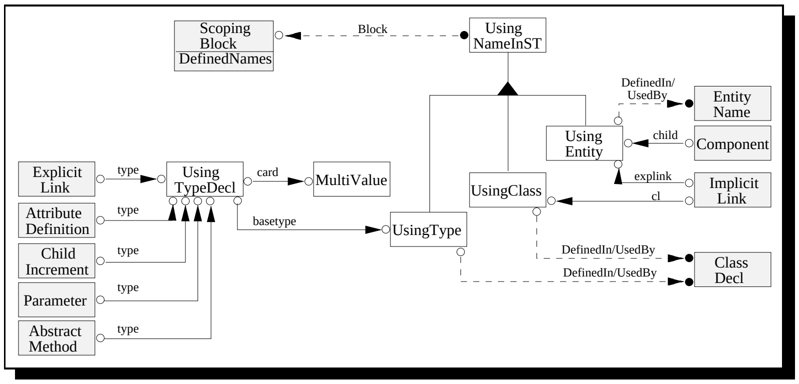


Figure 9.12: Refinement of Subsystem `UsingDecls`

9.3.2.4 Static Semantics

The implementation of scoping rules is inherited from classes `ScopingBlock`, `NameInST` and `UsingNameInST`. All increments of all subclasses of `NameInST` represent unique identifiers in their respective block. All instances of subclasses of `UsingNameInST` are checked for the existence of an increment that matches the definition. In these subclasses, we only have to redefine the deferred methods that compute the path to the respective scoping block, compute tool-specific error descriptors and implement reactions to change propagations.

On the basis of the definition of these scoping rules, we can then define typing rules. As an example consider static semantic Constraint A.21, defined in the Appendix A.2.1.1 on Page 273. It requires component items in unparsing item lists to denote abstract syntax children. This constraint is implemented in the following semantic rule of class `UsingEntity`.

```
INCREMENT SPECIFICATION UsingEntity;
...
  SEMANTIC RULES
  ON CHANGED(DefinedIn)
  ACTION
    IF (DefinedIn != NIL) AND
      (DefinedIn.father.IS_OF_CLASS("ChildIncrement")) THEN
        SELF.clear_error(#NotAnAbstractSyntaxChild) ELSE
        SELF.append_error(#NotAnAbstractSyntaxChild)
      ENDIF
    ENDIF
  END ACTION;
END SEMANTIC RULES;
...
END INCREMENT SPECIFICATION UsingEntity.
```

Note that the previously defined entity relationship model is particularly useful for understanding the path expressions that appear in this semantic rule. According to Figure 9.12, `DefinedIn` in class `UsingEntity` denotes the link with an increment of class `EntityName`. The relationship is established under control of a semantic rule that is inherited from class `UsingNameInST` (c.f. Page 218). The rule performs a symbol table lookup in order to obtain a defining name increment that has the same lexical value as the using name. If no such increment exists, the rule inserts an error descriptor into the set of errors and assigns the undefined value of `NIL` to the link. If it exists, the target increment `DefinedIn` could also be the name of an attribute or a link. This can be seen from the entity relationship diagram in Figure 9.11, since three aggregation relationships lead to `EntityName`. We, therefore, have to check whether the abstract syntax father of the target increment of `DefinedIn` is of class `ChildIncrement`. In that case, we remove the respective error descriptor, otherwise we insert it into the set of errors for the using entity increment.

Similar rules are defined for the other constraints. Most of them, however, also require a check for consistency with increments contained in other class definition documents. Checking for correctness of covariant redefinition or for the match between implicit and explicit links, for instance, means establishing inter-document relationships such as `InheritFrom/PassOnTo` or `ImportFrom/ExportTo`. We, therefore, consider inter-document consistency now.

9.3.3 Implementation of Inter-Document Consistency Constraints

9.3.3.1 Inter-Document Consistency to other Class Interfaces

To establish semantic relationships between increments of different documents that are of the same type, we need an increment that serves as a common root for all existing documents of that type. We refer to these increments as *document pools* in the following. The main purpose of this root increment is to provide access to all documents. GTSL, therefore, contains a predefined abstract increment class `DocumentPool`. Every document has a semantic relationship `docPool/registeredDocuments` with a document pool. Upon creation of a document, the relationship is instantiated by assigning the document pool increment to the explicit link `docPool`. The implicit link `registeredDocuments` then contains references to all documents.

Queries for a particular document in the document pool are most often associative and then the `registeredDocuments` link is inappropriate. We would have to iterate over the link with a `FOREACH` statement, which is neither concise in specification nor efficient during execution. To define these associative queries more appropriately, `DocumentPool` defines a symbol table attribute `definedDocuments`. The predefined operators `includes` and `increment_at` may then be used for associative access to the set of documents. The associative queries are usually based on a document name, which refers to the lexical value of some increment contained in the document. Obviously, the name is tool-specific and, therefore, cannot be predefined. Instead, tool builders have to define a subclass of `DocumentPool`, which defines semantic rules for determining the contents of the symbol table. Below, there is an example taken from the GTSL class editor specification. It defines the semantic rules of class `InterfacePool`, which is a subclass of `DocumentPool`.

```
INCREMENT SPECIFICATION InterfacePool;
  ...
  SEMANTIC RULES
   ON EXISTS(doc: Interface IN SELF.registeredDocuments):CHANGED(doc.name.value)
    ACTION
      SELF.definedDocuments.associate(doc.name.value, doc);
    END ACTION;

   ON EXISTS(doc: Interface IN SELF.registeredDocuments):DELETED(doc)
    ACTION
      SELF.definedDocuments.deassociate(doc.name.value);
    END ACTION;
  END SEMANTIC RULES;
```

The rules define that the contents of the symbol table `definedDocuments` are indexed with lexical values of class names. This is appropriate because we also refer to class names in super class definitions and import statements. The first semantic rule is executed whenever a class interface name is created or changed. To define this dependency, the `ON` clause contains an `EXISTS` predicate that is defined via the implicit link `registeredDocuments`. The body of the semantic rule associates the document, which was unified with the `EXISTS` predicate, with the lexical value of the class name in `definedDocuments`. Upon deletion of a document, the reverse operation is performed by the second semantic rule; the association between the class name and the class document is deleted from symbol table `definedDocuments`.

The document pool can now be exploited for the definition of inter-document consistency constraints. As an example consider the constraint that each super class included in an inherit section must be the name of some other class. This constraint is implemented in the semantic rule of class `SuperClass` below.

```
INCREMENT SPECIFICATION SuperClass;
...
SEMANTIC RULES
    ON CHANGED(myDocument.docPool.definedDocuments) OR
       CHANGED(SELF.value)
    VAR i:Interface;
    ACTION
      i:=<Interface> myDocument.docPool.definedDocuments.increment_at(value);
      IF i!=NIL THEN
        InheritFrom:=i.name;
        Errors.clear_error(#UnknownClass);
      ELSE
        InheritFrom:=<ClassName>NIL;
        Errors.append_error(#UnknownClass);
      ENDIF
    END ACTION;
  END SEMANTIC RULES;
...
```

The rule is fired as soon as either the symbol table attribute in the document pool is changed or the lexical value of the super class name is created or changed. If it is fired, it performs a symbol table lookup and searches for a class that has the same name as the lexical value of the super class increment contained in the inherit section. If such a class exists, the semantic relationship `InheritFrom/PassOnTo` is established by assigning the class name increment that is reached from the document via the abstract syntax child `name` to the explicit link `InheritFrom`. In that case, the error descriptor `#UnknownClass` is removed from the set of errors, otherwise the semantic relationship is deleted and the error descriptor is added to the error set.

In the same style, a semantic rule in class `ImportClass` controls the semantic relationship `ImportedFrom/ExportedTo`. `InheritFrom/PassOnTo` and `ImportedFrom/ImportedTo` are then exploited during a number of inter-document consistency checks. To check whether a used entity is inherited from some class, for instance, we can now traverse to the set of inherited classes along the link `InheritFrom`.

9.3.3.2 Inter-Document Consistency to other Document Types

In order to check and preserve inter-document consistency with documents of other types, we have to import the respective document pool into the tool configuration of the tool that checks and preserves the constraint. We can then associatively access the document pools that store documents of other types. In addition, we have to import classes in order to use them as types of semantic relationship links or in path expressions. Moreover, classes of the imported tool configuration must export methods that enable the other tool to perform change propagations, for instance.

As an example, we now consider the inter-document (type) consistency constraints between ENBNFs and class interfaces. Consistency constraints that have been discussed in Chapter 7 are implemented in terms of change propagations from the ENBNF editor into the class definition editor. Therefore, class `InterfacePool` exports method `CreateDocument` and `DeleteDocument` to create and delete class interfaces respectively. These operations are invoked whenever a symbol on the left-hand side of a production is created or deleted. Moreover, the interface editor classes export operations to

- modify a class name (`ChangeClassName`),
- add an inheritance relationship (`AddInheritance`),
- create an abstract syntax child (`CreateChild`),
- delete an abstract syntax child (`DeleteChild`),
- create a regular expression definition (`CreateRegexp`) and
- change a regular expression definition (`ChangeRegexp`).

To actually import these operations into the ENBNF editor configuration the class interface editor configuration must export the classes that export the operations. This is achieved by the following tool configuration definitions. They represent relationships between subsystems of the environment's entity relationship diagram.

```
CONFIGURATION INT                CONFIGURATION SV
...                              ...
   EXPORT                           IMPORT FROM CONFIGURATION INT:
    INCREMENT CLASSES:                INCREMENT CLASSES
      InterfacePool,                    InterfacePool,
      Interface,                        Interface,
      NonterminalInterface,             NonterminalInterface,
      TerminalInterface,                TerminalInterface,
      AbstractInterface;                AbstractInterface;
   END EXPORT;                      END IMPORT;
...
END CONFIGURATION INT
```

Then these imported classes can be used in increment classes of the ENBNF editor specification to check and preserve inter-document consistency. As an example let us revisit the class `DefiningSymbol`. It defined a method `change_symbol` that was invoked whenever a symbol on the left-hand side was created. This symbol can now be changed in order to also invoke method `rename_class` from class `Interface` to propagate the change of the symbol identifier to the class that corresponds to the symbol:

```
METHOD change_symbol(new_symbol:STRING):BOOLEAN;
VAR depending_int:Interface;
BEGIN
  depending_int:=intPool.definedDocuments.increment_at(SELF.value);
  IF depending_int!= NIL THEN
    depending_int.ChangeClassName(new_symbol);
  ENDIF;
  FOREACH sym:UsingNameInST IN SELF.UsedBy DO
    (<UsingSymbol>sym).react_on_change(new_symbol)
  ENDDO;
  RETURN(SELF.scan(new_symbol));
END
```

Method `ChangeClassName` then, in turn, changes the class name of the GTSL interface class. Before actually changing it, it exploits the semantic relationships discussed above in order to propagate the change further to all increments such as imports or inheritance lists which use the class name. In this way static semantic correctness of GTSL class interfaces is likewise unaffected by the change.

9.3.3.3 Inter-Document Consistency Constraints in the BA SEE

Tool-specific Services: For the integration of the Booch tool with the class definition tool an integration strategy different from the one in the GENESIS environment has to be chosen. The reason is that the Booch tool is a "foreign" tool, which does not store its documents as abstract syntax graphs in O_2. Therefore, inter-document reference edges cannot be used as a basis for tool integration. Instead, we define a number of tool-specific services. The Booch tool will then exploit the message router to send messages to the class definition tool. Upon receipt of such a message, the class definition tool will, in turn, translate the message into a call of a method defined in the schema. This means that we have to implement tool-specific services as methods in GTSL, which the GTSL compiler will then translate into methods of the tool's schema.

```
INCREMENT SPECIFICATION DataMemberList;
...
METHOD AddAggregation(TheClass : STRING;
                      MemberName : STRING;
                      ByPointer : BOOLEAN;
                      StaticMember : BOOLEAN);
VAR i : DataMember;
    s : STRING;
BEGIN
  IF (StaticMember) THEN
    s := CONC("static ", TheClass);
  ELSE
    s := TheClass;
  ENDIF;
  IF (ByPointer) THEN
    s := CONC(s, " * ");
  ENDIF;
  s := CONC(s, " "); s := CONC(s, MemberName); s := CONC(s, ";");
  i := NEW DataMember(SELF); i := i.parse(s);
  IF (NOT expanded) THEN
    SELF.expand();
  ENDIF;
  TheList.ADD_LAST(i);
END AddAggregation;
```

Figure 9.13: Method Implementing a Tool-specific Service

As an example of such a method, consider method `AddAggregation` in Figure 9.13. The purpose of the method is to implement the tool-specific service that reacts to the creation of a new has-relationship in the Booch diagram. It is invoked with a number of parameters that specify the relationship in detail. The first parameter identifies the target class of the has-relationship and thus determines the type of the data member that is to be inserted. The next parameter determines the data member name. Two further parameters are used to pass adornments of the relationship. They determine whether the instance variable is a pointer and whether it is a static class member. The body of the method then interprets these parameters and computes an equivalent string representation. The string is then parsed by means of the implicit method `parse` of class `DataMember`. The parse method returns a reference to the abstract syntax tree of the data member root node. `AddAggregation` then checks if the data member list is still a place holder. If so it is expanded and then the reference to the data member is inserted into the data member list.

We note that the tool specification languages as we suggested them in this thesis are incomplete. What is missing is a primitive for defining messages, their components and their synchronisation and routing properties. Therefore, the tool-specific message classes of the software process communication protocol subsystem (c.f. Page 100) cannot be defined in GTSL. For the implementation of the actual integration we had to hand-code them in C++. Moreover, we cannot define message interpretation, i.e. the binding of a message to a GTSL method. The problem is currently being addressed in a Master's thesis [Wag95]. The proposition in this thesis is to declare messages together with their components, properties and their binding to methods of the root increment class in the tool configuration. The root increment class methods may then use GTSL path expressions to invoke methods of other GTSL classes.

The decision to have, with explicit methods, a low-level, yet flexible primitive for defining abstract syntax graph traversals and modifications proves to be very appropriate here. Methods may be substituted if more abstract integration patterns can be found. This, however, requires more experience in a-posteriori integration. We would not have been able to define this a-posteriori tool integration without the flexibility provided by GTSL methods.

Moreover, the example strongly suggests that message passing between tools, which is classified by Wassermann as a control integration primitive can also be used for data integration purposes. The two terms *control integration* and *data integration* [Was89] are, therefore, not at all orthogonal, but rather two sides of the same coin.

Common Conceptual Schema: To implement tool integration between the class definition, method implementation and IPF documentation tool, we choose yet another strategy. The syntax graphs for a C++ class definition, the method implementation and the IPF documentation are, from a structural point of view, very similar and, therefore, contain a significant amount of redundant information. For each class of a library, subgraphs for method signatures, for instance, are included redundantly in all three graphs. The British Airways process model allows us to avoid this redundancy. Since the same user is responsible for all three documents, it is most appropriate if a change to a class definition is immediately reflected in the method implementation and IPF documentation. We can, therefore, define a common conceptual schema for all three tools. The schema defines the superposition of the different syntax graphs. The different tools then only use a subset of the definitions defined by a view of the conceptual schema.

Here we encounter a further weakness of our tool specification language. It does not yet include primitives to define common conceptual schemas and views on top of them. These two primitives are currently being added in a further Master's thesis [Bec95]. We can, nevertheless, exploit the tool specification languages as they were defined in this thesis for the construction of these tools. We define one tool specification from which the abstract syntax graph structure of the conceptual schema is generated. Then we define further tool specifications as views for each of the three tools. Compared with the conceptual schema specification, these tool specifications have an identical class hierarchy, the same abstract syntax, regular expression, attribute, semantic relationship, method and semantic rule sections as the conceptual schema. We merely modify interactions and unparsing schemes in a tool-specific way. Compared with a full view mechanism as proposed in [Bec95], consistency between views and conceptual schema, however, has to be checked manually. Moreover, we cannot hide particular definitions from being accessed from one or the other view and a view cannot include specific methods for a tool.

9.3.4 Quantitative Analysis of the Specification Sizes

We now discuss how much time has to be spent in order to construct a tool using the languages presented in this thesis. Table 9.1 displays the result of a quantitative analysis that we performed with the ENBNF, GTSL class interface and C++ class definition tool specifications.

	GTSL ENBNF	GTSL Class Interface	C++ Class Interface
Classes	22	74	100
Interactions	1	2	49
Explicit Methods	8	17	45
Semantic Rules	5	14	47
Tool-Specific Services	0	0	15
Size of Interfaces (LOC)	665	2,600	3,400
Size of Specifications (LOC)	265	806	6,400
Total Size (KBytes)	23	89	273
Generated Code Size (MBytes)	3.7	8.2	16.8
Ratio	160:1	92:1	62:1

Table 9.1: Size of Tool Specifications

The number of classes required for a tool specification is dominated by the number of productions in the ENBNF of the respective language. In the ENBNF of the ENBNF tool specification there are 18 productions and four further classes were added for the specification of static semantic properties. Likewise, the ENBNF of the GTSL class interface tool consists of 69 productions, five further abstract classes were added to specify static semantic properties. The C++ ENBNF consists of 87 productions and 13 abstract classes were added to the derived ones to specify static semantics and inter-document consistency.

The average size of a class specification depends largely on the number of tool-specific commands and services that have to be defined. Class definitions in the ENBNF tool specification have a size of less than 1,000 Bytes. The reason is that only one command is specified in an interaction, but any other interaction is inherited from predefined classes. In the C++ tool, however, a number of interactions had to added in order to meet the specific requirements of British Airways, for instance, with respect to browsing. Moreover, a number of methods had to be added to implement tool-specific services. The average size of GTSL classes in that specification has, therefore, been increased to 2,730 Bytes.

If we assume that the code generated by the GTSL compiler is as compact as hand-coded code, the ratio between the size of the generated code and the size of the specification gives evidence of the advantages of using our approach compared with the implementation of tools with object-oriented programming languages. The ratio merely depends on the size of the GTSL class specifications. The ratio is higher for tools, like the ENBNF tool, that do not need to have specific features than it is for tools with specific commands and services. In reality, a tool builder might not save as much time as indicated by these ratios because he or she might be able to write code that is twice or three times as compact as the generated code, but still there is a significant benefit in using our approach.

9.3.5 Summary

In this section, we have evaluated how the various tool-specific concerns can be specified with the languages that we suggested. We have shown the appropriateness of ENBNFs to define the abstract and concrete syntax of a language. Then we have indicated how entity relationship diagrams can be used to define further structural properties, namely semantic relationships and attributes. We have seen how well they visualise the navigation paths that are then used in the behavioural specification of static semantics and inter-document consistency constraints. Finally, we have seen how tool commands and tool-specific services can be defined in GTSL.

We have exploited the various facilities that the language offers for component reuse. Reuse of properties is mainly based on inheritance. Unlike object-oriented programming languages where only attributes and methods are inherited, our languages support also inheritance of semantic relationships, semantic rules and interactions. Reuse is not confined to predefined specification components, but tool builders can specify their own components for reuse purposes, like we defined class `ScopingBlock`. Moreover, reuse is supported since arbitrary definitions can be customised by redefining them in subclasses. We exploited this, for instance to adapt the definition of an interaction that was inherited from a predefined class.

The object-oriented paradigm that we used for the language definitions not only contributes to the reusability of specification components, but also guides tool builders to structured tool specifications. A complex tool specification is structured into manageable component specifications in terms of GTSL increment classes. Structuring is not only supported for GTSL classes, but also on a more coarse-grained level for subsystems in our entity relationship notation. Due to these structuring facilities tool specifications, even for so complex tools as the C++ class definition tool or the GTSL class interface tool, are still manageable. Thus the fact that we have been able to construct these tools provides evidence that our approach scales up.

We have also measured the effort that is required to construct so complex tools. The effort merely depends on the number of increments in the language grammar, the number of static semantic constraints and the number of tool-specific commands. The effort is small compared to the effort that is required when hand-coding the tool with object-oriented schema definition and programming languages.

9.4 Performance of Tool Execution

In order to evaluate the performance of generated tools we conduct an experiment with the C++ class definition tool. Of the tools discussed in this chapter, the C++ class editor is the most complex. We have chosen this tool as a platform for our experiment because it is also among the most complex that will occur in practice. Therefore, response times that we evaluate for the C++ class definition can be considered as the worst case. During the experiment, we will execute a number of archetypical commands. They will be used to measure the performance of browsing, template expansion, static semantics and inter-document consistency checks, change propagations and parsing of freely input text. We first present the experiment in detail, then describe the measurement environment and finally discuss the results. While doing so, we focus on how well the response time requirements, outlined in Subsection 2.3.3, are met and we compare the results with the execution times of the Merlin Benchmark.

The experiment consists of seven actions that we describe below. As was the case with the benchmark measurements, the times we are interested in are the elapsed real times.

Unparsing: The action that dominates the response time during browsing is the time required to unparse an ASG representation, i.e. to load it from the database, compute a textual representation thereof and display it to the user in an editor window. The ASG that we use in this action represents the largest class definition we found in a library that we obtained from the British Airways reuse department. The textual representation has 220 lines of code and the respective syntax graph consists of 507 nodes.

Dumping: During this action we perform almost the same operation as during unparsing, but this time we do not present the textual representation in an edit window, but store it in an operating system file. We measure the time required for the computation of the representation and the dump.

Place Holder Expansion: During this action we set out to measure the time for template insertions that occur when place holders are expanded. These templates are typically inserted into or appended to list increments. As an archetypical example of this we measure the time that the tool needs to insert a parameter template into a list of parameters that currently includes three parameters. The time not only includes the required syntax graph modification, but also the time taken to insert new syntax graph nodes into the version unit and the time for redisplaying the contents of the affected window.

Static Semantic Check: During this action we explore the performance of commands that perform static semantic checks. As an example, we expand the name identifier of a previously expanded parameter. The time we measure includes checking the lexical correctness of the identifier, storing the value in a node attribute, checking the uniqueness of the identifier in the parameter list and incrementally redisplaying the affected window.

Inter-Document Consistency Check: The purpose of this action is to measure the performance of the creation of a new dependency relation between two documents. For that purpose we have chosen a forward declaration of a class. The measured time, therefore, includes the check as to whether the class referenced in the forward declaration exists, the creation of a semantic relationship between the class name and the forward declaration, the storage of the new dependency for configuration management purposes and the incremental redisplay of the affected document parts.

Change Propagation: The purpose of this action is to measure the performance of intra-document change propagations. We change the name of a type that is used within the document four times. The time required for that includes lexical analysis of the new type identifier, analysis of uniqueness, propagation of the change to all using types and incremental redisplay of the affected document parts.

Parsing: The purpose of this action is to measure the performance of the parsing that is required after free textual input. We chose parameter lists of methods that are, in C++, reasonably complex. The measured time includes syntactic analysis of a character string against the C++ grammar for parameter lists, semantic analysis for uniqueness of parameter names and existence of parameter types, creation of a syntax tree for the parameter list, creation of semantic relationships between parameter types and their declaration, inclusion of all nodes of the tree into the version unit and incremental redisplay of the parameter list. The string to be parsed includes three parameters. It is syntactically correct and one parameter type is undefined.

During the experiment, the tool and the database server were running on the same machine, a Sun SparcStation 10/40 with 64 MBytes of main memory. The database resided on a local 2 GBytes SCSI-II disk. The page cache of the database server was 3 MBytes and the object cache of the database engine was 6 MBytes large. The database was in a warm state, i.e. methods executed during actions had already been executed before. The actions were performed while no other user was accessing the database.

O_2 allows any of the ACID properties for a database server that serves tool sessions to be given up. As discussed on Page 73 it is unreasonable to give up the atomicity and durability property. We, therefore, only experimented with different concurrency control schemes. A database server in mode `NC_A_R`, does not perform locking and, therefore, only enables one session to access and modify a database. A server in `C_A_R` performs page-level locking as a concurrency control scheme. We experimented with several concurrent tool sessions and this mode proved to be inappropriate. It reveals unnecessary concurrency control conflicts when concurrent transactions access objects that by chance reside on the same page. A server in `OC_A_R` remedies the problem and performs page-level locking but switches to object-level locking as soon as a conflict is detected.

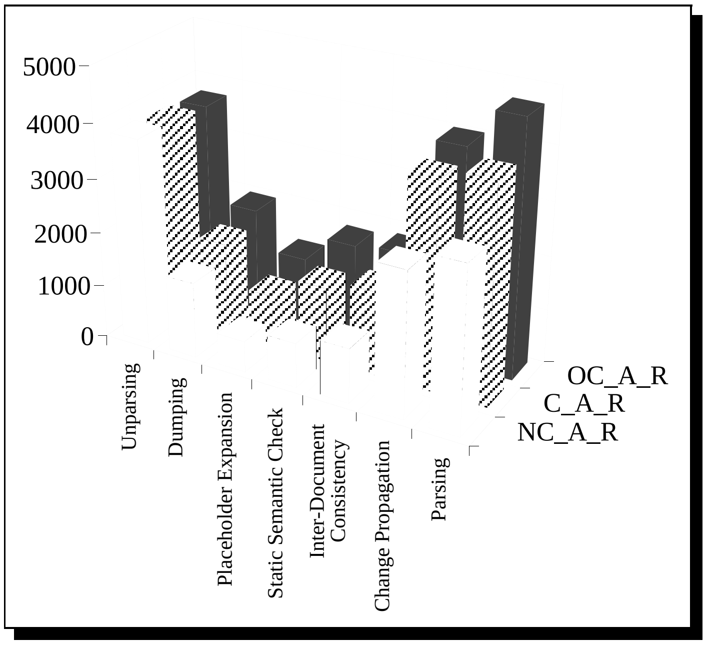


Figure 9.14: Response Time of Archetypical Tool Commands

Figure 9.14 displays the measurement results in milliseconds for each of the above actions in the three different concurrency modes. The front row represents the results without locking. The middle row represents page-level locking and the back row represents object-level locking.

The dumping of a large class definition, place holder expansion, static semantic checks and inter-document consistency checks perform in any mode in less than one and a half seconds, which is not as quick as we required but still reasonable. The operations are reasonably fast because we employed an incremental unparsing algorithm that only redisplays those parts of a document that were affected by a change.

Unparsing a complete document requires in any mode 3.8 seconds, which is too slow. It is worthwhile to note that the poor performance is not a problem of the database, but rather of how we used it in the layout computation class. In principle, the algorithm that is performed for unparsing is the same as that during dumping and the dump performs reasonably fast. There are two minor differences. Firstly, the user interface is not invoked at all during a dump and the representation is completely computed by O_2C methods. Secondly, the `Errors` attributes of syntax graph nodes are not accessed since static semantic errors and inter-document inconsistencies cannot be visualised in files. Knowledge of these differences can be exploited for the acceleration of unparsing. We did not take the complexity of algorithms executed in user interface classes into account since we assumed that they would only have a minor influence on the overall performance. This assumption now turns out to be wrong. Moreover, we can accelerate the decision whether to visualise an increment as erroneous. If the `Errors` attribute of an increment is clean and does not contain an error, which is the usual case, this could be stored in a boolean attribute of class `Increment`. In this case the `LayoutComputation` can use this attribute to decide whether to mark an increment as erroneous. The attribute value can be determined without additional costs during the propagation phase.

During parsing of a parameter list some 60 O_2 objects have to be created. They all have to be included in the version unit. Then local or even global symbol tables have to be queried six times for declarations and the use of identifiers. The use of three type identifiers has to be materialised in semantic relationships. Users might tolerate the response time of 2.8 seconds that we obtained in the mode without locking, but the performance of more than four seconds in the case with object-level locking is unreasonable. In the same way a performance of 2 seconds for change propagation might be tolerated because a user will require more time to perform this propagation with a conventional text editor, whereas 4.5 seconds with object-level locking is already on the border-line.

We recognise a significant performance decrease due to locking in all actions except unparsing. The performance of unparsing is independent of the locking mode in our experiment since all semantic rules were clean. Therefore, unparsing was performed as a read-only transaction that does not lock objects. It would also be influenced by the locking mode if there were dirty error attributes. To evaluate these, the tool would also have to start a transaction. Commands that are executed with page-level locking perform on an average about 30-40% slower compared with the mode that does not lock objects. Object-level locking, in turn, is about 10-25% slower than page-level locking. Without concurrency control but with atomicity and durability, commands for template expansion perform in half a second, which is even faster than we required. Static semantic checks and even inter-document consistency checks perform in about a second, which is fast enough. As argued on Page 29, isolation of a number of commands in PSDEs might be guaranteed by the process engine. Then we know from these commands that they cannot cause concurrency control conflicts. Unfortunately, we cannot exploit the `NC_A_R` mode. O_2 applies the mode to **all** transactions of a session but those commands that access increments of other documents must not be executed in this mode. The overhead for locking, in general, reinforces our concern regarding customisable concurrency control schemes. The concurrency control scheme should, therefore, be a property of transactions or actions as we required in Chapter 3 rather than a property of the database server execution.

It is interesting to note the differences of the above results from those we obtained with the Merlin Benchmark. When we executed the Merlin Benchmark with O_2, the object-level concurrency control was not available and the benchmark was executed in mode `C_A_R`. In fact, object-level locking is still a prototype implementation, developed during the GOODSTEP project, and the performance might be improved in a later product version. Unparsing was twice as fast during the Merlin Benchmark as dumping in this experiment. The reason is that in the Merlin Benchmark the syntax graphs were about half the size. Moreover, template expansion was faster than in our case. The reason here is that firstly, version management requires additional time for inserting newly created objects into a version unit and secondly, in the above experiment, the document representation is recomputed whereas during the Merlin Benchmark only the expansion was considered. Parsing and change propagations were not considered during the Merlin Benchmark.

9.5 Summary

In this section we have evaluated our tool construction approach. We assessed two different environments against the functional requirements that we discussed in Chapter 2. On the basis of the evaluation of these two scenarios, we revealed that tools constructed on top of an object database system meet our functional requirements.

The textual tools in the environments were specified with the languages suggested in this thesis and generated by the GTSL compiler. GTSL, as we presented it in this thesis, is capable of a-priori tool integration based on semantic relationships between increments of different document types. An extension to specify a-posteriori and view-based integration is on its way. We have indicated the effort that is required to specify a tool and have indicated the improvements compared to using an object-oriented programming language for tool constructions.

Our performance requirements are only partly satisfied. For template expansion, static semantic checks and inter-document consistency checks the performance is acceptable independent of the concurrency control protocol used by the database server. Commands that parse texts after free textual input or perform a change propagation perform only sufficiently fast if the database does not perform locking. The unparsing required during browsing is not fast enough, but this might be improved without having to change the O_2 database system. The database system should, however, be changed in order to permit sessions with activities and transactions.

Chapter 10

Summary and Open Problems

Summary

We have developed techniques to simplify the construction of sophisticated syntax-directed software development tools. In particular, we have defined GTSL, a family of languages that can be used to describe different concerns of a tool at different levels of abstraction. The languages support the structuring of specifications and address specification reuse in terms of inheritance. To simplify tool construction, we have suggested a library of reusable component specifications that solve common problems. We have delineated consistency constraints between the various languages. These consistency constraints were used as guidelines for integrating the tools of GENESIS, an environment for tool specification.

The languages and tools can now be used to accelerate the construction of integrated software development tools. The tools, in turn, are significantly improved compared with current state-of-the-art. We have demonstrated this on the basis of the environments that we constructed for evaluation purposes. In particular, tools can be defined in such a way that they check inter- and intra-type inter-document consistency constraints. They can even arrange for automatic constraint preservation in terms of change propagation, if appropriate. Tools ensure persistence of changes made during a command. This preserves user effort against hardware and software failures. At the same time it makes changes visible to concurrent users as soon as tool commands have been completed. Tight cooperation among developers is, therefore, no longer hampered by the isolation inherent to other editors. Tools constructed with our environment are rather group editors that support cooperative work.

Unlike group editors in CSCW, however, our tools may be driven by the process engine. Tools, therefore, offer a set of well-defined generic services that the process engine can use to perform certain interactive or off-line activities. In particular, the process engine can arrange for isolation of developers, if required, by means of the version management services offered by our tools.

The effective construction of tools offering these advances was only made possible by the database technology that became available recently. Using relational database systems, syntax graphs had to be squeezed into a set of tables with the result that version management could no longer be supported by the database system. Structurally object-oriented database systems overcome the problem of modelling graph structures. However, they cannot define access and modification operations for graphs within their schema. In addition, these systems suffer from a granularity problem because those that performed efficiently enough lacked functionality with respect to distribution and transaction management. Those that provided transaction management and distribution, however, performed too slowly. Object database systems solve these problems adequately. Abstract syntax graph structure and behaviour can be appropriately modelled in terms of classes. The versioning of subgraphs of a project-wide abstract syntax graph is supported by primitives for version management of composite objects. Object database systems support the structuring of the schema definition. Horizontal structuring is supported by schema-import and -export primitives. Vertical structuring is supported by views. Both mechanisms are suitable for tool integration purposes in order to develop tools independently. The schema update facilities of object databases simplify maintenance of tools because the way in which existing graphs can migrate into a new graph structure can be defined. Object databases offer ACID transactions, which are used to implement tool commands. Tool commands are, therefore, performed in isolation from concurrent commands and their effect is persistent as soon as the transaction is completed. To achieve acceptable tool performance, in particular during concurrent editing, database systems must enable activities to be performed without locking. Tools can access a set of documents stored in a central database distributedly. This is achieved by exploiting the multi-level client/server architecture of current object database systems.

Open Problems

The tool specification languages that we have suggested in this thesis cannot define graphical tools. In particular, they do not provide language primitives for the concrete syntax and the unparsing scheme of graphical languages. In [Ges95] a first attempt was made to extend our specification languages, the set of predefined classes and the tool architecture to cover graphical languages, too. The thesis, therefore, suggests a formal model considering graphical documents as graphs with atomic, expandable and hyper nodes that are connected with atomic and hyper edges. Atomic nodes only have a shape but are not further structured. They would, for instance, be used in data flow diagrams to model stores or terminators. Expandable nodes have a graphical subdiagram enclosed within the node. Examples occur in state charts, where a state can have enclosing states or in Nassi Shneiderman diagrams where a statement is enclosed in a statement list. Hyper nodes are refined by a sub-diagram. As an example, consider subsystems in our entity relationship model that are refined by a sub-diagram, to be displayed in a separate window and printed on a separate sheet of paper. Nodes within a diagram may be connected with atomic edges. A hyper node is connected by a hyper edge with its refining diagram. Then a set of extensions to our specification languages are suggested in order to define the syntax of graphical documents. A number of predefined classes are added to our class library in order to provide primitives for nodes, expand nodes, hyper nodes, edges and hyper edges. Finally, the architecture of our tools is extended to provide primitives at the user interface to deal with graphical documents. The results of [Ges95] remain to be integrated into the work described in this thesis, i.e. the language extensions suggested in the thesis must be implemented in our compiler and the class interface tool. The architectural changes must be merged into our tool architecture. Then we would also be able to bootstrap tools for inheritance and entity relationship diagrams for the GENESIS environment.

In Chapter 2, we have differentiated between generic and tool-specific services. The generic services were implemented within the tool architecture that we discussed in Chapter 5. Tool-specific services, however, need to be specified by the tool builder and our languages do not yet include all primitives for that purpose. To extend our tool specification language with primitives for messages is the subject of [Wag95], a forthcoming Master's thesis. The implementation of these primitives will then permit the generation of tools that can react to tool-specific service requests of a process engine or other tools.

In a companion PhD project [Jun95b], a set of high-level languages called ESCAPE [Jun95a] are defined for process modelling purposes. ESCAPE supports modelling document types and their relationships at a coarse granularity during the early stages of process modelling. Document types and relationships may need to be refined in terms of our ENBNF and entity relationship model in order to define the structure of an underlying project-wide abstract syntax graph. Hence, there is an obvious need to define the coarse-grained process model in an integrated way with the more fine-grained tool specification. In order to do so, the process modelling languages must be integrated with the tool specification languages in terms of inter-document consistency constraints. These constraints can then ensure that, for instance, relationships between document types that were identified in the process model are refined in terms of more fine-grained relationships between increments in the tool specification. If tool support is considered, the need arises to integrate the GENESIS and the PROMOTOR environment, which supports modelling in ESCAPE. This integration, in turn, will be simplified significantly by the fact that both environments already store their documents as syntax graphs in O_2.

The predefined increment class `DocumentVersion` provides the means for merging documents in the version history graph on the basis of primitives offered by the database system. We have not yet tackled the problem of merging the contents of different versions. An algorithm for merging different versions of an abstract syntax graph is suggested in [Wes91]. It is based on the assumption that the differences between the syntax graph versions to be merged are known. The O_2 database offers an operation `diff` that can be used for computing the difference. The computation is based on a predecessor/successor relationship that the database maintains for objects that occur in multiple versions.

The implementation of the implicit `parse` method is not based on incremental parsing techniques. The parser, in a crude way, deletes an existing abstract syntax tree and all reference edges to nodes not contained in the tree and constructs a new tree. The reference edges are reestablished by reevaluating semantic rules. Without considering version management incremental parsing would only be an optimisation. The situation is different when we consider free textual input of documents that are under version control. Here the fine-grained predecessor/successor relationship between different nodes of a syntax-tree, managed by the database system, is lost when the tree is deleted and constructed anew after free textual input. In fact, the `diff` operation then returns wrong results. To retain this relationship as far as possible in order not to hamper version merging requires application of incremental parsing techniques [GM79]. Then parsing would delete only those nodes that should no longer occur and create only those nodes that were not included before.

The problem of configuration management has not been sufficiently addressed in this thesis. Semantic relationships with other document versions are established during editing as determined by the semantic rules. They are, however, only created with those other versions of documents that have either been selected explicitly or are the default version. In that way a user accesses exactly one configuration at a time. What is not yet supported is the explicit construction of a configuration. To facilitate this, tools would have to compute the set of document versions that are consistent with each other. This obviously interferes with evaluation of semantic rules and it is not clear to us when the required evaluations can best be done. This will be studied in detail in a future PhD project.

Bibliography

[AB91] S. Abiteboul and A. Bonner. Objects and Views. *ACM SIGMOD Record*, 20(2):238–247, 1991. Proc. of the 1991 ACM SIGMOD Conf. on Management of Data, Denver, Co.

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[ABD+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proc. of the* 1^{st} *Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan*, pages 223–240. North-Holland, 1990.

[ABJ89] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of transitive Relationships in Large Data and Knowledge Bases. *ACM SIGMOD Record*, 18(2):253–262, 1989. Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, OR.

[ABM+90] T. L. Anderson, A. J. Berre, M. Mallison, H. H. Porter, and B. Schneider. The HyperModel Benchmark. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the International Conference on Extending Database Technology*, volume 416 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 1990.

[AHS91] T. Andrews, C. Harris, and K. Sinkel. Ontos: A Persistent Database for C++. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, pages 387–406. Prentice-Hall, 1991.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[AJPO88] Ada Joint Program Office. Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Revision A. Technical Report DoD-STD-1838A, U.S. Department of Defense, 1988.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilers – Principles, Techniques and Tools.* Addison Wesley, 1986.

[Bay95] B. Bayard. Konzeption einer objektorientierten Erweiterung der Designsprache und des Werkzeugs OPUS. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1995.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1992.

[Bec95] W. Beckmann. Datenintegration in generierten, syntax-gesteuerten Software-Entwicklungsumgebungen. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1995.

[Ber87] P. A. Bernstein. Database System Support for Software Engineering. In *Proc. of the 9th Int. Conf. on Software Engineering, Monterey, Cal.*, pages 166–178, 1987.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology — EDBT'92, 3rd Int. Conf on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 1992.

[BFG93a] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, 1993.

[BFG93b] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proc. of the 2nd Int. Conf. on the Software Process, Berlin, Germany*, pages 75–83. IEEE Computer Society Press, 1993.

[BFGG92] S. Bandinelli, A. Fuggetta, C. Ghezzi, and S. Grigolli. Process Enactment in SPADE. In J. C. Derniame, editor, *Proc. of the 2nd European Workshop on Software Process Technology, EWSPT '92, Trondheim, Norway*, volume 635 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 1992.

[BG81] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–222, 1981.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BK90] N. S. Barghouti and G. E. Kaiser. Modeling Concurrency in Rule-Based Development Environments. *IEEE Expert*, pages 15–27, December 1990.

[BKK85] F. Bancilhon, W. Kim, and H. F. Korth. A model of CAD transactions. In A. Pirotte and Y. Vassilou, editors, *Proc. of the 11th Int. Conf. on Very Large Databases, Stockholm, Sweden*, pages 25–33. Morgan Kaufmann, 1985.

[BL85] T. Brandes and C. Lewerentz. GRAS: A non-standard data base system within a software development environment. In *Proc. of the GTE Workshop on Software Engineering Environments for Programming in the Large, Harwichport*, pages 113–121, 1985.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, pages 61–72, May 1988.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[BOSV89] J. Boarder, H. Obbink, M. Schmidt, and A. Völker. Advanced techniques and methods of system production in a heterogeneous, extensible, and rigorous environment. In N. Madhavji, W. Schäfer, and H. Weber, editors, *Proc. of the 1st Int. Conf. on System Development Environments and Factories, Berlin, Germany*, pages 199–206, London, 1989. Pitman Publishing.

[BPR88] M. R. Blaha, W. J. Premerlani, and J. E. Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414–427, 1988.

[Bru94] J. Brunsmann. Versions- und Konfigurations-Verwaltung in syntax-gesteuerten Software-Entwicklungswerkzeugen. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1994.

[BS82] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, 6(4):557–575, 1982.

[Bud92] F. Buddrus. Generierung von syntaxgesteuerten Werkzeugen auf der Basis eines objektorientierten Datenbanksystems. Master's thesis, University of Dortmund, Dept. of Computer Science, June 1992.

[Cag90] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[Car85] L. Cardelli. Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1985.

[Cat93] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufman, 1993.

[CCS94] C. Collet, T. Coupaye, and T. Svensen. NAOS Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proc. of the* 20th *Int. Conf. on Very Large Databases, Santiago, Chile*, 1994.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. *ACM SIGMOD Record*, 22(3):12–21, 1993. Proc. of the 1993 ACM SIGMOD Conf., Washington, D.C.

[CFGGR91] J. Cramer, W. Fey, M. Goedicke, and M. Große-Rhode. Towards a Formally Based Component Description Language as a Foundation for Reuse. *Structured Programming*, 12(2):91–110, 1991.

[CH74] R. H. Campbell and A. N. Haberman. The Specification of Process Synchronization by Path Expressions. In *Operating Systems – Proc. of an Int. Symposium, Rocquencourt, France*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer, 1974.

[Che76] P. P. Chen. The Entity-Relationship Model – Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CLZ94] A. Coen-Porisini, L. Lavazza, and R. Zicari. Assuring Type-Safety of Object Oriented Languages. *Journal of object-oriented Programming*, 6(9):25–30, February 1994.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. *ACM SIGMOD Record*, 14(2):316–325, 1984. Proc. of the ACM SIGMOD 1984 Int. Conf. on the Management of Data, Boston, MA.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CPW87] S. Ceri, B. Pernici, and G. Wiederhold. Distributed Database Design Methodologies. *Proc. of the IEEE*, 75(5):533–546, 1987.

[CS92] R. G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, 1992.

[CW76] H. J. Curnow and B. A. Wichman. A synthetic benchmark. *Computer Journal*, 19(1):43–49, 1976.

[Dar92] S. A. Dart. The Past, Present, and Future of Configuration Management. In J.v Leeuwen, editor, *Proc. of the International Federation for Information Processing (IFIP) World Congress, Vol.1, Spain*, pages 244–248, 1992.

[Dat86] C. J. Date. *Introduction to Database Systems, Vol. 1.* Addison Wesley, 1986.

[Dat89] C. J. Date. *A Guide to the SQL standard.* Addison Wesley, 1989.

[DEH$^+$91] S. Dißmann, W. Emmerich, B. Holtkamp, K. Lichtinghagen, and L. Schöpe. OMSs Comparative Study. Deliverable ESPRIT Project ATMOSPHERE D2.4.3-rep-1.0-UDO-EL, Commission of the European Communities, DG XIII, 1991.

[DEL92] S. Dewal, W. Emmerich, and K. Lichtinghagen. A Decision Support Method for the Selection of OMSs. In *Proc. of the 2nd Int. Conf. on Systems Integration, Morristown, N.J.*, pages 32–40. IEEE Computer Society Press, 1992.

[Des88] T. Despeyroux. TYPOL – A Framework to Implement Natural Semantics. Technical Report 94, INRIA, Roquencourt, 1988.

[DeW91] D. DeWitt. The Wisconsin Benchmark: Past, Present, & Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction processing Systems*, chapter 3, pages 119–166. Morgan Kaufmann, 1991.

[DG90] W. Deiters and V. Gruhn. Managing Software Processes in MELMAC. *ACM SIGSOFT Software Engineering Notes*, 15(6):193–205, 1990. Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming Environments based on Structure Editors: The Mentor Experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, 1984.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and M. Mélèse. Document structure and modularity in Mentor. *ACM SIGSOFT Software Engineering Notes*, 9(3):141–148, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[DGL86] K. R. Dittrich, W. Gotthard, and P. C. Lockemann. DAMOKLES – A database system for software engineering environments. In R. Conradi, T. M. Didriksen, and D. H. Wanvik, editors, *Proc. of an Int. Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer, 1986.

[DHS+91] S. Dewal, H. Hormann, L. Schöpe, U. Kelter, D. Platz, and M. Roschewski. Bewertung von Objektmanagementsystemen für Software-Entwicklungsumgebungen (in German). In *Proc. of the GI Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 404–411. Springer, 1991.

[Dit86] K. R. Dittrich. Object-oriented Database Systems: the Notion and the Issues. In K. Dittrich and U. Dayal, editors, *Proc. of the 1986 Int. Workshop on Object-Oriented Database Systems*, pages 2–4. IEEE Computer Society Press, 1986.

[dM78] T. de Marco. *Structured Analysis and System Specification.* Yourdan, 1978.

[DM93] C. Delobel and J. Madec. Version Management in O_2. Technical report, O_2-Technology, 1993.

[dSR95] A. del Soldato and S. Rampichini. From OMT to Beta: An Integrated Environment. Master's thesis, University of Pisa, Dept. of Mathematics, 1995.

[ECM89] ECMA. Introducing PCTE+. Technical Report ECMA/TC33/89/48, Independent European Programme Group – Technical Area 13, 1989.

[EK92] W. Emmerich and M. Kampmann. The Merlin OMS Benchmark – Definition, Implementations and Results. Technical Report 65, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1992.

[EKS93] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the* 4^{th} *Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[ELS87] G. Engels, C. Lewerentz, and W. Schäfer. Graph-grammar engineering: A Software Specification Method. In *Proc. of the* 3^{rd} *Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 186–201. Springer, Berlin, 1987.

[ENS87] G. Engels, M. Nagl, and W. Schäfer. On the Structure of Structure oriented Editors for Different Applications. *ACM SIGPLAN Notices*, 22(1):190–198, 1987. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal.

[ES89] G. Engels and W. Schäfer. *Programmentwicklungsumgebungen - Konzepte und Realisierung.* Teubner, 1989.

[ES94] W. Emmerich and W. Schäfer. Groupie — An Environment supporting Group-Oriented Architecture Development. Technical Report 71, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1994. Submitted for Publication.

[ESW92] W. Emmerich, W. Schäfer, and J. Welsh. Suitable Databases for Process-centred Environments Do not yet Exist. In J. C. Derniame, editor, *Proc. of the* 2^{nd} *European Workshop on Software Process Technology, EWSPT '92, Trondheim, Norway*, volume 635 of *Lecture Notes in Computer Science*, pages 94–98. Springer, 1992.

[ESW93] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the* 4^{th} *European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[Feh93] R. Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 1993.

[Fei91] Peter H. Feiler. *Configuration Management Models in Commercial Environments.* Technical Report CMU/SEI-91-TR-7, ESD-9-TR-7, 1991.

[Fel79] S. I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 4(3):255–256, 1979.

[FMZ94a] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for an Object Database System. In *Proc. of the* 6^{th} *International Workshop on Persistent Object Systems, Tarascon, France*, Workshops in Computing, pages 284–301. Springer, 1994.

[FMZ94b] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the* 20^{th} *Int. Conference on Very Large Databases, Santiago, Chile*, pages 261–272, 1994.

[GE90] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. Compiler Generation Report 20, GMD Research Center at University of Karlsruhe, 1990.

[Ger94] S. Gerle. DoBench Communication Server — Ein Werkzeug für den Nachrichtenaustausch zwischen Entwicklungsumgebungen. Master's thesis, University of Dortmund, Dep. of Computer Science, Software-Technology, 1994.

[Ges95] B. Gesell. Generierung von grafischen syntax-gesteuerten Software-Entwicklungswerkzeugen. Master's thesis, University of Dortmund, Dept. of Computer science, Software Technology, 1995.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In O. Nierstrasz, editor, *ECOOP '93 — Proc. of the* 7^{th} *European Conf. on Object-Oriented Programming, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–131, 1992.

[GL81] H. J. Genrich and K. Lautenbach. System Modelling with High-Level Petri Nets. *Theoretical Computer Science*, 13(1):109–136, 1981.

[GL85] W. Gotthard and P. C. Lockemann. Datenbanksysteme für Software-Produktionsumgebungen – Anforderungen und Konzepte (in German). In W. E. Proebster, R. Remshardt, and H. A. Schmid, editors, *Methoden und Werkzeuge zur Entwicklung von Programmsystemen – Fachberichte und Referate Band 16*, pages 185–210. Oldenbourg, 1985.

[GM79] C. Ghezzi and D. Mandrioli. Incremental Parsing. *ACM Transactions on Programming Languages and Systems*, 1:58–70, 1979.

[GM84] D. B. Garlan and P. L. Miller. GNOME: An introductory Programming Environment based on a Family of Structure Editors. *ACM SIGSOFT Software Engineering Notes*, 9(3):65–72, 1984.

[GMT87] F. Gallo, R. Minot, and I. Thomas. The Object Management System of PCTE as a Software Engineering Database Management System. *ACM SIGPLAN NOTICES*, 22(1):12–15, 1987.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison Wesley, 1984.

[Gol85] A. Goldberg. *Smalltalk 80: The Language and its Implementation.* Addison Wesley, 1985.

[GOO94] GOODSTEP Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In K. Ohmaki, editor, *Proc. of the Asia-Pacific Software Engineering Conference, Tokyo, Japan*, pages 410–420. IEEE Computer Society Press, 1994.

[Gor87] K. E. Gorlen. An Object/Oriented Class Library for C++ Programs. *Software – Practice and Experience*, 17(12):181–207, 1987.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and Update Semantics of Consistent Views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

[Gra78] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating systems – An advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F., pages 393–481. Springer, 1978.

[GRDM90] J. Giavotto, G. Rosuel, A. Devarenne, and A. Mauboussin. Design Decisions for the Incremental Adage Framework. In *Proc. of the* 12th *Int. Conf. on Software Engineering, Cannes, France*, pages 86–95, 1990.

[HK88] S. E. Hudson and R. King. The Cactis Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, 1988.

[HN86] A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Hoo87] R. Hoover. *Incremental graph evaluation*. PhD thesis, Cornell University, Dept. of Computer Science, Ithaca, NY, 1987. Technical Report No. 87-836.

[HP81] A. N. Habermann and D. E. Perry. System Composition and Version Control for Ada. In H. Hünke, editor, *Proc. of the Symposium on Software Engineering Environments, Lahnstein, FRG*, pages 331–344. North-Holland, 1981.

[Hru87] P. Hruschka. ProMod – in the age 5. In H. K. Nichols and D. Simpson, editors, *Proc. of the 1st European Software Engineering Conference, Strasbourg, France*, volume 289 of *Lecture Notes in Computer Science*, pages 288–296. Springer, 1987.

[Hud87] S. E. Hudson. Incremental Attribute Evaluation: An Algorithm for Lazy Evaluation in Graphs. Technical Report 87-20, University of Arizona, 1987.

[HVZ90] G. Harrus, F. Velez, and R. Zicari. Implementing schema updates in an object-oriented database system: a cost analysis. Technical report, GIP Altair, 1990.

[HW91] B. Holtkamp and H. Weber. Object-Management Machines: Concept and Implementation. *Journal of Systems Integration*, 1:367–389, 1991.

[HZ90] S. Heiler and S. B. Zdonik. Object Views: Extending the Vision. In *Proc. of the 6th Int. Conf. on Data Engineering, Los Angeles, CA*, pages 86–93. IEEE Computer Society Press, 1990.

[ISO86] ISO 8879. Information processing – Text and Office Systems – Standardised General Markup Language SGML. Technical report, International Standards Organisation, 1986.

[Jah94] J.-H. Jahnke. Objekt-orientierte Spezifikation und Validierung von Kontextbedingungen in der Werkzeugspezifikations-Sprache GTSL. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1994.

[JF82] G. F. Johnson and C. N. Fisher. Non-syntactic attribute flow in language based editors. In *Proc. of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 185–195. ACM Press, 1982.

[JPAR68] W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross. Automatic Generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, 1968.

[JPSW94] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development through a Knowlege-based Environment. In A. C. W. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Advances in Software Process Technology*, pages 103–129. Wiley, 1994.

[Jun95a] G. Junkermann. A Dedicated Process Design Language based on EER-Models, Statecharts and Tables. In *Proc. of the 7th Int. Conf. on Software Engineering and Knowledge Engineering, Rockville, Maryland*, pages 487–496. Knowledge Systems Institute, 1995.

[Jun95b] G. Junkermann. *ESCAPE — Eine graphische Sprache zur Spezifikation von Software-Prozeßmodellen.* PhD thesis, University of Dortmund, Dept. of Computer Science, 1995. Forthcoming.

[Kas80] U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[Kas94] U. Kastens. Construction of application generators using Eli. Technical Report tr-ri-94-143, University of Paderborn, Germany, 1994.

[Kat84] R. H. Katz. Transaction Management in the Design Environment. In E. Gardarin and E. Gelenbe, editors, *New Applications of Data Bases*, pages 259–273. Academic Press, 1984.

[KBC$^+$89] W. Kim, N. Ballou, H.-T. Chou, J. F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. Addison Wesley, 1989.

[Kel89] U. Kelter. Implementation of Group Transactions in Structurally Object-Oriented Database Systems. Technical Report 37, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1989.

[KFP88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, pages 40–49, May 1988.

[KHPW90] G. E. Kaiser, W. Hseush, S. S. Popovich, and S. F. Wu. Multiple Concurrency Control Policies in an Object-Oriented Programming System. In *Proc. of the* 2^{nd} *IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas*, pages 623–626, 1990.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer, 1982.

[KLM83] G. Kahn, B. Lang, and B. Melese. Metal: a Formalism to Specify Formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KR78] B. W. Kerningham and D. M. Ritchie. *Programming in C*. Prentice Hall, 1978.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. *ACM SIGMOD Record*, 14(4):388–401, 1985. Proc. of the ACM SIGMOD 1985 Int. Conf. on the Management of Data, Austin, Texas.

[KSW92] N. Kiesel, A. Schürr, and B. Westfechtel. Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications. Technical Report AIB 92-44, Aachen Technical University, Dept. of Computer Science, 1992.

[Lew88] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 13(5):173–182, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[Lin84] M. A. Linton. Implementing Relational Views of Programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):51–63, 1991.

[LMN93] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented Programming in the BETA Programming Language.* Addison Wesley, 1993.

[LR89] C. Lécluse and P. Richard. The O_2 Database Programming Language. In *Proc. of the* 15^{th} *Int. Conf. on Very Large Data Bases, Amsterdam, The Netherlands*, pages 411–422. Morgan Kaufmann, 1989.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. *ACM SIGMOD Record*, 17(3):424–433, 1988. Proc. of the 1989 ACM SIGMOD Int. Conf. on the Management of Data, Portland, OR.

[LS88] C. Lewerentz and A. Schürr. GRAS, a management system for graph-like documents. In *Proc. of the* 3^{rd} *Int. Conf. on Data and Knowledge Bases*, pages 19–31. Morgan Kaufmann, 1988.

[Mai89] D. Maier. Making Database Systems Fast Enough for CAD Applications. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 573–582. Addison Wesley, 1989.

[Mey88] B. Meyer. *Object-oriented software construction.* Prentice Hall, 1988.

[Mey89] B. Meyer. *EIFFEL – The Libraries.* Interact. Software Engineering, 1989.

[Mey92] B. Meyer. *EIFFEL – The Language.* Prentice Hall, 1992.

[MF81] R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Transactions on Software Engineering*, 7(5):472–482, 1981.

[Mos85] J. B. Moss. *Nested Transactions An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[MS91] N. H. Madhavji and W. Schäfer. Prism – Methodology and Process-Oriented Environment. *IEEE Transactions on Software Engineering*, 17(12):1270–1283, 1991.

[Nag85] M. Nagl. An Incremental and Integrated Software Development Environment. *Computer Physics Communications*, 38:245–276, 1985.

[NW94] J. Neuhaus and X. Wu. Implementing a General OMS Benchmark on PCTE and ECMA PCTE. In T. Lindquist and H. Koehnemann, editors, *Proc. of the PCTE'94 Conference, San Francisco, CA*, pages 5–20. PCTE Interface Management Board Association, 1994.

[Obe82] R. Obermarck. Deadlock Detection for All Resource Classes. *ACM Transactions on Database Systems*, 7(2):187–208, 1982.

[Pea90] P. K. Pearson. Fast Hashing of Variable-Length Text Strings. *Communications of the ACM*, 33(6):677–680, 1990.

[Per89] D. E. Perry. The Inscape Environment. In *Proc. of the* 11*th* *Int. Conf. on Software Engineering, Pittsburgh, PA*, pages 2–12. IEEE Computer Society Press, 1989.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. Aarhus Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[PPT88] M. H. Penedo, E. Ploedereder, and I. Thomas. Object Management Issues for Software Engineering Environments – Workshop Report. *ACM SIGSOFT Software Engineering Notes*, 13(5):226–234, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[PS92] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proc. of the* 14*th* *Int. Conf. on Software Engineering, Melbourne, Australia*, pages 262–279. IEEE Computer Society Press, 1992.

[PSW92] B. Peuschel, W. Schäfer, and S. Wolf. A Knowledge-based Software Development Environment Supporting Cooperative Work. *International Journal for Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Rei84] S. P. Reiss. PECAN: Program Development that Supports Multiple Views. *IEEE Transactions on Software Engineering*, 11(3):276–285, 1984.

[Rei90] S. P. Reiss. Interacting with the FIELD environment. *Software – Practice and Experience*, 20(S1):S1/89–S1/115, 1990.

[Rod95] R. Rodriguez. A SLANG process model to support development of C++ libraries at British Airways. Master's thesis, Politechnico di Milano, Dept. of Eletrotecnica, 1995.

[Roy70] W. W. Royce. Managing the Development of Large Software Systems. In *Proc. WESCON*, 1970.

[RT81] T. W. Reps and T. Teitelbaum. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):449–477, 1981.

[RT84] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[RT88] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator – a system for constructing language based editors.* Springer, 1988.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual, third Edition.* Springer, 1989.

[SAD94] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In J. Bubenko M. Jarke and K. Jefferey, editors, *Proc. of the* 4*th* *Int. Conf. on Extending Database Technology, Cambridge, UK*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 1994.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, 1973.

[Sch86] W. Schäfer. *Eine integrierte Softwareentwicklungsumgebung: Konzepte, Entwurf und Implementierung*. PhD thesis, University of Osnabrück, 1986.

[Sch91a] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD thesis, RWTH Aachen, 1991.

[Sch91b] A. Schürr. PROGRES, A VHL-Language Based on Graph Grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of the* 4^{th} *Int. Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1991.

[Sch94] D. Schnorpfeil. Effiziente Verwaltung von Syntaxgraphen in Relationalen Datenbanksystemen. Master's thesis, University of Dortmund, Dept. of Computer Science, March 1994.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases – Proc. of the* 2^{nd} *Int. Conf., DOOD '91, Munich, Germany*, volume 566 of *Lecture Notes in Computer Science*, pages 189–207. Springer, 1991.

[SP82] H.-J. Schek and P. Pistor. Data Structures for an Integrated Database Management and Information Retrieval System. In *Proc. of the* 12^{th} *Int. Conf. on Very Large Databases, Mexico City, Mexico*, pages 197–207. Morgan Kaufmann, 1982.

[Spe92] Special Working Group on Ada Programming Support Environments. International Requirements and Design Criteria for the Portable Common Interface Set. PCIS Technical Report, PCIS, July 1992.

[Spi88] J. M. Spivey. *Understanding Z*. Cambridge University Press, 1988.

[SS95] S. Sachweh and W. Schäfer. Version Management for tightly integrated Software Engineering Environments. In M. S. Verrall, editor, *Proc. of the* 7^t*h International Conference on Software Engineering Environments, Nordwijkerhout, The Netherlands*, pages 21–31. IEEE Computer Society Press, 1995.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[Sun93] SunSoft. *ToolTalk 1.1.1 Reference Manual*. SunSoft, 2550 Garcia Avenue, Mountain View, CA 94043, USA, Solaris 2.3 edition, 1993.

[SW87] H.-J. Schek and G. Weikum. DASDBS – Concepts and architecture of a novel database system (in German). *Informatik Forschung und Entwicklung*, 2(3):105–121, 1987.

[SW89] W. Schäfer and H. Weber. European Software Factory Plan – The ESF-Profile. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering – Foundations and current perspectives*, chapter 22, pages 613–637. Van Nostrand Reinhold, New York, 1989.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[SY82] G. M. Sacco and S. B. Yao. Query Optimization in Distributed Database Systems. In M. C. Yovits, editor, *Advances in Computers*, volume 21, pages 225–273. Academic Press, 1982.

[Tho89] I. Thomas. Tool Integration in the PACT Environment. In *Proc. of the* 11th *Int. Conf. on Software Engineering, Pittsburg, Penn.*, pages 13–22. IEEE Computer Society Press, 1989.

[Tho93] I. Thomas. Observations on Object Management Systems. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the* 4th *European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, volume 717 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1993.

[Tic85] W. F. Tichy. RCS – A System for Version Control. *Software – Practice and Experience*, 15(7):637–654, 1985.

[Wag95] M. Wagener. Prozeßgesteuerte Software-Entwicklungswerkzeuge. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1995.

[Was89] A. I. Wassermann. Tool Integration in Software Engineering Environments. In F. Long, editor, *Proceedings of the International Workshop on Environments*, pages 137–149, Chinon, France, September 1989. Springer-Verlag. Proceedings published as Lecture Notes in Computer Science, Vol. 467.

[WBK91] J. Welsh, B. Broom, and D. Kiong. A Design Rational for a Language-based Editor. *Software – Practice and Experience*, 21(9):923–948, 1991.

[Wei84] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct. 1984.

[Wes89a] B. Westfechtel. Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control. Technical Report AIB 89-8, Aachen Technical University, Dept. of Computer Science, 1989.

[Wes89b] B. Westfechtel. Revision Control in an Integrated Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 17(7):96–105, 1989.

[Wes91] B. Westfechtel. *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung.* Springer, 1991. Informatik Fachberichte 280.

[WG84] W. B. Waite and G. Goos. *Compiler Construction.* Springer, 1984.

[Wol94] S. Wolf. *Unterstützung kooperativer Softwareentwicklung – Ein transaktionsbasierter Ansatz.* PhD thesis, University of Dortmund, 1994.

[WP87] A. I. Wassermann and P. A. Pircher. A Graphical, Extensible Integrated Environment for Software Development. *ACM SIGPLAN Notices*, 22(1):131–142, 1987. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal.

[WWFT88] A. L. Wolf, J. C. Wileden, C. D. Fisher, and P. L. Tarr. P Graphite: An Experiment in Persistent Typed Object Management. *ACM SIGSOFT Software Engineering Notes*, 13(5):130–142, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

Index

Appendix A

Language Definition of GTSL

The formal definition of GTSL serves a number of purposes. In the first place, its aim is to achieve an unambiguous language definition. To have a formal, thus unambiguous definition of a tool specification language enables multiple tool builders to cooperate since they can discuss specifications on the basis of well-defined language concepts. Moreover, the formal definition serves as a specification for the correctness of our compilers. For that reason, we restrict ourselves to define the tool configuration, class interface and class specification languages formally here, since these are the languages that must be handled by our compiler. In an informal way, the required preciseness will hardly ever be achieved. Finally, we will have to thoroughly think about the various language concepts during their formal definition. This will cleanse the language.

A number of different concerns have to be addressed by the formal definition of a language. The context-free syntax of a language needs to be defined as a starting point. The definition of static semantics restricts the set of syntactically correct sentences to those that are meaningful. In our case, we will define the static semantics of GTSL in a way that a high number of meaningless tool specifications are excluded. This is one of the main advantages of defining an application oriented language rather than using a general purpose object-oriented language such as Eiffel or O_2C. Finally, dynamic semantics gives a sentence of the language a meaning. In our case dynamic semantics will define the impact of a concept on tool execution.

The context free syntax of a language is best defined by a context-free grammar [Sal73]. As a language for defining the context-free grammar we use the grammar formalism offered by Eli and are then able to generate a parser directly from the definition. Furthermore, we introduce some generic productions in order to simplify the presentation of the grammar. *Optional* non-terminal symbols are denoted with the prefix `opt_`. For each optional production, another production `opt_<x>::= | <x>` will be defined that decides whether or not to expand the respective symbol. These productions are omitted in the presentation and are considered being implicitly defined. Moreover, we consider *list* non-terminal symbols. They are all labelled with the suffix `_list`. For each of these symbols, a production `<x>_list::=<x>| <x> <x>_list` is considered as implicitly defined.

There have been various suggestions on how to define static semantics of a language. Attribute grammars [Knu68, Kas80], rules [Des88] and graph grammars [ENS87] have been defined in a way that a static semantic analyser can automatically be generated from the specification. Although this looks promising for a language definition at first glance, it turns out to be inappropriate for the purpose of presenting the static semantics definition in an understandable way. The level of abstraction in these formalisms must be considered too low. In attribute grammars, for instance a number of auxiliary attributes have to be defined that are used in long-distance attribute propagation and confuse the reader rather than clarifying the language definition. In rule-based formalisms, concepts like mappings or sets that would contribute to the comprehensible language definition have to be implemented in terms of rules and axioms. The same holds for graph grammars. Here sets have to be represented by sets of nodes and edges and mappings have to be implemented in terms of node-set operators. These formalisms rather have to be considered as high-level implementation languages. In fact, we have used ordered attribute grammars for the implementation of static semantic analysis in the GTSL compiler (c.f. Section 8.1). For the definition here, we use first-order logic, since it is purely declarative. As a language for first-order logic we use the standard mathematical notation. We could have equally well used Z [Spi88] for this purpose, but did not because more readers will be familiar with the standard notation.

There are several approaches to specify the dynamic semantics of a language. In a *denotational semantic specification* a number of functions between syntactic domains are defined. Then functionals are defined that map functions to more specific functions. The semantics of the language are then the fixpoints of these functionals, i.e. those functions that do not get more specific when being applied to the functional. In an *axiomatic semantic specification*, a number of axioms and inference rules are defined. An axiom is true by definition. Then inference rules can be applied to prove a goal from axioms and already proven goals. The semantics of the language is then the set of provable goals. Finally, an *operational semantic specification* is given by the formal definition of an abstract machine that interprets sentences of the language. The problem with denotational and axiomatic semantic definitions is that they cannot appropriately handle persistence. In particular, the transaction semantics of interactions can hardly be defined in a denotational or axiomatic way. The remaining option is thus an operational semantic specification. Therefore, an abstract GTSL machine must be defined that interprets GTSL specifications. The definition of this machine can best be based on the abstract O_2 machine that is part of the semantic definition of the O_2 data model [BDK92]. This has the additional advantage that GTSL code generation can be directly deduced from the operational specification. The code generation is correct then, if the following diagram commutes:

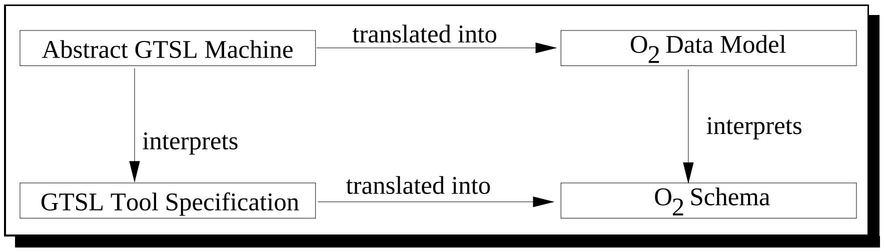


Figure A.1: Specification of Dynamic Semantics of GTSL

The definition of this abstract GTSL machine, however, will be far too time-consuming to be performed during the course of this thesis. A technique that is often used to reduce the complexity of a semantics definition is bootstrapping. The idea is to formally define a small language kernel. Then other language concepts can be expressed in terms of this kernel. Unfortunately, this technique cannot be applied to our problem, since the language kernel itself had to include most of the language concepts. This is because GTSL's multiple paradigms can hardly be expressed by each other. Therefore, we define the dynamic semantics informally.

As mentioned above, GTSL is used together with a number of pre-defined classes. Most of these classes will be implemented in GTSL itself. The semantics of these pre-defined classes is, therefore, defined by bootstrapping in GTSL.

A.1 Configurations

Context-free Syntax We want to allow for incremental compilation of GTSL class definitions. Therefore, the set of classes which belong to the specification of a tool must be known. Otherwise, we could not check in the import part of a class interface whether the imported class really exists. The tool configuration, therefore, declares all classes that belong to the specification of the tool. The declaration of these classes is split into two parts. A mandatory section (production `conf_inc_part`) defines the increment classes and an optional section (production `opt_conf_nsc_part`) determines the specification's non-syntactic classes. The next section determines the root increment type for the tool. Since the root increment is always a complete document, we use the term *document* synonymously.

A PSDE contains a number of tools. Typically each tool operates on a different type of document whose structure is determined in a tool specification. Semantic relationships between increments are not confined to increments of the same document type. Due to inter-document consistency constraints, they frequently exist between documents of different types. For the declaration of these semantic relationships, a class has to use classes from other tool specifications. The dependency between different tool specifications, induced by this use relationship, should not be hidden in the specification part of increment classes, but should be explicitly defined. Therefore, the configuration definition contains an import list that declares those classes that are imported from other tool specifications (production `opt_conf_import_list`). A tool configuration also defines the set of exported classes (`opt_conf_export_part`). Classes not contained in this export are hidden such that other tool configurations cannot import them. Therefore, configuration definitions play the same role as systems [HP81] or subsystems [ES94] in architecture definitions, since they display dependencies between components on a higher-level of abstraction than classes or modules.

The management of static semantic error messages or messages denoting inter-document consistency constraint violations is merely pre-defined. In particular, there are a number of pre-defined error descriptors each of which represents an error message. A tool builder may want to add tool-specific error descriptors and use these in semantic rules (as in the examples on Pages 152–154). These error messages cannot be defined in an increment class. Moreover, they cannot be defined in a tool-specific non-syntactic class since they rather extend a set of pre-defined error-descriptors. In order to avoid introducing yet another document for error messages, we include a section in the configuration definition where the domain of the pre-defined atomic type `ERROR` can be extended. The tool configuration is an appropriate location, since the type `ERROR` is visible in any GTSL class. The context-free syntax of configurations then looks as follows:

```
configuration :  'CONFIGURATION' IDENTIFIER ';'
                   opt_conf_import_list
                   conf_inc_part
                   opt_conf_nsc_part
                   conf_root_part
                   opt_conf_export_part
                   opt_conf_errors
                 'END' 'CONFIGURATION' IDENTIFIER '.' .
```

Static Semantics For the formal definition of import-export relationships between configurations, we will have to consider the set of all configurations that define tools of a PSDE. Let, therefore, SDE denote the set of all configurations of PSDE tools. Moreover, let A be the set of all characters included in the ASCII character set and A^* be the free monoid over A. We define a mapping $name : SDE \rightarrow A^*$ with $t \mapsto$ string that matches the first `IDENTIFIER` symbol in production configuration. For a configuration $t \in SDE$, $name(t)$ shall be the name of the tool. We require uniqueness of tool names in the PSDE, so as to use names for tool identification purposes:

$$\bigwedge_{t_1,t_2 \in SDE} name(t_1) = name(t_2) \Rightarrow t_1 = t_2 \tag{A.1}$$

A.1.1 Classes of a Configuration

Context-free Syntax The declaration of a class in a configuration also encompasses the definition of its super classes. The inheritance relationship cannot reasonably be defined in increment class interfaces for two reasons. Firstly, the inheritance relationship must be acyclic. This can be hardly checked if its definition is spread over a number of interface definitions that are incrementally compiled[1]. Secondly, the inheritance relationship has to be known completely in order to check some of the static semantic conditions of class interfaces. Therefore, each class declaration in a configuration is accompanied by the declaration of its super classes. If the configuration document is compiled before compiling the class interface documents, the inheritance relationship will be known during interface compilation.

The syntax definition of configurations forces the tool builder to specify for each class at least one super class. This assures that all increment classes (transitively) inherit from the pre-defined increment class `Increment`, the common super-class for all pre-defined classes. We can then define the common properties of any increment in this class.

```
conf_import :       'IMPORT' 'FROM' 'CONFIGURATION' IDENTIFIER ':'
                       'INCREMENT' 'CLASSES' ident_list ';'
                       opt_conf_imp_nsc_part
                    'END' 'IMPORT' ';'.
ident_list :        IDENTIFIER / IDENTIFIER ',' ident_list .
conf_imp_nsc_part : 'NON' 'SYNTACTIC' 'CLASSES' ident_list ';'.

conf_inc_part :     'INCREMENT' 'CLASSES'
                       conf_increment_id_list
                    'END' 'INCREMENT' 'CLASSES' ';' .
conf_increment_id : IDENTIFIER 'INHERIT' ident_list ';' .

conf_nsc_part :     'NON' 'SYNTACTIC' 'CLASSES'
                      conf_nsc_id_list
                    'END' 'NON' 'SYNTACTIC' 'CLASSES' ';' .
conf_nsc_id :       IDENTIFIER 'INHERIT' ident_list ';' .

conf_root_part :    'ROOT' 'INCREMENT' 'IS' IDENTIFIER .
```

Static Semantics In the sequel, let $\mathcal{C}_t := INC_t \uplus NSC_t$ denote the set of classes defined in a tool specification t. INC_t denotes the set of increment classes and NSC_t denotes the

[1]The Eiffel compilers that are available to date suffer from this problem. Before compiling a single class, they perform a pass on all classes that belong to a system only for the purpose of computing the inheritance relationship. A particularly bad performance of the compilers is an immediate consequence.

set of non-syntactic classes. If the context is clear, we will omit the index t. Moreover, let IMP_t denote the set of imported classes and LIB the set of classes that are pre-defined (c.f. Appendix B). Then $name : \mathcal{C} \uplus IMP \uplus LIB \to A^*$ denotes the name of the class as given by the string that matches symbol `IDENTIFIER` in the respective productions.

In order to enforce that mapping *name* becomes injective so as to use it for class identification purposes, we require that class names are unique within the scope of a configuration definition:

$$\bigwedge_{c_1, c_2 \in \mathcal{C} \uplus IMP \uplus LIB} name(c_1) = name(c_2) \Rightarrow c_1 = c_2 \tag{A.2}$$

The identifier lists given after the `INHERIT` keyword for each class determine the inheritance relationship. Let, therefore, $inherit : \mathcal{C} \to \mathcal{P}(A^*)$ denote the set of strings that match with the `IDENTIFIER` symbols in the `ident_list` of the declaration of a class. We then require that each of these strings denotes the name of a pre-defined class or a class defined in the tool configuration:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{id \in inherit(c)} \bigvee_{c' \in \mathcal{C} \uplus LIB} id = name(c') \tag{A.3}$$

Note, that according to this condition a class cannot inherit from an imported class. This is excluded for the following reason: The inheritance relationship imposes a much stronger dependency between classes than the import relationship. This is because most properties of a class are inherited to subclasses. Then a subclass can redefine and access them in any way. This means that information defined in a class is not hidden from its subclasses. All subclasses will immediately be affected whenever an inherited property is changed in a super-class. Opposed to that, a class can only import methods and properties from some other class. Client classes are only affected if the method signature or the property type is changed. Now, tool configurations play the role of subsystems in module architectures and are, therefore, the unit for task assignment, i.e. most likely different tool builders work on different configurations. In order to decrease the need for communication between the tool builders involved in a PSDE construction, we have to minimise the dependencies between different configurations. Therefore, we do not allow inheriting from a class that has been imported from another configuration.

To continue the formal definition, we have to define the inheritance relationship. The direct super classes of a class are defined by $pred : \mathcal{C} \cup LIB \to \mathcal{P}(\mathcal{C} \cup LIB)$, $c \mapsto \{c_i \in \mathcal{C} \cup LIB \mid name(c_i) \in inherit(c)\}$. Similarly, the direct subclasses of a class are declared as $succ : \mathcal{C} \cup LIB \to \mathcal{P}(\mathcal{C} \cup LIB)$, $c \mapsto \{c_i \in \mathcal{C} \cup LIB \mid name(c) \in inherit(c_i)\}$.

The transitive closure of mappings *pred* and *succ* are the formal basis for a number of static semantic definitions. They are used, e.g. to define formally that the inheritance hierarchy must be acyclic. They are recursively defined as follows: $pred^* : \mathcal{C} \cup LIB \to \mathcal{P}(\mathcal{C} \cup LIB)$, where

$$c \mapsto \begin{cases} c, & if\ pred(c) = \emptyset \\ \bigcup_{c_i \in pred(c)} pred^*(c_i), & otherwise \end{cases}$$

and $succ^* : \mathcal{C} \cup LIB \to \mathcal{P}(\mathcal{C} \cup LIB)$, where

$$c \mapsto \begin{cases} c & if\ succ(c) = \emptyset \\ \bigcup_{c_i \in succ(c)} succ^*(c_i) & otherwise \end{cases}$$

It is not reasonable to inherit properties such as regular expressions, interactions, unparsing schemes or abstract syntax children that are defined for increment classes to non-syntactic classes. Vice versa, it is unreasonable to declare increment classes as subclasses of non-syntactic classes. This is required by the following condition:

$$\begin{aligned} \bigwedge_{c \in INC} succ^*(c) \cap NSC = \emptyset \\ \bigwedge_{c \in NSC} succ^*(c) \cap INC = \emptyset \end{aligned} \tag{A.4}$$

We then require that a class does not transitively inherit from itself, i.e. that the inheritance hierarchy is acyclic:

$$\bigwedge_{c \in \mathcal{C}} (pred^*(c) \cap succ^*(c)) = c \tag{A.5}$$

Not requiring this property would prevent us, for instance, from reasonably defining the static semantics of polymorphism, since then a class' super class could at the same time be its subclass.

Finally, we have to require that the root class is an increment class. Otherwise, the syntax of the document would not be defined, since non-syntactic classes do not have an unparsing scheme. For the formal definition, let *id* be the string that matches with symbol `IDENTIFIER` in production `conf_root_part`. We require then:

$$\bigvee_{c \in INC} name(c) = id \tag{A.6}$$

A.1.2 Configuration Export

Context-free Syntax We want to be able to exclude particular classes from being imported by other configurations. This will allow us to apply the information hiding paradigm on an architectural level also to configurations. The export part section of a configuration is, therefore, used to enumerate those classes that are exported. Any class not included in that section is considered as being hidden.

```
conf_export_part :  'EXPORT CLASSES'
                      ident_list
                    'END' 'EXPORT' ';'.
```

Static Semantics Let $export : SDE \rightarrow \mathcal{P}(A^*)$ denote the set of strings that match with symbol `IDENTIFIER` in the `ident_list` of production `conf_export_part`. We then require that the export section denotes only classes that have been defined in the tool configuration:

$$\bigwedge_{t \in SDE} \bigwedge_{id \in export(t)} \bigvee_{c \in \mathcal{C}_t} name(c) = id \tag{A.7}$$

Let $CLASSES := \bigcup_{t \in SDE} \mathcal{C}_t$ be the set of all classes that belong to configurations of a PSDE. $exp : SDE \rightarrow \mathcal{P}(CLASSES), t \mapsto \{c \in \mathcal{C}_t \mid name(c) \in export(t)\}$ then computes the set of classes that are exported from a configuration. $\mathcal{C}_t^{\setminus exp(t)}$ are the hidden classes.

We can now complete the static semantics of the import interface of a configuration. For its formal definition, we define the set of imports IMP_t of a configuration t. Let $name : IMP_t \to A^*$ with $i \mapsto$ string that matches with symbol **`IDENTIFIER`** in production `conf_import`. We then require that the respective imported configuration exists:

$$\bigwedge_{t \in SDE} \bigwedge_{i \in IMP_t} \bigvee_{t' \in SDE} name(i) = name(t') \tag{A.8}$$

Let $impconf : IMP_t \to SDE$ with $i \mapsto t$ where $name(i) = name(t)$ compute the imported configuration.

Then let $imp\_classes : IMP_t \to \mathcal{P}(A^*)$ be the set of strings that match with symbol **`IDENTIFIER`** in `import_list` of productions `conf_import` and `conf_imp_nsc_part`. These strings must denote exported classes of the imported configuration:

$$\bigwedge_{t \in SDE} \bigwedge_{i \in IMP_t} \bigwedge_{id \in imp\_classes(i)} \bigvee_{c \in exp(impconf(i))} name(c) = id \tag{A.9}$$

A.1.3 Error Descriptors

Context-free Syntax The pre-defined type **`ERROR`** defines a number of error messages such as lexical errors and various kinds of syntax errors that are used in any tool specification. The error message section of a configuration definition can be used to further extend the doimain of **`ERROR`** with tool-specific error messages. Therefore, error descriptors are enumerated together with a string that represents the error message. The association between error descriptors and strings is available in a pre-defined operation `as_string` that returns the string for a given error descriptor. The context-free syntax is then as follows:

```
conf_errors :     'ADDITIONAL' 'ERRORS' conf_err_list 'END' 'ADDITIONAL' 'ERRORS' .
conf_err_list :   conf_err / conf_err ';' conf_err_list .
conf_err :        conf_const ':' STRING_CONST .
conf_const :      '#' IDENTIFIER .
```

Static Semantics Let $pd\_errors := \{OK, LE, IKW, CCP, TFE, ICC, CD\}$ be the set of pre-defined errors. Let moreover, ts_errors be the set of errors that have been specified in the error section of a tool configuration. We define a mapping $name : pd\_errors \cup ts\_errors \to A^*$ by

$$e \mapsto \begin{cases} \text{string that matches } \texttt{IDENTIFIER} \text{ in } \texttt{conf\_const} & if\ e \in ts\_errors \\ "OK", & if\ e \in pd\_errors \wedge e = OK \\ "LexicalError", & if\ e \in pd\_errors \wedge e = LE \\ "IncorrectKeyword", & if\ e \in pd\_errors \wedge e = IKW \\ "CharsAfterCorrectParse", & if\ e \in pd\_errors \wedge e = CCP \\ "TooFewElementsInList", & if\ e \in pd\_errors \wedge e = TFE \\ "InCompatibleConfiguration", & if\ e \in pd\_errors \wedge e = ICC \\ "CyclicDependency", & if\ e \in pd\_errors \wedge e = CD \end{cases}$$

We then require uniqueness of error descriptor names of a configuration $t \in SDE$ in order to use these names for identification purposes:

$$\bigwedge_{e_1, e_2 \in pd\_errors \cup ts\_errors} name(e_1) = name(e_2) \Rightarrow e_1 = e_2 \tag{A.10}$$

The domain of the atomic type `ERROR` is then defined as union of pre-defined and tool-specific error descriptors: $error := pd\_errors \cup ts\_errors$.

A.2 Class Interfaces

Context free syntax A GTSL increment class consists of an interface part and a specification part. We syntactically distinguish the different kinds of classes in the interface. This distinction enables detection of specification errors, such as definition of a regular expression for a non-terminal increment class or declaration of a deferred method in a class that is a leaf class in the inheritance hierarchy. The distinction could be inferred from the actual use of different concepts. It is rather introduced in the context free syntax in order to force the tool builder to explicitly think about the concepts he or she is using. The syntax is then defined as follows:

```
class
                : class_interface   class_specification .

class_interface
                : abstract_increment_interface
                / nonterminal_increment_interface
                / terminal_increment_interface
                / nonsyntactic_interface .

abstract_increment_interface
                : 'ABSTRACT' 'INCREMENT' 'INTERFACE' class_name ';'
                    inherit_section
                    opt_import_interface_section
                    abi_export_interface
                  'END' 'ABSTRACT' 'INCREMENT' 'INTERFACE' class_name '.' .
nonterminal_increment_interface
                : 'NONTERMINAL' 'INCREMENT' 'INTERFACE' class_name ';'
                    inherit_section
                    opt_import_interface_section
                    nti_export_interface
                  'END' 'NONTERMINAL' 'INCREMENT' 'INTERFACE' class_name '.' .
terminal_increment_interface
                : 'TERMINAL' 'INCREMENT' 'INTERFACE' class_name ';'
                    inherit_section
                    opt_import_interface_section
                    ti_export_interface
                  'END' 'TERMINAL' 'INCREMENT' 'INTERFACE' class_name '.' .
nonsyntactic_interface
                : 'CLASS' class_name ';'
                    inherit_section
                    opt_import_interface_section
                    nsc_export_interface
                  'END' 'CLASS' class_name '.' .

class_name      : IDENTIFIER .
```

Hence, a GTSL class consists of a class interface and a class specification. Interfaces are formally defined in this section, while specifications are subject of Subsection A.3.

Static Semantics Let INT be the set of class interfaces. Let $name : INT \to A^*$, with $i \mapsto$ string that matches with symbol `IDENTIFIER` in production `class_name`, computes the name of an interface. Moreover, let mapping $kind : INT \to \{ABI, NTI, TI, NSC\}$ define the kind of increment class according to the choice in production `class_interface`.

Obviously, each class defined in a tool configuration must have an interface definition. Vice versa, each interface definition must refine a class defined in a configuration. We, therefore, require that there is an isomorphism $int : \mathcal{C}_t \to INT$ for each tool configuration $t \in SDE$ that respects the following condition:

$$\bigwedge_{c \in \mathcal{C}_t} (name(c) = name(int(c))) \wedge (c \in NSC_t \Leftrightarrow kind(int(c)) = NSC) \tag{A.11}$$

We then require that terminal and nonterminal increment classes are used as leaf nodes of the inheritance hierarchy only.

$$\bigwedge_{c \in \mathcal{C}} kind(int(c)) \in \{TI, NTI\} \Rightarrow succ(c) = \emptyset \tag{A.12}$$

A.2.1 Export Interface

Context Free Syntax The export interfaces of different increment classes contain different sections. In order to exclude a number of potential specification errors, without having to define additional static semantic constraints the context-free syntax defines the different sections that may or have to be specified in the different classes.

```
abi_export_interface :
 'EXPORT' 'INTERFACE'
   opt_abs_syntax_section
   opt_attribute_section
   opt_sem_rel_section
   opt_method_section
 'END' 'EXPORT' 'INTERFACE' ';'.

ti_export_interface :
 'EXPORT' 'INTERFACE'
   reg_exp_section
   opt_attribute_section
   opt_sem_rel_section
   method_section
 'END' 'EXPORT' 'INTERFACE' ';'.
```

```
nti_export_interface :
 'EXPORT' 'INTERFACE'
   opt_abs_syntax_section
   unparsing_section
   opt_attribute_section
   opt_sem_rel_section
   method_section
 'END' 'EXPORT' 'INTERFACE' ';'.

nsc_export_interface :
 'EXPORT' 'INTERFACE'
   opt_attribute_section
   opt_method_section
 'END' 'EXPORT' 'INTERFACE' ';'.
```

The export interface of abstract increment classes contains a number of optional sections. We do not enforce any section here, in order to allow for abstract increment classes whose only purpose is to serve as a common super class of several other increment classes. For all properties that can be inherited from abstract increment classes, respective sections are provided, namely an abstract syntax section, an attribute section, a semantic relationship section and a method section. These sections have to be included in the interface of the abstract increment class since they contribute to the effective export interface of all subclasses.

The export of a non-terminal increment class has two mandatory sections, namely an unparsing section and a method section. The unparsing section is mandatory in order to enforce definition of the grammar of the tool's language. The method section is mandatory, since at least an implicit `init` method must be declared here, in order to be able to create increments of that class. The other sections are optional because the required property definitions could be inherited from super classes.

The export interfaces of terminal increment classes contain also two mandatory sections, namely a regular expression section and a method section. The method section is included for the same reason as for non-terminal increment interfaces. The regular expression section is required to define the lexical syntax of terminal increments. Again the other sections are optional, since the required declarations could be inherited from super classes.

The export interface of a non-syntactic class contains two optional sections, namely a construction section and a method section. The construction section defines how instances of the class are constructed from component instances. The method section defines the available methods for the non-syntactic class.

A.2.1.1 Properties

We now discuss the definition of properties, i.e. abstract syntax children, semantic relationships and attributes. Each of them is defined in a particular section of the class interface definition. All properties have in common that they have a name and a static type. The name will be used to uniquely identify the property and the type will be used for checking assignments against conformance to the polymorphism rule.

Contex-free Syntax We have to support single and multi-valued abstract syntax children. Single-valued children (production `elem_decl` on the next page) are required to model structure productions of a normalised EBNF. Multi-valued abstract syntax children are always ordered (production `list_decl`).

The cardinality of semantic relationships can be 1:n or m:n. GTSL does not include 1:1 semantic relationships, because they are a special case of 1:n relationships. In addition, they do not occur in practice. Semantic relationships are rather used to model dependencies between source and target increments such as use/declare, super class/subclass, import/export or outer/inner. Any target increment of these dependencies may possibly have multiple source increments: a variable or type may be used in multiple expressions, a class may have multiple subclasses, an export can be imported from multiple imports and an outer scoping block may have multiple inner blocks.

Semantic relationships will be specified as links in the semantic relationship section. The explicit link can be single-valued (symbol `elem_decl`) in order to specify a 1:n relationship or multi-valued (symbol `set_decl`) for a m:n relationship. If a value is assigned to an explicit link the relationship is established. The class defining a target increment of an explicit link may declare an implicit link (`impl_decl`). For an instance of the target class, it will contain all increments that have explicit links with the specified name to the target increment.

GTSL includes a rich variety of language constructs for attribute declarations. Attributes may be hidden if their existence is of no concern for other classes. Hidden attributes cannot be imported in client class specifications then. Attributes may be single-valued (production `elem_decl`) or multi-valued. For multi-valued attributes, type constructors for lists (`list_decl`), sets (set_decl), bags (bag_decl) and dictionaries (dict_decl) are offered. The context-free syntax of properties is then defined as follows:

```
abs_syntax_section :      attribute_section :       sem_rel_section :
 'ABSTRACT' 'SYNTAX'        'ATTRIBUTES'              'SEMANTIC' 'RELATIONSHIPS'
   child_list                 att_decl_list             sem_rel_list
 'END' 'ABSTRACT'           'END' 'ATTRIBUTES' ';' .  'END' 'SEMANTIC'
 'SYNTAX' ';' .                                       'RELATIONSHIPS' ';' .

att_decl :                att :                     sem_rel :
    'HIDDEN' att             elem_decl                 elem_decl
  / att .                  / list_decl               / set_decl
child :                    / set_decl                / impl_decl .
    elem_decl              / bag_decl
  / list_decl .            / dict_decl .

elem_decl : IDENTIFIER ':' IDENTIFIER ';' .
list_decl : IDENTIFIER ':' 'LIST' 'OF' IDENTIFIER ';' .
set_decl  : IDENTIFIER ':' 'SET' 'OF' IDENTIFIER ';' .
bag_decl  : IDENTIFIER ':' 'BAG' 'OF' IDENTIFIER ';' .
dict_decl : IDENTIFIER ':' 'DICTIONARY' 'OF' IDENTIFIER ';' .
impl_decl : 'IMPLICIT' IDENTIFIER ':' 'SET' 'OF' IDENTIFIER '.' IDENTIFIER ';' .
```

Hidden attributes could have been defined in the class specification. Then they would not only be logically hidden, but really not be seen at all by a tool builder developing a client class (only interfaces will be shown to him or her). This, however, would severely complicate the dependencies between classes during incremental class compilation. Consider that an attribute a was defined in the class specification of A. Interfaces of subclasses of A then not only depend on A's interface, but also on its specification. This is because they also inherit hidden properties. As a consequence, subclass interfaces could not be compiled, before the specification of A is complete. Moreover, they had to be compiled whenever the specification of A is changed. This would then cause a large chain of recompilations, since the specifications of the subclasses depend on its interfaces. In order to avoid these dependencies, we have for pragmatic reasons decided to support only logical hiding of attributes, but display their declaration in the interface.

Static Semantics For a class $c \in \mathcal{C}$, let E_c denote the set of properties that are declared in the class as children of the abstract syntax, attributes or as links of semantic relationships. We define a mapping $kind_c : E_c \to \{AS, AT, HID, EL, IL\}$ that keeps track of the section where a property is defined, since this is relevant for some static semantic properties.

$$e \mapsto \begin{cases} AS, & \text{if e is declared in an abstract syntax section} \\ AT, & \text{if e is declared in an attribute section and not hidden} \\ HID, & \text{if e is declared in an attribute section and hidden} \\ EL, & \text{if e is declared in a semantic relationship section as } \texttt{elem\_decl} \text{ or } \texttt{set\_decl} \\ IL, & \text{if e is declared as } \texttt{impl\_decl} \end{cases}$$

An abstract increment class that has declared a child in the abstract syntax cannot reasonably serve as a super class of a terminal increment class. Otherwise, the terminal increment class would inherit abstract syntax children which are meaningless. This is formally defined by the following condition:

$$\bigwedge_{c \in \mathcal{C}} \left(\bigvee_{e \in E_c} kind(e) = AS \right) \Rightarrow \bigwedge_{c_i \in succ^*(c)} kind(int(c_i)) \neq TI \tag{A.13}$$

We require that a property declared in a class $c \in \mathcal{C}$ is uniquely identified by a name. This is defined by mapping $name_c : E_c \to A^*$ with $e \mapsto$ string that matches the first occurrence of symbol `IDENTIFIER` in the respective declaration production. We are going to omit the index c if it is clear to which mapping *name* we refer. Uniqueness of property names is then formally defined for a class $c \in \mathcal{C}$ as follows:

$$\bigwedge_{e_1, e_2 \in E_c} name(e_1) = name(e_2) \Rightarrow e_1 = e_2 \tag{A.14}$$

As a prerequisite for the formal definition of inheritance of properties, we have to introduce the concepts of *types*. *Atomic types* are elements of the set $\mathcal{A} := \{bool, int, char, string, error\}$. The set of types $\mathcal{T}$of a tool specification is then defined as

$$\mathcal{T} := \mathcal{A} \cup \mathcal{C} \cup LIB \cup IMP \cup \bigcup_{t \in \mathcal{A} \cup \mathcal{C} \cup LIB \cup IMP} (list(t) \cup set(t) \cup bag(t) \cup dictionary(t))$$

The set of types, therefore, includes atomic types, tool-specific classes, pre-defined classes, imported classes and *multi-valued types* that implement *collections* of elements of a *base type*. Note, that $\mathcal{T}$does not include nested types like, $list(list(t))$. These nested types are not required, since any of these nested types must itself be a class in order to be able to define methods encapsulating them. If such a nested structure is to be modelled, a class for the inner list type is defined and then a list type constructor can be applied to that class to obtain the overall type.

A subtype relationship $\leq \subseteq (\mathcal{T} \times \mathcal{T})$ is then defined as follows.

1. if $t \in \mathcal{T} \cup \mathcal{A}$, then $t \leq t$.
2. if $t_1, t_2 \in \mathcal{C} \cup LIB \wedge t_2 \in pred^*(t_1)$ then elements of the subtype relationship are:

$$\begin{gathered} t_1 \leq t_2, \\ list(t_1) \leq list(t_2), \\ set(t_1) \leq set(t_2), \\ bag(t_1) \leq bag(t_2), \\ dictionary(t_1) \leq dictionary(t_2) \end{gathered}$$

We then define the *static type of properties* declared in a class as follows: $type_c : E_c \to \mathcal{T}$ with

$$e \mapsto \begin{cases} bool, & \text{if the second identifier in } \texttt{elem\_decl} \text{ matches } \texttt{'BOOLEAN'} \\ int, & \text{if the second identifier in } \texttt{elem\_decl} \text{ matches } \texttt{'INTEGER'} \\ char, & \text{if the second identifier in } \texttt{elem\_decl} \text{ matches } \texttt{'CHAR'} \\ string, & \text{if the second identifier in } \texttt{elem\_decl} \text{ matches } \texttt{'STRING'} \\ c', & \text{if the second identifier in } \texttt{elem\_decl} \text{ matches } name(c') \\ list(c'), & \text{if the second identifier in } \texttt{list\_decl} \text{ matches } name(c') \\ set(c'), & \text{if the second identifier in } \texttt{set\_decl} \text{ matches } name(c') \\ bag(c'), & \text{if the second identifier in } \texttt{bag\_decl} \text{ matches } name(c') \\ dictionary(c'), & \text{if the second identifier in } \texttt{dict\_decl} \text{ matches } name(c') \\ undefined, & otherwise \end{cases}$$

Again, we will omit the index c of *type* if the context is clear.

Abstract syntax children must be increments. Otherwise, the grammar of the language would not reasonably be defined, since neither atomic types nor non-syntactic classes have unparsing schemes. We, therefore, require:

$$\bigwedge_{e \in E_c} (kind(e) = AS) \Rightarrow \bigvee_{c_i \in \mathcal{C}} c_i = type(e) \wedge kind(int(c_i)) \in \{ABI, TI, NTI\} \tag{A.15}$$

Semantic relationships are only reasonable between increment classes. Non-syntactic classes are excluded since these classes define semantic structures that are used as attributes, i.e. that are local to an increment. Therefore, we require:

$$\bigwedge_{e \in E_c} (kind(e) = EL) \Rightarrow \bigvee_{c_i \in \mathcal{C}} c_i = type(e) \wedge kind(int(c_i)) \in \{ABI, TI, NTI\} \tag{A.16}$$

For implicit links (production `impl_decl`) declared in a class $c \in \mathcal{C}$, we have to assure that they correspond to the explicit link given in the declaration. Let *cl_id* denote the string that matches with the second occurrence in of symbol `IDENTIFIER` in production `impl_decl`. Let *expl* denote the string that matches with the third occurrence of symbol `IDENTIFIER` in production `impl_decl`. We then require:

$$\bigwedge_{e \in E_c} (kind(e) = IL) \Rightarrow \bigvee_{c_i \in \mathcal{C}} name(c_i) = cl\_id \wedge \bigvee_{e_{c_i} \in E_{c_i}} ((type(e_{c_i}) = c) \wedge (name_{c_i}(e_{c_i}) = expl)) \tag{A.17}$$

Properties are inherited from super classes. To formally define this inheritance, we define for each class $c \in \mathcal{C}$ the *set of inherited properties*:

$$E_c^* := E_c \cup \{e_i \in \bigcup_{c_i \in pred(c)} E_{c_i}^* \mid \bigwedge_{e \in E_c} name(e) \neq name(e_i)\}$$

The mappings *name*, *kind* and *type* are continued on the set of inherited properties as follows:

$$\begin{array}{ll} name^*: E_c^* \rightarrow A* & e \mapsto name_{c'}(e),\ if\ e \in E_{c'} \\ kind^*\ : E_c^* \rightarrow \{AS, HID, AT, EL, IL\} & e \mapsto kind_{c'}(e),\ if\ e \in E_{c'} \\ type^*\ : E_c^* \rightarrow \mathcal{T} & e \mapsto type_{c'}(e),\ if\ e \in E_{c'} \end{array}$$

Let $c_i, c \in \mathcal{C}$ be classes with $c_i \in pred^*(c)$. If there are two properties $e_i \in E_{c_i}$ and $e \in E_c$ with $name_c(e) = name_{c_i}(e_i)$, e is said to *redefine* e_i. The redefinition is statically correct, if e and e_i are of the same kind and the type of e is a subtype of e_i, i.e.:

$$kind_c(e) = kind_{c_i}(e_i) \wedge type_c(e) \leq type_{c_i}(e_i) \tag{A.18}$$

We are now going to formalise further correctness conditions to avoid incorrect multiple inheritance. Let $c \in \mathcal{C}$ be a class. Then the following condition must hold:

$$\bigwedge_{c_1, c_2 \in pred^*(c)} \ \bigwedge_{e_1 \in E_{c_1}, e_2 \in E_{c_2}} \left(\begin{array}{l} name_{c_1}(e_1) = name_{c_2}(e_2) \\ \Rightarrow \ \bigvee_{c_3 \in pred^*(c_1) \cap pred^*(c_2)} \ \bigvee_{e_3 \in E_{c_3}} name_{c_3}(e_3) = name_{c_1}(e_1) \end{array} \right) \tag{A.19}$$

The condition requires that if in two different super classes c_1 and c_2 of c different properties with the same names are declared that there must be a common super class c_3 of c_1 and c_2 that also declares a property with that name. Classes c_1 and c_2, then only redefine the property and, therefore, have to respect Rule A.18.

A further problem arises from the combination of property redefinition and multiple inheritance. Suppose there are two super classes c_1 and c_2 of class c. Assume further that c_1 and c_2 have a common super class c_3. Properties that are declared in c_3 are inherited in c via both c_1 and c_2. This situation is known as *repeated inheritance* and does not cause further problems. An ambiguity that must be avoided arises, if some property defined in class c_3 is redefined in c_1 or c_2. Then it is ambiguous whether the original declaration of the property from class c_3 or the redefined property is valid in c. This ambiguity is called *incorrect repeated inheritance*. We can only recover from that situation if the property is redefined in class c itself. In general, we require that the following condition holds for all classes $c \in \mathcal{C}$ in order to exclude incorrect repeated inheritance:

$$\bigwedge_{c_m \in \bigcap_{c_i \in pred(c)} pred^*(c_i)} \ \bigwedge_{e_m \in E_{c_m}} \left(\begin{array}{l} (\bigvee_{c_i \in pred(c)} \bigvee_{e_i \in E_{c_i}} name_{c_m}(e_m) = name_{c_i}(e_i)) \Rightarrow \\ \bigvee_{e \in E_c} name_c(e) = name_{c_m}(e_m)) \end{array} \right) \qquad (A.20)$$

We require for all properties e_m in all common super classes c_m of the direct super classes c_i of c that if e_m is redefined in some c_i it must also be redefined in c. Then this redefinition resolves the ambiguity. Obviously, this redefinition must obey also Rules A.18 and A.19.

A.2.1.2 Unparsing Schemes

Context-free Syntax The unparsing schema of a non-terminal increment class defines both, the output representation for unparsing and the input representation for parsing instances of that class. It consists of a list of unparsing items. An unparsing item can be a pretty printing item, a simple regular expression, a keyword or a component of the abstract syntax. Pretty printing items are given in round brackets. They are inserted during unparsing only. Regular expressions demand mandatory white spaces during both, unparsing and parsing. Keywords are enclosed in quotation marks. They are considered during parsing and unparsing. Finally, component items are declared by including the identifier of the respective abstract syntax child. If the child is a list increment, an additional delimiter specification can be defined. It allows for specification of items that have to be input or output between elements of the list.

```
unparsing_section :                          unparsing_item :
  'UNPARSING' 'SCHEME'                          pretty_printing
    unparsing_item_list                        / regdef
  'END' 'UNPARSING' 'SCHEME' ';' .             / keyword_item
                                               / component_item .

unparsing_item_list : unparsing_item / unparsing_item ',' unparsing_item_list .

pretty_printing :     "("\"(.)+\"")"| (NL) .
regdef :              WS[\*\+] .
keyword_item :        \"[a-zA-Z0-9.;:,\(\)\[\] ]*\" .
component_item :      IDENTIFIER opt_delimiter .
delimiter :           'DELIMITED' 'BY' delimiter_item_list 'END' .
delimiter_item :      pretty_printing / keyword_item / regdef .
```

Static Semantics The items that can occur as components in an unparsing schema must be abstract syntax children. We do not allow attributes or links of semantic relationships as component items for the following reasons: Firstly, attributes and semantic relationships define static semantic properties whereas the unparsing scheme contributes to the definition of the syntax of the underlying language. Secondly, we do not foresee that attribute values need to be displayed in-line in documents. Instead increments will be underlined in order to visualise semantic errors and a dedicated window will be used to display detailed error messages, i.e. contents of increments' attribute values. Finally, links of semantic relationships may well lead to enclosing increments and could thus cause a cyclic unparsing that never terminates.

Let *child* denote the character string that matches symbol `IDENTIFIER` in a `component_item` production of class c. Then the following condition must hold:

$$\bigvee_{e \in E_c^*} name^*(e) = child \ \wedge \ kind^*(e) = AS \tag{A.21}$$

This condition assures that component items in an unparsing schema denote children of the abstract syntax. Such a child need not be declared in the class itself, but can be inherited from a super class as well.

Delimiter are only meaningful, if they are applied to abstract syntax children that have been constructed with a list type constructor. We, therefore, exclude any other application of delimiter items. Let *child* denote the character string that matches the identifier in a `component_item` production of class c. If the optional delimiter after *child* is provided, the following condition must hold:

$$\bigvee_{e \in E_c^*} \left(name^*(e) = child \ \wedge \bigvee_{c \in \mathcal{C}} type(e) = list(c) \right) \tag{A.22}$$

A.2.1.3 Regular Expressions

Context-free Syntax A regular expression is defined for each terminal increment class in a regular expression section. This regular expression defines the lexical syntax of terminal increments that are instances of this class. The complete syntax definition is for reasons of brevity omitted here. It is identical to the syntax of extended regular expressions used in UNIX commands such as `ed`, `egrep`, or `lex`.

```
reg_exp_section:  'REGULAR' 'EXPRESSION'
                    regular_exp
                  'END' 'REGULAR' 'EXPRESSION' ';' .
```

A.2.1.4 Methods

Context-free Syntax The method section of a class export interface defines the methods that can be invoked on instances of that class or its subclasses. Methods are defined by a name, a parameter list and an optional result type. The methods may be *implicit.* Then their body is generated by the tool generator based on the definitions given in other sections. Methods can be *deferred.* Then their body need not be defined, but the methods must be redefined in its subclasses. Other methods are called *explicit.* Their bodies have to be determined in the specification part. The context-free syntax is then defined as follows:

```
method_section : 'METHODS' method_list 'END' 'METHODS' ';' .

method :        opt_category 'METHOD' IDENTIFIER
                '(' opt_param_list ')' opt_result_type ';' .

category    :    'DEFERRED' / 'IMPLICIT' / 'HIDDEN' .
param_list  :    parameter / parameter ';' parameter_list .
param       :    IDENTIFIER ':' type .
result_type :    ':' type .
type        :    IDENTIFIER / list / set / bag / dict .
list        :    'LIST' 'OF' IDENTIFIER .
set         :    'SET' 'OF' IDENTIFIER .
bag         :    'BAG' 'OF' IDENTIFIER .
dict        :    'DICTIONARY' 'OF' IDENTIFIER.
```

Static Semantics For a class $c \in \mathcal{C}$, let M_c denote the set of methods that have been declared within the method section of a class.

We define mappings $kind_c : M_c \to \{EXP, HID, IMP, DEF\}$ with

$$m \mapsto \begin{cases} EXP, & \text{if no } \texttt{category} \text{ is defined for } m \\ HID, & \text{if } \texttt{category} \text{ of } m \text{ is } \texttt{'HIDDEN'} \\ IMP, & \text{if } \texttt{category} \text{ of } m \text{ is } \texttt{'IMPLICIT'} \\ DEF, & \text{if } \texttt{category} \text{ of } m \text{ is } \texttt{'DEFERRED'} \end{cases}$$

It is unreasonable to define deferred methods in terminal or non-terminal increment classes. These classes cannot have subclasses due to Rule A.12. Deferred methods could then never be explicitly defined. We, therefore, require for all classes $c \in \mathcal{C}$:

$$kind(int(c)) \in \{TI, NTI\} \Rightarrow \bigwedge_{m \in M_c} kind_c(m) \neq DEF \tag{A.23}$$

Implicit methods provide a homogeneous specification interface to the various other sections. Some of these sections are not available in abstract increment classes. Moreover, information given in available sections is still incomplete and will be completed or redefined in the respective sections of subclasses. We, therefore, do not support the declaration of implicit methods in abstract increment classes and impose the following condition on all classes $c \in C$:

$$kind(int(c)) = ABI \Rightarrow \bigwedge_{m \in M_c} kind_c(m) \neq IMP \tag{A.24}$$

We require that a method is uniquely identified within the scope of a class by the method name. This implies that we do not want to support overloading of methods by choosing different parameter types, like in C++. If the methods have different parameter types, they will perform different operations and, therefore, should be given different names. To formalise this requirement, let $c \in \mathcal{C}$ be a class, then names of methods in that class are defined by $name_c : M_c \to A^*$ with $m \mapsto$ lexical value that matches with symbol `IDENTIFIER` in production `method`. Uniqueness of method names is then defined as:

$$\bigwedge_{m_1, m_2 \in M_c} name(m_1) = name(m_2) \Rightarrow m_1 = m_2 \tag{A.25}$$

A method may have an optional result type. This result type is defined by mapping $mtype : M_c \rightarrow \mathcal{T}$ with

$$m \mapsto \begin{cases} bool, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'BOOLEAN'} \\ int, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'INTEGER'} \\ char, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'CHAR'} \\ string, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'STRING'} \\ c', & \text{if the identifier in } \texttt{type} \text{ matches } name(c') \\ list(c'), & \text{if the identifier in } \texttt{list} \text{ matches } name(c') \\ set(c'), & \text{if the identifier in } \texttt{set} \text{ matches } name(c') \\ bag(c'), & \text{if the identifier in } \texttt{bag} \text{ matches } name(c') \\ dictionary(c'), & \text{if the identifier in } \texttt{dict} \text{ matches } name(c') \\ undefined, & otherwise \end{cases}$$

Let P_c denote the set of all parameters of class $c \in \mathcal{C}$. Let $P_c^i := \overbrace{P_c \times \ldots \times P_c}^{i\ times}$ denote the set of all i tuples of parameters. A parameter list of length i is then formally considered as an element of P_c^i. Let $P_c^* := \bigcup_{i \in I\!N} P_c^i$ be the set of all possible parameter lists. Let $|| : P_c^* \rightarrow I\!N$ with $pl \mapsto i$, if $pl \in P_c^i$ denote the cardinality of a tuple from P_c^*. Let further $[] : P_c^* \times I\!N \rightarrow P_c$ with $pl[j] \mapsto p_j$ if $pl = \langle p_1, \ldots, p_j, \ldots, p_{|pl|} \rangle$ denote the j^{th} element in a tuple. Each method of class $c \in \mathcal{C}$ is associated with its parameter list. This is formally defined by the mapping $params_c : M_c \rightarrow P_c^*$ with $m \mapsto pl$ and $|pl|$ = number of elements in the parameter list. Parameters have a name that is defined by $name_c : P_c \rightarrow A^*$ with $p \mapsto$ character string that matches the `IDENTIFIER` in production `param` that generated the respective parameter. We then require that parameters are uniquely identified by their name within the scope of their methods:

$$\bigwedge_{m \in M_c} \bigwedge_{pl \in params(m)} \bigwedge_{i,j \in \{1,\ldots,|pl|\}} name(pl[i]) = name(pl[j]) \Rightarrow i = j \qquad \text{(A.26)}$$

A type is associated with a parameter list. Similar to result types of methods, this type is defined by mapping $type_c : P_c \rightarrow \mathcal{T}$ with

$$p \mapsto \begin{cases} bool, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'BOOLEAN'} \\ int, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'INTEGER'} \\ char, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'CHAR'} \\ string, & \text{if the identifier in } \texttt{type} \text{ matches } \texttt{'STRING'} \\ c', & \text{if the identifier in } \texttt{type} \text{ matches } name(c') \\ list(c'), & \text{if the identifier in } \texttt{list} \text{ matches } name(c') \\ set(c'), & \text{if the identifier in } \texttt{set} \text{ matches } name(c') \\ bag(c'), & \text{if the identifier in } \texttt{bag} \text{ matches } name(c') \\ dictionary(c'), & \text{if the identifier in } \texttt{dict} \text{ matches } name(c') \\ undefined, & otherwise \end{cases}$$

The implementation of implicit method bodies is generated from the various other sections, which a tool builder has specified. This implementation must match a particular signature, i.e method types and parameter types are pre-defined. The signature has to be included in the interfaces of increment classes in order to check correctness of the way implicit methods are used. These methods cannot be defined as deferred methods in e.g. a pre-defined increment class, because the signatures vary from class to class. Since the signature of implicit method bodies cannot be adapted to the one given in the interfaces (the implementation of the generator will not be changeable), we will have to force the tool builder to obey in the interface particular signatures for implicit methods. This is defined by a number of static semantic rules. For all classes $c \in C$, implicit methods have to obey particular names:

$$\begin{aligned} kind(int(c)) = NTI \quad &\Rightarrow \bigwedge_{m \in M_c} \begin{array}{l} kind(m) = IMP \Rightarrow \\ name(m) \in \{'init','expand','parse','unparse','collapse'\} \end{array} \\ kind(int(c)) = TI \quad &\Rightarrow \bigwedge_{m \in M_c} \begin{array}{l} kind(m) = IMP \Rightarrow \\ name(m) \in \{'init','scan','unparse','collapse'\} \end{array} \end{aligned} \tag{A.27}$$

Let $inc := cl$ with $name(cl) =' Increment'$ be the pre-defined increment class `Increment`. For all classes $c \in \mathcal{C}$ and all implicit methods $m \in M_c$, the following condition must hold for parameter list and result types of m:

$$name(m) = \begin{cases} 'init' & \Rightarrow \ |params(m)| = 1 \wedge type(params(m)[1]) = inc \\ 'expand' & \Rightarrow \ |params(m)| = 0 \\ 'collapse' & \Rightarrow \ |params(m)| = 0 \\ 'scan' & \Rightarrow \ |params(m)| = 1 \wedge type(params(m)[1]) = string \wedge \\ & \quad\ type(m) = bool \\ 'parse' & \Rightarrow \ |params(m)| = 1 \wedge type(params(m)[1]) = string \wedge \\ & \quad\ type(m) = c \\ 'unparse' & \Rightarrow \ |params(m)| = 0 \wedge type(m) = string \end{cases} \tag{A.28}$$

Similar to properties, methods are inherited from super classes. The formal definition is straight-forward. For each class $c \in \mathcal{C}$, the set of inherited methods is

$$M_c^* := M_c \cup \{m_i \in \bigcup_{c_i \in pred(c)} M^*c_i \mid \bigwedge_{m \in M_c} name_c(m) \neq name_{c_i}(m_i)\}$$

Similar to properties, mappings $name_c, kind_c, type_c$ and $params_c$ are continued on M_c^* and denoted as $name^*, kind^*, type^*$ and $params^*$ respectively.

As a consequence of being able to redefine properties, a tool builder requires redefinition of methods as well. In particular, parameter and result types of methods need to be specialised. Otherwise, a parameter p may not be assignable to a redefined property due to the fact that assignments are restricted to subtypes.

The static semantic condition for redefinition of methods is slightly more complicated for methods. Let $c_i, c \in \mathcal{C}$ be classes with $c_i \in pred^*(c)$. If there are two methods $m_i \in M_{c_i}$ and $m \in M_c$ with $name_c(m) = name_{c_i} m_{c_i}$ then m is said to redefine m_i. This redefinition is correct, only if m and m_i have parameter lists of the same length, the redefinition respects the visibility, the result type of m is a subtype of the respective type in m_i and each parameter type in the parameter list of m is a subtype of the respective parameter type in m_i:

$$\begin{array}{ll} & |params_c(m)| = |params^*(m_i)| \\ \wedge & kind_c(m) = HID \Leftrightarrow kind_{c_i}(m_i) = HID \\ \wedge & type_c(m) \leq type_{c_i}(m_i) \\ \wedge & \bigwedge_{j \in \{1,\ldots,|params_c(m)|\}} type_c(params_c(m)[j]) \leq type_{c_i}(params(m_i)[j]) \end{array} \tag{A.29}$$

Similar to properties, ambiguities of methods can occur due to multiple inheritance. They have to be either avoided or be resolved by redefinition for the same reasons as for properties. The definition of the respective two static semantic rules is straight-forward:

$$\bigwedge_{c_1,c_2 \in pred^*(c)} \bigwedge_{m_1 \in M_{c_1}, m_2 \in M_{c_2}} \left(\begin{array}{l} name_{c_1}(m_1) = name_{c_2}(m_2) \Rightarrow \\ \bigvee_{c_3 \in pred^*(c_1) \cap pred^*(c_2)} \bigvee_{m_3 \in M_{c_3}} name_{c_3}(m_3) = name_{c_1}(m_1) \end{array} \right) \tag{A.30}$$

$$\bigwedge_{c_m \in \bigcap_{c_i \in pred(c)} pred^*(c_i)} \bigwedge_{m_m \in M_{c_m}} \left(\begin{array}{l} (\bigvee_{c_i \in pred(c)} \bigvee_{m_i \in M_{c_i}} name_{c_m}(m_m) = name_{c_i}(m_i)) \Rightarrow \\ \bigvee_{m \in M_c} name_c(m) = namec_m(m_m)) \end{array} \right) \tag{A.31}$$

If a method is declared as deferred, it must be redefined in all subclasses so as to assure that dynamic binding (c.f. Page 284) will find a method implementation. In abstract classes, this redefinition can be achieved with yet another deferred method or with an explicit method. In terminal or non-terminal classes, however, Condition A.23 assures that the redefinition is done with an explicit or implicit method. We, therefore, require that for all classes $c \in \mathcal{C}$ the following condition holds:

$$\bigwedge_{m \in M_c} \left(kind_c(m) = DEF \Rightarrow \bigwedge_{c_i \in succ^*(c)} \bigvee_{m_i \in M_{c_i}} name_{c_i}(m_i) = name_c(m) \right) \tag{A.32}$$

In Condition (A.54), we will define that no instances of abstract classes are created. Under this assumption, declaring a deferred method, thus assures that the method is effectively available for all instances of all subclasses, without having to specify an artificial implementation in the class where the method was first introduced.

Dynamic Semantics Implicit methods have the following semantics: The method `init` is executed upon construction of new increments. It initialises all properties of the increment according to the discussion in Subsection A.3.2. One property, that is inherited from the pre-defined class `Increment` is the abstract syntax father. Property `father` stores a reference to the enclosing increment. The father is initialised upon construction with the parameter that is passed to the `init` method. After creation, each increment represents a place holder and child increments do not yet exist. Method `expand` changes this and creates all child increments of an increment. Method `collapse` performs the reverse operation, i.e. it deletes all child increments and transforms the increment back to the state where it is a place holder. Method `scan` is used to check for lexical correctness of terminal increments. The regular expression section of the class that contains the method definition of scan is transformed into a finite automaton. The automaton reads the string that is passed as parameter. Method `scan` returns `true`, if the automaton accepts the string and `false` otherwise. If the string is accepted it is stored in the attribute `value`. Method `parse` checks whether the string that is passed as an argument can be generated by the grammar that is induced by abstract syntax definitions and unparsing schemes of the class and its child increment classes. If the string is correct, `parse` will generate a tree of increments and return the root increment of that tree. Otherwise `parse` will return `NIL`. Method `unparse` performs the reverse operation and translates an increment tree into its textual equivalent according to the unparsing scheme.

A.2.2 Import Interface

Context-free Syntax The dependencies between class interfaces must be explicitly declared. Dependencies exist due to use of other classes in e.g. property types. Considering reuse of GTSL classes, the declaration of class dependencies eases the identification of other classes that either have to be replaced or to be reused as well. Regarding configuration management, the declaration of dependencies eases the identification of the other classes belonging to the configuration. The dependencies could equally well be inferred from the use of types. Forcing a tool builder to explicitly specify them, however, forces him or her to be aware of the dependencies. This is the first step for minimising them.

```
import_interface_section :  'IMPORT' 'INTERFACE'
                               import_list
                            'END' 'IMPORT' 'INTERFACE' ';'
import :                    'IMPORT' IDENTIFIER ';'
```

Static Semantics Let I_c denote the set of all imports. The mapping $name_c : I_c \rightarrow A^*$ with $i \mapsto$ lexical value that matches with terminal `IDENTIFIER` in production `import` defines the import name. We then require for all classes $c \in \mathcal{C}$ that the import interface does not include redundant imports:

$$\bigwedge_{i_1, i_2 \in I_c} name_c(i_1) = name_c(i_2) \Rightarrow i_1 = i_2 \tag{A.33}$$

Moreover, we require that the imported class exists:

$$\bigwedge_{i \in I_c} \bigvee_{c' \in \mathcal{C} \cup IMP \cup LIB} name_c(i) = name(c') \tag{A.34}$$

I_c induces the set of types $T_c \subseteq \mathcal{T}$ that are *valid* in the interface of class $c \in \mathcal{C}$. Let, therefore, $IC := \{c \in \mathcal{C} \cup IMP \cup LIB \mid \bigvee_{i \in I_c} name(c) = name_c(i)\}$ denote the set of those classes that have been imported in the interface. Then the set of valid types are the atomic types, the inherited classes, the imported classes and any class that can be constructed from these using GTSL's type constructors:

$$T_c := \mathcal{A} \cup pred^*(c) \cup IC \cup \bigcup_{t \in pred^*(c) \cup IC \cup \mathcal{A}} (list(t) \cup set(t) \cup bag(t) \cup dict(t))$$

We then require that any type used in a property declaration, method or parameter type of the class interface must be valid:

$$\bigwedge_{c \in \mathcal{C}} \bigcup_{e \in E_c} type(e) \cup \bigcup_{m \in M_c} type(m) \cup \bigcup_{p \in P_c} type(p) \subseteq T_c \tag{A.35}$$

A.3 Class Specifications

We have now completed the definition of those parts of GTSL that express class interface definitions, i.e. the public part of a GTSL class. We are now going to define the language components for class specifications, which are used for specifying the hidden properties of a GTSL class.

Context-free Syntax There are two different kinds of class specifications, namely increment and non-syntactic classes specifications. Some properties have to be specified for increment classes while they are irrelevant for non-syntactic classes. These properties are semantic rules, commands and attribute initialisations. Semantic rules are not required in non-syntactic classes because there are no attributes or semantic relationships whose values had to be determined. Constructs for attribute initialisations are obsolete for the same reason. Since instances of non-syntactic classes are not visible at the user-interface, but only used as attributes of increment classes, it is unreasonable to allow a tool builder defining commands for non-syntactic classes. We, therefore, syntactically distinguish specifications of non-syntactic classes from increment class specifications. We, however, need not distinguish the various kinds of increment classes, namely abstract, terminal and non-terminal classes, since all increment specification sections must be available for any of these.

```
class_specification :   increment_spec
                       / non-syntactic_spec

increment_spec :       'INCREMENT' 'SPECIFICATION' IDENTIFIER ';'
                         opt_import_interface_section
                         opt_attribute_section
                         opt_semantic_rule_section
                         opt_method_section
                         opt_interaction_section
                       'END' 'INCREMENT' 'SPECIFICATION' IDENTIFIER '.' .

non-syntactic_spec :   'SPECIFICATION' IDENTIFIER ';'
                         opt_import_interface_section
                         method_section
                       'END' 'SPECIFICATION' IDENTIFIER '.' .
```

Static Semantics A class specification must match with one and only one class interface definition. To define this more precisely, let $SPEC$ be the set of increment class specifications. Let $name : SPEC \rightarrow A^*$ with $sp \mapsto$ string that matches with first occurrence of symbol `IDENTIFIER` in production `increment_spec` and `non-syntactic_spec`, respectively. Let furthermore $kind : SPEC \rightarrow \{INC, NSC\}$ be a mapping that defines whether the specification is an increment specification or the specification of a non-syntactic class according to the choice in production `class_specification`. We then require an isomorphism $spec : \mathcal{C} \rightarrow SPEC$ to exist that satisfies the following condition:

$$\bigwedge_{c \in \mathcal{C}} \begin{array}{c} spec\_name(spec(c)) = name(c) \wedge \\ kind(spec(c)) = NSC \Leftrightarrow c \in NSC \end{array} \qquad \text{(A.36)}$$

A.3.1 Import Interface

Context-free syntax We include an import interface into class specifications for the same reason as for interfaces. We want to support a tool builder in reusing class specifications and ease configuration management of classes. Therefore, we have to make the dependencies to other classes explicit. Opposed to class interface definitions that only use classes in property declarations, we will have to anticipate the use of properties and methods in path expressions of specifications. The specification import interface, therefore, not only imports classes, but also properties and methods. The context-free syntax is as follows:

```
spec_import_interface_section : 'IMPORT' 'INTERFACE'
                                  spec_import_list
                                'END' 'IMPORT' 'INTERFACE' ';' .

spec_import :                   'IMPORT' IDENTIFIER opt_including ';'  .

including :                     'INCLUDING' ident_list .
```

Static Semantics Let SI_c denote the set of all specification imports. Mapping $name_c : SI_c \rightarrow A^*$ with $i \mapsto$ string that matches symbol `IDENTIFIER` in production `spec_import`. We do not want to have redundant imports:

$$\bigwedge_{i_1,i_2 \in SI_c} name(i_1) = name(i_2) \Rightarrow i_1 = i_2 \tag{A.37}$$

Let IEM be the set of imported properties or methods. Mapping $incl : SI_c \rightarrow \mathcal{P}(IEM)$ associates a set of imported properties and methods with each specification import according to symbol `ident_list` in production `including`. Moreover, $name : IEM \rightarrow A^*$ returns the string for an imported property or method that matches with symbols `IDENTIFIER` in production `ident_list`. Obviously, imported classes must exist and export the respective imported properties and methods:

$$\bigwedge_{i \in SI_c} \bigvee_{c' \in \mathcal{C}} \begin{array}{c} name(i) = name(c') \wedge \\ \bigwedge_{iem \in incl(i)} \bigvee_{d \in E_c^* \cup M_c^*} name(iem) = name^*(d) \wedge kind^*(d) \neq HID \end{array} \tag{A.38}$$

The import interface induces the set of types $T_c^s \subseteq \mathcal{T}$ that are *visible* in a class specification. Let, therefore, $IC^s := \{c \in \mathcal{C} \cup IMP \cup LIB \mid \bigvee_{i \in SI_c} name(i) = name(c)\}$ be the set of imported classes of a specification. Similar to interfaces, the set of valid types of a specification is then:

$$T_c^s := \mathcal{A} \cup pred^*(c) \cup IC^s \cup \bigcup_{t \in pred^*(c) \cup IC^s \cup \mathcal{A}} (list(t) \cup set(t) \cup bag(t) \cup dictionary(t))$$

We define a mapping $decl : IEM \rightarrow \bigcup_{c \in \mathcal{C} \cup IMP \cup LIB} E_c^* \cup M_c^*$ with

$$em \mapsto \begin{cases} m & \text{if } em \in incl(i) \wedge name(i) = name(c) \wedge m \in M_c^* \wedge name^*(m) = name(em) \\ e & \text{if } em \in incl(i) \wedge name(i) = name(c) \wedge e \in E_c^* \wedge name^*(e) = name(em) \end{cases}$$

This mapping associates imported properties or methods with their corresponding exports. It is now used to define the set of properties and methods that are *visible* in a class $c \in \mathcal{C}$. This set $EM_c \subseteq \bigcup_{c \in \mathcal{C}^s} E_c^* \cup M_c^*$ is then defined as:

$$EM_c := \bigcup_{iem \in \bigcup_{i \in SI_c} incl(i)} decl(iem)$$

A.3.2 Property Initialisation

Upon creation of increments, the properties declared in the increment's class will be initialised. Properties of atomic types will be initialised as follows: `BOOLEAN` $\rightsquigarrow$ `FALSE`, `INTEGER` $\rightsquigarrow$ `0`, `CHAR` $\rightsquigarrow$ `' '`, `STRING` $\rightsquigarrow$ `""` and `ERRORS` $\rightsquigarrow$ `#OK`. A property whose type is multi-valued is initialised with the respective empty collection. A property whose type is a class is initialised with the undefined value `NIL`.

There may be situations, where a tool builder wants to define an initial value for a property that differs from the default. To anticipate these situations, we include an initialisation section into the specification of an increment class.

Context-free Syntax The initialisation section syntactically consists of a list of assignments. The left-hand-side of the assignment operator has to be a property of the class, the right-hand-side can either be an expression or the creation of a new increment or attribute.

```
attribute_section : 'INITIALIZATION'
                       attribute_init_list
                    'END' 'INITIALIZATION' ';' .

attribute_init :    assignment .
```

Static Semantics For the definition of static semantics of assignments as well as for the other statements, we have to formalise the concept of *scope*. We consider the scope of a statement to be the set of those declarations that are visible for the statement. These declarations could be properties of the class in which the statement is contained, parameters of a method if the statement is contained in a method body, quantor variables if the statement is part of a semantic rule, cursors of iteration statements, or local variables that are declared in an interaction, method or semantic rule. We, therefore, define a mapping $scope : STMT \rightarrow \mathcal{P}(DECL)$ where $STMT := ASS \uplus LOOP \uplus IT \uplus BRA \uplus CAS \uplus PE \uplus RET$ is the disjoint union of the various sets of statements and $DECL := PAR \uplus VAR \uplus QVAR \uplus CU \uplus \bigcup_{c \in \mathcal{C}} E_c$ is the set of declarations. Furthermore, we define two mappings $name : DECL \rightarrow A^*$ and $type : DECL \rightarrow \mathcal{T}$ that associate names and types with declarations. For all statements contained in a class $c \in \mathcal{C}$ and all properties $e \in E_c$ these mappings are defined as follows:

$$\begin{aligned} name(e) &:= name_c(e) \\ type(e) &:= type_c(e) \end{aligned}$$

As we go along with the definition of static semantics of the various sections, we will incrementally complete mappings *scope*, *name* and *type*. For assignments $ass \in ASS$ contained in the initialisation section of a class $c \in \mathcal{C}, scope(ass) := E_c^*$.

Dynamic Semantics Upon creation of an increment, the initialisations defined in this section are executed. In addition, initialisation sections defined in super classes are executed in order to initialise inherited properties. These inherited initialisations are executed in the order down the class hierarchy, i.e. before the execution of an initialisation section of a class, all initialisations defined in super classes are executed. This enables a tool builder to redefine an initialisation of an inherited property in a subclass.

To complete the static semantics of initialisations, we have to define the polymorphism rule for assignments. This is not possible for the moment, because we have not yet introduced expressions and their types. We will, therefore, postpone the definition to the discussion of statements and consider expressions first. The definition of expressions, in turn, requires the introduction of path expressions and we now continue with these.

A.3.3 Path Expressions

The purpose of path expressions in an increment class is to statically determine navigation paths to other increments that are possibly contained even in other documents. This is required, in particular, for instantiating semantic relationships, e.g. in semantic rules. A path expression consists of a sequence of steps, which are delimited by a dot. A step, in turn, can be a navigation to an abstract syntax child, to the abstract syntax father or along a link of a semantic relationship. A step can also be a method call. This supports structuring path expressions, since subpaths can then be defined in methods. As methods can be called recursively, path expressions can be recursive as well.

Path expressions may be method calls. A method call is always resolved like in Eiffel or O_2C using dynamic binding. This means that the choice which method is invoked depends on the dynamic class of the increment that is referenced by the proceeding subpath. If no method is found in that class, the direct super classes are considered and so on. Sometimes, a tool builder wants to influence this search strategy and start at a particular class. As an example consider that a method is redefined in a subclass. The implementation of the redefinition often can rely on the original definition as it is often a specialisation. In Smalltalk, the `super` directive is offered for this purpose [Gol85]. We take the approach of O_2C, which is more general. We allow a tool builder to explicitly address the class in which the method search is to be started. Therefore, method names may be followed by an `@` sign and a class name.

Context-free Syntax The context-free syntax of path expressions is then defined as follows:

```
path_expr :                 '<' IDENTIFIER '>' uncasted_path_expr
                          / uncasted_path_expr .

uncasted_path_expr :        step
                          / uncasted_path_expr '.' step
                          / '(' '<' IDENTIFIER '>' uncasted_path_expr ')' '.' step .

step :                      IDENTIFIER
                          / method_name '(' opt_actual_parameter_list ')' .

method_name :               IDENTIFIER
                          / IDENTIFIER '@' IDENTIFIER .

actual_parameter_list :     actual_parameter
                          / actual_parameter ',' actual_parameter_list .

actual_parameter :          expression
```

Static Semantics Let S denote the set of all steps that occur in paths. A step has a name that is defined by $name : S \rightarrow A^*$ with $s \mapsto$lexical value that matches the first `IDENTIFIER` in productions `step` or `method_name`. A path expression is then considered as a tuple of steps, i.e. as an element of $PE \subseteq S^*$. Similar to parameter lists, the mappings $[] : S^* \times I\!N \rightarrow S$ and $|| : S^* \rightarrow I\!N$ denote projection and path lengths.

The first step in a path then has to denote a valid variable or property declaration. In addition, it can denote the pre-defined variable $self$ that always refers to the increment for which currently a method, interaction or semantic rule is executed. This variable is in particular required to facilitate recursive method calls. To formalise these concerns, we require:

$$\bigwedge_{pe \in PE} name(pe[1]) =' SELF' \vee \bigvee_{e \in scope(pe)} name(e) = name(pe[1]) \tag{A.39}$$

We then require that successive steps in path expressions are properly imported. This requirement not only assures that the import interface reflects the actual use of other classes' properties and methods, but also, that the property or method referenced in a step is exported by the respective class (due to Condition A.38).

$$\bigwedge_{pe \in PE} \bigwedge_{i \in \{2,\ldots,|pe|\}} \bigvee_{em \in EM_c} name(em) = name(pe[i]) \tag{A.40}$$

The type of a path expressions is defined in three steps. Mapping *stype* defines the type of steps contained in a path expression without considering the existence of type casts. Mapping *stype'* then considers the existence of casts and finally, *petype* defines the type of a path expression based on the type of the result that *stype'* returns when applied to the last step.

Mapping $stype : PE \times I\!N \mapsto \mathcal{T}$ recursively defines the type of a step without considering type casts:

$$(pe, i) \mapsto \begin{cases} type(d), & if \quad i = 1 \wedge d \in scope(pe) \wedge name(d) = name(pe[1]) \\ type_{stype'(pe[i-1])}(e) & if \quad e \in E^*_{stype'(pe[i-1])} \wedge name_c(e) = name(pe[i]) \\ type_{stype'(pe[i-1])}(m) & if \quad m \in M^*_{stype'(pe[i-1])} \wedge name_c(m) = name(pe[i]) \end{cases}$$

The type of the first step is determined by the type of the declaration referenced by the name of the first step. Successive steps are considered as references to exported properties or methods of the class that is identified by the type of the proceeding step (with considering casts). Their type is thus the type of the referenced property or method.

To assure well-definedness of *stype*, we have to assure that the type of any but the last step is a class. Otherwise the indexes of sets E and M in the definition of *stype* would be meaningless. We, therefore, require:

$$\bigwedge_{pe \in PE} \bigwedge_{i \in \{1,\ldots,|pe|-1\}} \bigvee_{c \in \mathcal{C}} stype(pe, i) = c \tag{A.41}$$

Let s be a step in a path expression *pe* at position i. If s is a method call applied with an `@` sign followed by an identifier whose lexical value is the string *id*, we have to assure that *id* is the name of a super class of the step before s. To enforce safe use of this concept, we have furthermore to require that the method is imported, i.e.:

$$\bigvee_{c \in \mathcal{C}} \left(name(c) = id \wedge c \in pred^*(stype(s, i \Leftrightarrow 1)) \wedge \bigvee_{m \in EM_c} name(s) = name(m) \right) \tag{A.42}$$

We now formally define type casts and continue the definition of *stype* in *stype'*. Let, therefore, TC be the set of type casts. Mapping $name : TC \rightarrow A^*$ with $tc \mapsto$ lexical value of the identifier that matches the `IDENTIFIER` symbol between the pointed brackets. We require that type casts denote imported classes only:

$$\bigwedge_{tc \in TC} \bigvee_{c \in \mathcal{C} \cap T^s_c} name(tc) = name(c) \tag{A.43}$$

Let $class : TC \to \mathcal{C}$ with $tc \mapsto c$ where $name(tc) = name(c)$ denote the class identified by a cast. A type cast belongs to exactly one path. The path that includes the cast is identified by $inpath : TC \to PE$ with $tc \mapsto$ the path expression that syntactically includes the path. A cast is applied to one step of a path. This is reflected by mapping $pos : TC \to I\!N$ with $tc \mapsto$ the position of the last step in the subpath to which tc is applied. We can then formally determine the type of a step that takes casts into account as $stype' : PE \times I\!N \to \mathcal{T}$ with

$$(pe, i) \mapsto \begin{cases} class(tc) & if\ inpath(tc) = pe \wedge pos(tc) = i \\ stype(pe, i) & otherwise \end{cases}$$

In order to restrict the applicability of type casts to those cases where they are really required and thus support their safe use, we require that casts can only be performed down the class hierarchy, i.e. casts to brother or sister classes are inhibited:

$$\bigwedge_{tc \in TC} class(tc) \leq stype(inpath(tc), pos(tc)) \tag{A.44}$$

The type of a path expression is then defined as $petype : PE \to \mathcal{T}$ with $pe \mapsto stype'(pe, |pe|)$.

Dynamic Semantics Static semantics Condition A.39 requires that the first step in a path expression is a property or a variable. During execution, the value of that property or variable becomes the first *execution context*. Note, that according to static semantic Condition A.41, the execution context is always an instance of an increment class. Due to Condition A.54 it will always be a terminal or non-terminal increment. The next execution context of a path expression is determined by interpreting the next step. If the step is a property, the value of that property becomes the execution context. Otherwise, the step denotes a method of an increment class. Then the method is invoked for the execution context. *Dynamic binding* is used for searching the method body. During dynamic binding, search starts at the class specified after the `@` sign, if any. Otherwise it begins at the dynamic class of the current execution context. In case no method body is found there, the super classes are considered. Note, that the static semantic definition for method declarations assures that there will be one and only one method found. Opposed to Smalltalk, where run-time errors can occur due to unbound methods, GTSL is safe to this respect. The next execution context will then be the increment that is returned by the method. Determining the next execution context thus navigates from one increment to some other one. The dynamic semantics of a complete path expression, in turn, is the interpretation of the last step in the last execution context. Note, that this can be also an instance of a non-syntactic class or an atomic value due to Condition A.41.

Elements of actual parameter lists that can follow method calls in path expressions are considered as expressions. To complete the formal definition of path expressions we, therefore, have to define expressions.

A.3.4 Expressions

Context-free Syntax The context-free syntax is as follows:

```
expr :      bool_expr / int_expr / path_expr / constant / '(' expr ')' .
bool_expr : expr 'OR' expr / expr 'AND' expr / 'NOT' expr /
            expr rel_op expr / expr eq_op expr .
int_expr :  expr add_op expr  / unary_op expr .

eq_op :     '=' / '!=' / '==' / '!==' .
rel_op :    '<' / '>'  / '<=' / '>=' .
add_op :    '+' / '-' .
unary_op :  '-' / '+' .

constant :  BOOL_CONST / STRING_CONST / INT_CONST / CHAR_CONST / '#' IDENTIFIER .
```

Static Semantics Let EX denote the set of all expressions. Let $CO \subseteq EX$ be the set of constant expressions. The type of a constant expression is defined as mapping $cotype : CO \rightarrow \mathcal{T}$ based on the alternative of production `constant` the respective constant is matched with:

$$co \mapsto \begin{cases} bool & \text{if constant is a } \texttt{BOOL\_CONST} \\ string & \text{if constant is a } \texttt{STRING\_CONST} \\ int & \text{if constant is a } \texttt{INT\_CONST} \\ char & \text{if constant is a } \texttt{CHAR\_CONST} \\ error & \text{if constant is matched with } \texttt{\#IDENTIFIER} \end{cases}$$

The type of expressions is then recursively defined by mapping $type : EX \rightarrow \mathcal{T}$ with:

$$ex \mapsto \begin{cases} bool & \text{if expression is a } \texttt{bool\_expr} \\ int & \text{if expression is an } \texttt{int\_expr} \\ type(e) & \text{if expression matches production } \texttt{'(' e ')'} \\ cotype(co) & \text{if expression matches constant co} \\ petype(pe) & \text{if expression matches path expression pe} \end{cases}$$

Let e be an expression. We then require the following obvious typing rules:

$$e \text{ matches} \begin{cases} e_1 \texttt{ 'AND' } e_2 \Rightarrow type(e_1) = type(e_2) = bool \\ e_1 \texttt{ 'OR' } e_2 \Rightarrow type(e_1) = type(e_2) = bool \\ \texttt{'NOT' } e_1 \Rightarrow type(e_1) = bool \\ e_1 \texttt{ eq\_op } e_2 \Rightarrow type(e_1) = type(e_2) = int \\ e_1 \texttt{ rel\_op } e_2 \Rightarrow type(e_1) = type(e_2) \end{cases} \tag{A.45}$$

Having finished the formal definition of expressions, we can now complete the definition of path expressions by defining the static semantics of actual parameter lists. Let, therefore, AP be the set of actual parameters. An actual parameter list is considered as a tuple of elements from AP, i.e. as an element of AP^*. Projection $[] : AP^* \times I\!N \rightarrow AP$ and length $|| : AP^* \rightarrow I\!N$ are defined as usual. We then associate parameter lists to steps of a path expressions by the partial mapping $act\_params : S \rightarrow AP^*$ with

$$s \mapsto \begin{cases} \text{element of } AP^* \text{ that matches } \texttt{opt\_parameter\_list} \text{ in } \texttt{step} \\ \text{undefined, if } \texttt{IDENTIFIER} \text{ is matched in production } \texttt{step} \end{cases}$$

Furthermore, we define the type of an actual parameter to be the type of the respective expression, i.e. $type : AP \rightarrow \mathcal{T}$ with $ap \mapsto type(e)$ where e is the expression ap is matched with.

We then require the actual parameter list to be conform to the respective method declaration. This means that the number of actual parameters must be equal to the number of formal parameters. We do not want to allow variable length parameter lists or parameters with default values like in C++ since we consider these as inherent sources of specification errors. Furthermore, the types of the actual parameters must be subtypes of the formal parameter types. This is the formal definition of the polymorphism rule for parameters. We hence require for all path expressions $pe \in PE$ and for all steps at positions $i \in \{2, \ldots, |pe|\}$:

$$act\_params(pe[i]) \neq undefined \Rightarrow \bigvee_{c \in \mathcal{C}} c = stype'(pe, i \Leftrightarrow 1) \tag{A.46}$$
$$\bigvee_{m \in M_c^*} \left(\begin{array}{c} name^*(m) = name(pe[i]) \wedge \\ |params_c(m)| = |act\_params(pe)| \wedge \\ \bigwedge_{j \in \{1,\ldots,|params_c(m)|\}} type(act\_params(pe)[j]) \leq type_c(params_c(m)[j]) \end{array} \right)$$

A.3.5 Methods

Context-free Syntax The method section of a class contains the body specifications of methods defined in the interface part of a class definition. Each method specification again includes the method head for the purpose of better readability. Then variables may be declared that can be used to store intermediate results locally in the method body. Finally, the method body is given as a list of statements. The precise syntax definition is as follows:

```
method_section  :   'METHODS' method_body_list 'END' 'METHODS' ';' .

method_body :       'METHOD' IDENTIFIER '(' opt_parameter_list ')'
                     opt_result_type ';'
                     opt_var_decl
                    'BEGIN' statement_list 'END' IDENTIFIER ';' .

parameter_list :    parameter
                  / parameter ';' parameter_list .
parameter :         IDENTIFIER ':' type_declaration .

result_type :       ':' type_declaration .

var_decl :          'VAR' var_decl_item_list .
var_decl_item :     IDENTIFIER ':' type_declaration ';' .

type_declaration :  IDENTIFIER
                  / 'LIST' 'OF' IDENTIFIER
                  / 'SET' 'OF' IDENTIFIER
                  / 'BAG' 'OF' IDENTIFIER
                  / 'DICTIONARY' 'OF' IDENTIFIER ';'
```

Static Semantics Let MB_c be the set of method bodies that are included in the specification of a class $c \in \mathcal{C}$. Each of them is associated with a name by $name : MB_c \rightarrow A^*$ with $m \mapsto$ string that matches `IDENTIFIER` in production `method_body`. We require a body to be included for each explicit or hidden method defined in the interface:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{m \in M_c} kind(m) \in \{EXP, HID\} \Rightarrow \bigvee_{mb \in MB_c} name(mb) = name(m) \tag{A.47}$$

Vice versa, we do not want to have method bodies that are not declared in the interface since due to Condition A.38 these method bodies could never be invoked:

$$\bigwedge_{mb \in MB_c} \bigvee_{m \in M_c} name(mb) = name(m) \wedge kind(m) \in \{EXP, HID\} \tag{A.48}$$

Together with the uniqueness condition on method names (A.25), we can, therefore, define a bijective mapping between method interfaces and bodies for each class $c \in \mathcal{C}$: $body_c : \{m \in M_c \mid kind(m) \in \{EXP, HID\}\} \rightarrow MB_c$ with $m \mapsto mb$ if $name_c(m) = spec\_name(mb)$.

We then require parameter lists and result types of methods to match their counterparts in the body specifications. Let, therefore, PAR be the set of parameters with $name : PAR \rightarrow A^*$ and $type : PAR \rightarrow \mathcal{T}$ defined as usual. PAR^* represents parameter lists with mappings $||$ and $[]$ as usual. $spec\_params_c : MB_c \rightarrow PAR^*$ associates a tuple of parameters with a method body and $spec\_type : MB_c \rightarrow \mathcal{T}$ associates a type with the method body in the same way as defined for the class interface. We then require for all classes $c \in \mathcal{C}$ and all methods $m \in M_c$:

$$\begin{aligned}
&|params_c(m)| = |spec\_params_c(body(m))| \wedge \\
&\bigwedge\nolimits_{i \in \{1 \ldots |params_c(m)|\}} \begin{pmatrix} name(params_c(m)[i]) = name(spec\_params_c(body(m))[i]) \\ type(params_c(m)[i]) = type(spec\_params_c(body(m))[i]) \end{pmatrix} \wedge \\
&mtype_c(m) = spec\_type(body(m))
\end{aligned} \tag{A.49}$$

We now have to define the declarations that are valid in statements of a method bodies statement list. Let, therefore, VAR be the set of variable declarations. Mappings $name : VAR \rightarrow A^*$ and $type : VAR \rightarrow \mathcal{T}$ define name and type of a variable analogously to parameters. We then associate a set of variables to each method body by mapping $vars : MB_c \rightarrow \mathcal{P}(VAR)$ with $mb \mapsto \{v \in VAR \mid$ v matches symbol `var_decl_item` in production `var_decl_item_list` of $mb\}$. Likewise, we associate parameters with methods by $params : MB_c \rightarrow \mathcal{P}(PAR)$ with $mb \mapsto \{p \in PAR \mid \vee_{i \in \{1, \ldots |spec\_params_c(mb)|\}} p = spec\_params_c(mb)[i]\}$. We require types of each variable and parameter to be visible, i.e. to be imported:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{m \in M_c} \bigwedge_{v \in vars_c(body(m))} type(v) \in T_c^s \tag{A.50}$$

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{m \in M_c} \bigwedge_{p \in spec\_params_c(body(m))} type(p) \in T_c^s \tag{A.51}$$

The scope for method bodies is then defined as follows. Let $st \in STMT$ be a statement that is included in the statement list of a method m. The set of declarations valid for st is then defined as $scope(st) := \{self\} \cup E_c^* \cup params(body(m)) \cup vars(body(m))$. We have to require that local variables and parameters are unique within their scope:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{m \in M_c} \bigwedge_{v_1, v_2 \in vars(body(m)) \cup params(body(m)) \cup E_c^*} name(v_1) = name(v_2) \Rightarrow v_1 = v_2 \tag{A.52}$$

We will have to define a static semantic condition that assures that return statements are only contained in statements that belong to method bodies. To prepare this, we have to associate statements to method bodies. Let, therefore, $stmts : MB_c \rightarrow \mathcal{P}(STMT)$ with $mb \mapsto \{s \in STMT \mid$ s belongs to `statement_list` of $mb\}$.

Dynamic Semantics The parameter passing mechanism applied upon method invocation is call by value similar as in Eiffel or O_2C. Local variables are initialised with their default values and then the statement list is executed sequentially.

A.3.6 Statements

Statements occur in various places in GTSL class specifications such as bodies of explicit methods, interactions or semantic rules. Assignment statements are used to assign values that are identified by e.g. path expressions to properties such as a link in order to establish a semantic relationship. Method calls are used to invoke operations that modify increments of other classes or traverse along paths. Loops and iterations are required to repeatedly execute a list of statements. In a loop statement, the tool builder can specify an explicit termination condition. Iteration statements are introduced in order to perform a sequence of statements for all elements of a multi-valued property. Branches provide a means to specify alternative statement lists depending on a particular condition. Finally, return statements terminate a method execution and determine its returned result.

Context-free syntax

```
statement_list : statement
               / statement ';' statement_list .

statement :      assignment / method_call / loop / iteration
               / branch / case / return / empty .

assignment :     IDENTIFIER ':=' initialisation ';' .
initialisation : creation / expression .
creation :       'NEW' IDENTIFIER '(' opt_actual_parameter_list ')' .

method_call :    path_expr .

loop  :          'WHILE' expr 'DO' statement_list 'ENDDO'  .
iteration  :     'FOREACH' IDENTIFIER ':' IDENTIFIER 'IN' expr 'DO'
                  statement_list
                 'ENDDO'  .

branch :         'IF' expr 'THEN'
                   statement_list
                   opt_else_part
                 'ENDIF'
else_part :      'ELSE' statement_list   .

return :         'RETURN' '(' expr ')' .
empty :          .
```

Static Semantics We now define the static semantics of the various kinds of statements in the same order as they are given in the grammar.

Let ASS be the set of assignment statements. Furthermore, let $assname : ASS \rightarrow A^*$ with $ass \mapsto$ lexical value of the string that maches with symbol `IDENTIFIER` in production `assignment`, return the identifier of the assignment's left-hand-side. Moreover, let $assexp : ASS \rightarrow EXP$ with $ass \mapsto$ expression that matches symbol `expr` in production `assignment`, return the expression of a the right-hand-side. We then require that the identifier is defined within the scope of the statement. It must not be defined as an implicit link. Implicit links are read-only, since they are created as soon as an assignment to an explicit link is performed and they are deleted as soon as an existing link is overwritten. We then require for each assignment $ass \in ASS$:

$$\bigvee_{e \in scope(ass)} name(e) = assname(ass) \wedge e \notin \{l \in E_c \mid kind(l) = IL\} \tag{A.53}$$

For the creation of a new attribute or increment, we have to require that the identifier following the keyword `NEW` matches with some class $c \in \mathcal{C}$. Class c must not be abstract for the following reason. The dynamic type of the newly created instance will be C. For late binding of methods, resolution will start at c. If c was abstract, it could have deferred methods whose bodies are not defined. Then late binding would be unsafe, since invoking a deferred method could not be resolved. To formalise these concerns, let CRE be the set of creation directives and $crename : CRE \rightarrow A^*$ associate a name with a creation directive by $cre \mapsto$ string that matches symbol `IDENTIFIER` in production `creation`. We then require

$$\bigwedge_{cre \in CRE} \bigvee_{c \in \mathcal{C}} crename(cre) = name(c) \wedge kind(int(c)) \neq ABI \tag{A.54}$$

Moreover, we require that the polymorphism rule holds for assignments, i.e. the type of the expression or creation must be a subtype of the static type of the left-hand-side of the assignment. Therefore, we define a mapping $asstype : ASS \rightarrow \mathcal{T}$ with

$$ass \mapsto \begin{cases} type(e), & \text{if } \texttt{initialisation} \text{ matches expression } e \\ c, & \text{if } \texttt{initialisation} \text{ matches creation } cre \wedge crename(cre) = name(c) \end{cases}$$

that associates a type with an assignment. We then require the polymorphism rule to hold for each assignment $ass \in ASS$:

$$\bigvee_{e \in scope(ass)} name(e) = assname(ass) \wedge asstype(ass) \leq type(e) \tag{A.55}$$

Let $LOOP$ be the set of loop statements and $loopexpr : LOOP \rightarrow EXP$ with $l \mapsto$ expression that matches symbol `expr` in production `loop`. We then have to require that the expression's type is `BOOLEAN`:

$$\bigwedge_{l \in LOOP} type(loopexpr(l)) = bool \tag{A.56}$$

The static semantic conditions for iterations are more complicated. First of all the type of the expressions must be a multi-valued type. The name of the cursor defined after the `FOREACH` keyword has to be unique and its type must be a super type of the base type of the multi-valued type. Then elements of the multi-valued expression can iteratively be assigned to the cursor according to the polymorphism rule. In addition, the cursor is a declared property for all statements within the iteration's statement list, i.e it may be used within expressions. To formalise this, let IT be the set of iteration statements and CU be the set of cursor declarations. Mapping $itexpr : IT \rightarrow EXP$ with $it \mapsto$ expression that matches symbol `expr` in production `iteration` associates an expression with the iteration. Moreover, $cursor : IT \rightarrow CU$ associates a cursor to each iteration. The cursor name is defined by mapping $cuname : CU \rightarrow A^*$ with

$cu \mapsto$ string that matches the first `IDENTIFIER` symbol in production `iteration`. The type of a cursor is defined by mapping $cutype : CU \rightarrow \mathcal{T}$ with

$$cu \mapsto \begin{cases} bool, & \text{if the second identifier in } \texttt{iteration} \text{ matches } \texttt{'BOOLEAN'} \\ int, & \text{if the second identifier in } \texttt{iteration} \text{ matches } \texttt{'INTEGER'} \\ char, & \text{if the second identifier in } \texttt{iteration} \text{ matches } \texttt{'CHAR'} \\ string, & \text{if the second identifier in } \texttt{iteration} \text{ matches } \texttt{'STRING'} \\ c, & \text{if the second identifier in } \texttt{iteration} \text{ matches } name(c) \end{cases}$$

We then require that elements of the multi-valued property identified by the expression are subtypes of the cursor's type:

$$\bigwedge_{it \in IT} \bigvee_{t \in \mathcal{T}} \left(\begin{array}{c} type(itexpr(it)) \in \{list(t), set(t), bag(t), dictionary(t)\} \wedge \\ t \leq cutype(cursor(it)) \end{array} \right) \tag{A.57}$$

The cursor of an iteration extends the scope of the iteration's statement list. More precisely, the scope of any statement s contained in the statement list of an iteration it is $scope(s) := scope(it) \uplus \{cursor(it)\}$. Mappings $name$ and $type$ are defined as $name : CU \rightarrow A^*$ with $cu \mapsto cuname(cursor(it))$ and $type : CU \rightarrow \mathcal{T}$ with $cu \mapsto cutype(cursor(it))$

We then require that the cursor name is unique within the scope of statements contained in the iteration's statement list:

$$\bigwedge_{e_1, e_2 \in scope(it) \uplus \{cursor(it)\}} name(e_1) = name(e_2) \Rightarrow e_1 = e_2 \tag{A.58}$$

Let BRA be the set of all branch statements. Mapping $braexp : BRA \rightarrow EXP$ with $bra \mapsto$ expression that matches symbol $expr$ in production `branch` attaches an expression to a branch. Obviously, the branch expression's type must be `BOOLEAN`:

$$\bigwedge_{b \in BRA} type(braexp(b)) = bool \tag{A.59}$$

As interactions and semantic rules cannot have result types, this implies that return statements must not occur within interactions or semantic rules. This could have been defined in the context-free grammar, but would have complicated the grammar significantly. Therefore, we require the following static semantic condition:

$$\bigwedge_{ret \in RET} \bigvee_{c \in \mathcal{C}} \bigvee_{m \in M_c} ret \in stmts(body(m)) \tag{A.60}$$

The type of a return statement has to be compatible to the result type as defined for the method whose body contains the return statement. Compatibility here again means that according to the polymorphism rule, the actual type of the returned expression is a subtype of the formal result type. To define this formally, let $type : RET \rightarrow \mathcal{T}$ with $r \mapsto type(e)$ if e matches symbol `expression` in production `return` be the type of the returned expression. Furthermore, let $incl\_meth : RET \rightarrow M_c$ with $r \mapsto m$ if $r \in stmts(body(m))$. We then require

$$\bigwedge_{ret \in RET} type(ret) \leq type(incl\_meth(ret)) \tag{A.61}$$

Dynamic Semantics The semantics of assignments to local variables, quantors and cursors is similar to Eiffel. The effect of the assignment is lost as soon as the scope of the declaration of these assignments is left. The semantics of assignments to attributes and abstract syntax children, differs from Eiffel in that they are persistent. That is, as soon as the interaction, which has caused an assignment has successfully been completed with a commit, the effect of the assignment survives termination of the tool (irrespective whether the tool was terminated properly or by a hardware or software failure). To that respect the semantics of an assignment has the same effect as an assignment to an instance variable in O_2C. Assignments to links of semantic relationships even differ from O_2C. The expression that is assigned to an explicit link is a subclass of an increment class (A.16). This class has an implicit link whose type is multi-valued (A.17). Its base type is a super type of the class of the assignment's right-hand side (A.55, A.17). If the explicit link has a value, the current increment is removed from the implicit link of the class identified by that explicit value. Then the assignment to the explicit link is performed. After that the current increment is inserted into the implicit link of the new value of the explicit link. Thus it is assured that the contents of implicit links always correspond to explicit links and path expression interpretation can always traverse relationships in both directions using the respective explicit and implicit links.

The semantics of a method call is to start interpretation of the path expression that represents the method call. Path expressions have been defined on Page 284.

The dynamic semantics of loops is equal to `WHILE` statements in most imperative and object-oriented languages: The statement list is executed until the expression of type `BOOLEAN` (A.56) evaluates to `FALSE`.

If the multi-valued property identified by the expression of an iteration is empty, execution will continue with the statement after the iteration. Otherwise, the cursor is assigned to the first element of the multi-valued property. This is possible since the dynamic types of elements in the collection are subtypes of the cursor (A.57). If the property is a list, the cursor refers to the first list element. Otherwise, an element is chosen non-deterministically. Then the statement list is executed. After that, the next element is assigned to the cursor. Again the order is respected for lists and a non-deterministic choice is taken for other multi-valued properties. If there is no next element, execution is continued after the iteration statement, otherwise the statement list is executed for the current value of the cursor again.

The dynamic semantics of branches is equal to `IF` statements in most imperative and object-oriented languages: The statement list that follows the `THEN` keyword will be executed, if the boolean expression (A.59) evaluates to `TRUE`. If it is `FALSE` and there is an `ELSE` keyword, the statement list following this keyword is executed. Otherwise the statement that follows the branch statement will be executed.

The return statement terminates execution of a method body. Its argument expression determines the return value of the method.

A.3.7 Commands

Context-free Syntax Each interaction includes a mandatory selection clause that is introduced with the keywords `SELECTED IS`. This clause specifies the increment that must have been selected at the user interface in order to consider the interaction for inclusion into the menu of applicable commands. In principle, the clause would not be required. Then, however, the tool had to check all interactions of all increment classes whenever a context-sensitive menu is constructed. Including the clause into the language enables the generator to significantly optimise the menu construction procedure. Moreover, the selection clause as defined here enforces structuring of interactions according to the abstract syntax definition. The clause will allow specifying the class itself, a child of its abstract syntax, or an element of a multi-valued abstract syntax child. Hence, a command that applies to an increment of class `A` can only be defined as an interaction of class `A` itself or the father classes that declare an abstract syntax child of class `A`.

The syntax of the interaction section is then defined as follows:

```
interaction_section :   'INTERACTIONS' interaction_list 'END' 'INTERACTIONS' ';' .

interaction :           'INTERACTION' IDENTIFIER ';'
                          'NAME' STRING_CONST
                          'SELECTED' 'IS' selection
                          'ON'  expression
                          opt_variable_declaration
                          'BEGIN'
                           interaction_statement_list
                        'END' IDENTIFIER ';' .

selection :             IDENTIFIER / 'ELEMENT' 'IN' IDENTIFIER .
interaction_statement : statement / COMMIT / ABORT
```

Static Semantics Let $IACT_c$ be the set of interactions defined for a class $c \in \mathcal{C}$. Interaction names are captured by mapping $name_c : IACT_c \rightarrow A^*$ with $i \mapsto$ string that matches the first occurrence of symbol `IDENTIFIER` in production `interaction`. Interaction names must be unique so as to use the names for identification purposes. Therefore, we require:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{i_1, i_2 \in IACT_c} name_c(i_1) = name_c(i_2) \Rightarrow i_1 = i_2 \tag{A.62}$$

Interactions are inherited from super classes. Let, therefore, $IACT_c^*$ be the set of inherited interactions of a class with:

$$IACT_c^* := IACT_c \cup \bigcup_{c' \in pred(c)} IACT_{c'}^*$$

Mapping $name$ is continued canonically in $name_c^* : IACT_c^* \rightarrow A^*$. An interaction i defined in class c is said to be redefined if there is another class $c' \in succ^*(c)$ with another interaction i' and $name_c(i) = name_{c'}^*(i')$. Note, that we do not have to demand additional correctness criteria regarding redefinition like the covariant redefinition rule, since interactions are neither parametrised nor return a result.

Only increments can be selected at the user interface of a tool. Therefore, we must exclude that the selection clause identifies attributes or links of semantic relationships. Let, therefore, $sel_c : IACT_c \rightarrow A^*$, with $i \mapsto$ string that matches symbol `IDENTIFIER` in production `selection`. If the first alternative is chosen in that production, we require that sel denotes an abstract syntax child or the class itself:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{i \in IACT_c} \left(\begin{array}{c} sel(i) =' SELF' \vee \\ \bigvee_{e \in E_c^*} name^*(e) = sel(i) \wedge kind^*(e) = AS \end{array} \right) \tag{A.63}$$

Otherwise, the selection must be an element of a multi-valued abstract syntax child:

$$\bigwedge_{c\in\mathcal{C}} \bigwedge_{i\in IACT_c} \bigvee_{e\in E_c^*} \bigvee_{t\in\mathcal{T}} \begin{pmatrix} sel(i) = name^*(e) \wedge \\ kind^*(e) = AS \wedge \\ type^*(e) = list(t) \end{pmatrix} \tag{A.64}$$

obviously, the type of the expression that follows keyword `ON` must be *bool*. Let, therefore, $inttype_c : IACT_c \rightarrow \mathcal{T}$, with $int \mapsto type(e)$ if e matches symbol `expression` in production `interaction`, denote the type of the `ON` clause. We then require:

$$\bigwedge_{c\in\mathcal{C}} \bigwedge_{i\in IACT_c} inttype_c(i) = bool \tag{A.65}$$

We finally have to define the declarations that are valid in interaction statement lists. Obviously, all inherited properties and the local variables are valid. In addition, a tool builder might want to have access to the currently selected increment, in order to e.g. determine the list position of a new increment relative to the position of the current increment. Therefore, we introduce a pre-defined variable `CURSOR`. This variable is valid in interaction statement lists only. The type of the variable is inferred from the type given in the selection clause. For the formal definition of these concerns, let $vars_c : int_c \rightarrow \mathcal{P}(VAR)$ denote the set of local variables for an interaction of class $c \in \mathcal{C}$ with $i \mapsto \{v \in VAR \mid$ v matches symbol `var_decl_item_list of interaction` i$\}$. Moreover, $seltype_c : IACT_c \rightarrow \mathcal{T}$ identifies the type of the selection clause through:

$$i \mapsto \begin{cases} c, & if\ sel(i) =' SELF' \\ type^*(e), & if\ sel(i) = name^*(e) \wedge type^*(e) \in \mathcal{C} \\ t & if\ sel(i) = name^*(e) \wedge type^*(e) = list(t) \end{cases}$$

We then define that VAR includes for each class $c \in \mathcal{C}$ and each interaction $i \in IACT_c$ a variable that is identified by mapping $cursor_c : IACT_c \rightarrow VAR$. We define the following properties for these variables: $name(cursor_c(i)) :=$ `"CURSOR"` and $type(cursor_c(i)) := seltype_c(i)$. The declarations that are valid for a statement s included in the statement list of an interaction $i \in IACT_c$ of class $c \in \mathcal{C}$ are then $scope(s) := E_c^* \cup vars_c(i) \cup \{cursor(i)\}$. Again we have to require that local variable names are unique and do not interfere with property names:

$$\bigwedge_{c\in\mathcal{C}} \bigwedge_{i\in IACT_c} \bigwedge_{v_1,v_2\in E_c^*\cup vars_c(i)\cup\{cursor(i)\}} name(v_1) = name(v_2) \Rightarrow v_1 = v_2. \tag{A.66}$$

We have to require that local variable's types are included in the set of visible types, i.e. have been properly imported:

$$\bigwedge_{c\in\mathcal{C}} \bigwedge_{i\in IACT_c} \bigwedge_{v\in vars_c(i)} type(v) \in T_c^s \tag{A.67}$$

Dynamic Semantics The user of a tool specified in GTSL can select increments by clicking into a region of the screen with the selection mouse button (usually the left). Selected becomes the increment to which the region belongs according to the unparsing scheme. The user can then demand a menu of applicable commands by pushing the menu mouse button. During the successive menu construction phase, the tool will consider all interactions whose selection clauses match with the currently selected increment. For all (inherited) interactions of the current increment the clause matches if it equals `SELF`. It matches also in (inherited) interactions of the father increment if the selection clause demands the current increment as child increment or as element of a child increment. From these interactions, the tool selects the subset of those interactions whose `ON` expressions evaluate to `TRUE`. Then the external names of interactions are listed in a pop-up menu.

After the user menu has been created, the user can choose a command. Then an ACID transaction is started and the statement list included in the interaction specification is executed in isolation to interactions executed by concurrently running tools. If the execution of the statement list is completed, the ACID transaction is completed with an implicit commit. The tool may perform an implicit abort operation in case the interaction is involved in a deadlock. The interaction may be explicitly terminated with a `COMMIT` or `ABORT` statement. Explicit and implicit commits have the same effect: They achieve that the effect of the command execution is persistent and durable. Implicit and explicit aborts terminate the current interaction and undo all changes that have been performed since the interaction started.

A.3.8 Semantic Rules

Context-free Syntax The context-free syntax of semantic rules is as follows:

```
semantic_rule_section : 'SEMANTIC' 'RULES'
                          semantic_rule_list
                        'END' 'SEMANTIC' 'RULES' ';'  .

semantic_rule :         'ON' sr_condition opt_var_decl action ';' .

sr_condition :          predicate / '(' compound_expr ')' .
compound_expr :         changed_expr / changed_expr 'OR' compound_expr
predicate :             changed_expr / collapses_expr / exists_expr .
changed_expr :          'CHANGED' '(' path_expr ')' .
collapses_expr :        'DELETED' '(' path_expr ')' .
exists_expr :           'EXISTS' '(' IDENTIFIER ':' IDENTIFIER 'IN' path_expr ')' ':'
                         sr_condition .

action :                'ACTION' statement_list 'END' 'ACTION' .
```

Static Semantics We have to exclude method calls from path expressions contained in semantic rule conditions to avoid side-effects during rule evaluation. For the formal definition of this condition, let $PPE \subseteq PE$ be the set of path expressions that are used within conditions of semantic rules. We then require steps to be properties rather than methods:

$$\bigwedge_{pe \in PPE} \bigwedge_{i \in \{2,\ldots,|pe|\}} \bigvee_{e \in E^*_{stype'(pe,i-1)}} name^*(e) = name(pe[i]). \tag{A.68}$$

Dynamic Semantics Semantic rules are used to determine attribute values and instantiate semantic relationships depending on other increment's attributes and relationships. Hence semantic rules have to declare dependencies between attributes and relationships of different classes and possibly even different documents. The condition that follows after the `ON` keyword serves this purpose. It is called *pre-condition* of the rule. Three different kinds of primitives may be used to define these dependencies: `CHANGED`, `DELETED` and `EXISTS` predicates. A `CHANGED` predicate evaluates to true, iff the attribute or semantic relationship specified by the given path expression has been created, has become reachable or was changed otherwise. The `DELETE` predicate becomes true, iff the attribute or semantic relationship that is identified by the path expression is about to be no longer be reachable by the path expression. This could be because the increment the attribute or relationship belongs to is deleted or because the path no longer exists since an intermediate step was deleted. `EXISTS` predicates may be used to specify dependencies between the semantic rule and elements of a multi-valued semantic relationship. If the pre-condition becomes true, the statement list that belongs to the will be executed before any attribute or relationship modified in the statement list is accessed from elsewhere.

The first `IDENTIFIER` symbol in production `exists_expr` denotes the name of a variable and the second `IDENTIFIER` symbol denotes the variable's type. The type of the path expression must be multi-valued, otherwise it would useless to apply an `EXISTS` predicate. As an element of the multi-valued collection will during run-time be assigned to the variable, the variable's type must be a super type of the base type of the collection. Otherwise the polymorphism rule would be violated. Moreover, the type must be visible. To formalise these concerns, let $EXIST$ be the set of exist predicates and $QVAR$ be the set of variable declarations in these predicates. Let $vars : EXIST \rightarrow QVAR$ be the association of variables to exist predicates according to the matches in production `exists_expr`. Mappings $name : QVAR \rightarrow A^*$ and $type : QVAR \rightarrow \mathcal{T}$ are defined as usual. Let moreover $ex\_path : EXIST \rightarrow PPE$ denote the association between exists predicates and path expressions according to production `exists_expr`. We then require:

$$\bigwedge_{ex \in EXIST} \bigvee_{c \in \mathcal{C}} \left(\begin{array}{c} c \in T_c^s \wedge \\ kind(int(c)) \in \{NTI, ABI, TI\} \wedge \\ c \leq type(vars(ex)) \wedge \\ petype(ex\_path(ex)) \in \{list(c), set(c), bag(c), dictionary(c)\} \end{array} \right) \tag{A.69}$$

Let SR_c be the set of semantic rules in a class $c \in \mathcal{C}$. We then assume that mapping $ex\_pre_c : SR_c \rightarrow \mathcal{P}(EXIST)$ defines the set of `EXISTS` predicates contained in the condition of a semantic rule according to productions `sr_condition` and `exists_expr`. Furthermore, let $vars_c : SR_c \rightarrow \mathcal{P}(VAR)$ return the set of variables that are declared in the optional variable list of a semantic rule similarly mapping $vars_c$ on interactions. Then the valid declarations for a statement st contained in the statement list of a semantic rule $sr \in SR_c$ in class $c \in \mathcal{C}$ are the local variables and the variables declared in the `EXISTS` predicates:

$$scope(st) := E_c^* \cup vars_c(sr) \cup \bigcup_{ex \in ex\_pre_c(sr)} vars(ex)$$

Again, we have to require uniqueness of variable names so as to use $name$ for variable identification purposes:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{sr \in SR_c} \bigwedge_{v_1, v_2 \in E_c^* \cup vars_c(sr) \cup \bigcup_{ex \in ex\_pre_c(sr)} vars(ex)} name(v_1) = name(v_2) \Rightarrow v_1 = v_2 \tag{A.70}$$

Similar to local variables of interactions, the type of a local variable of a semantic rule has to be visible. Condition A.67 is, therefore, applied accordingly:

$$\bigwedge_{c \in \mathcal{C}} \bigwedge_{i \in SR_c} \bigwedge_{v \in vars_c(i)} type(v) \in T_c^s \tag{A.71}$$

Variables that are declared in `EXISTS` predicates may be used in inner predicates as well. Let, therefore, $ex\_pre_c : PRE \rightarrow \mathcal{P}(EXIST)$ reflect the containment hierarchy of exists predicates, i.e. a predicate is mapped to the set of `EXISTS` predicates that it is contained in according to production `exists_predicate`. For a path expression pe, contained in a predicate $p \in PRE := EXIST \uplus CHGD \uplus DEL$, the set of valid declarations is then:

$$scope(pe) := E_c^* \cup \bigcup_{ex \in ex\_pre_c(p)} vars(ex)$$

Note, that we do not have to require uniqueness of variable names declared in `EXISTS` predicates, since uniqueness is covered by (A.70) already.

Rule based specifications tend to become unstructured and hard to comprehend. In order to help a tool builder defining the static semantics and inter-document consistency constraints in a well-structured and comprehensible way, we enforce some restrictions on method invocations in statements of semantic rules. These restrictions will force a tool builder to specify semantic properties in those classes where they belong. Together with the visibility rules defined for properties, this will lead to well-structured specifications. To define these conditions we need to define an additional concept for methods. A method $m \in M_c$ defined in a class $c \in \mathcal{C}$ is said to be *modifying*, iff

$$\begin{aligned} &\bigvee_{a \in ASS \cap stmts(body(m))} \bigvee_{e \in E_c^*} assname(a) = name^*(e) \vee \\ &\quad \bigvee_{p \in PE \cap stmts(body(m))} \bigvee_{i \in \{2,\ldots,|p|\}} \bigvee_{m \in M^*_{stype'(p,i-1)}} \quad name(p[i]) = name^*(m) \wedge \\ &\qquad \text{m is modifying} \end{aligned}$$

We then require that modifying methods in semantic rules are invoked only for properties of the class the semantic rule belongs to. Hence modifications during static semantics or inter-document consistency constraint checks are specified in the semantic rule of the class that declares the changed property. We, therefore, require for all classes $c \in \mathcal{C}$ and all semantic rules $sr \in SR_c$:

$$\bigwedge_{p \in PE \cap stmts(sr)} \bigwedge_{i \in \{2,\ldots,|p|\}} \bigvee_{m \in M^*_{stype'(p,i-1)}} name(p[i]) = name^*(m) \wedge \text{ m is modifying} \Rightarrow \bigvee_{e \in E_c^*} name(p[i \Leftrightarrow 1]) = name^*(e) \qquad (A.72)$$

Moreover, semantic rules are meant to change attribute values and create or delete semantic relationships, which are used to store semantic properties. We do not want that rules are abused for performing arbitrary increment modifications. Increment methods should be invoked from interactions for these purposes. We, therefore, further restrict method invocations in semantic rules. As semantic relationships are created by assigning a value to a link, the only remaining reason for invoking a method is to access or modify an attribute. This is done using methods defined in non-syntactic classes. We, therefore, exclude any other method calls and require for each class $c \in C$ and each semantic rule $sr \in SR_c$:

$$\bigwedge_{p \in PE \cap stmts(sr)} \bigwedge_{i \in \{2,\ldots,|p|\}} \bigvee_{m \in M^*_{stype'(p,i-1)}} name(p[i]) = name^*(m) \Rightarrow stype'(p, i \Leftrightarrow 1) \in NSC \qquad (A.73)$$

Appendix B

The Library of Pre-defined Classes

B.1 Increment

Class `Increment` is the most general abstract increment class. Here, we define the properties that every increment has. First of all, any increment must have a reference to its abstract syntax father. Otherwise, navigations could only be performed from parents to children, but not in the opposite direction. Therefore the attribute `father` maintains this reference. The value of this attribute is determined by the parameter that is passed to the `init` method of `Increment`. Class `Increment` also defines a boolean attribute `expanded` that is `FALSE`, iff the increment is still a place holder. The attribute is modified by implicit methods such as `scan`, `parse` or `expand`, which replace a place holder with child increments and `collapse` that transforms an increment into a place holder. Moreover, class `Increment` defines an attribute `Errors` of type `SET OF Errors`. It stores a set of error descriptors that are inserted to or deleted from the set by semantic rules defined in subclasses. If the attribute `Errors` of an increment is not empty, the textual representation of the increment will be marked as erroneous at the user interface (by underlining). Then the user may obtain a textual representation of the error descriptors in a separate window. `Increment` exports a number of methods, such as `includes_error`, `append_error` and `clear_error` in order to query and modify the set. Finally, class `Increment` defines a path expression to the root increment of the document it is contained in. This path expression is declared as method `get_doc`. It simplifies access to information that is shared by arbitrary increments, because this information is usually stored at the root increment.

```
ABSTRACT INCREMENT INTERFACE Increment;

  IMPORT INTERFACE
    IMPORT Document;
    IMPORT Error;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      father: Increment;            // reference to parent of abstract syntax
      expanded: BOOLEAN;            // FALSE if increment is a place holder
      Errors: SET OF Error;         // set of errors that are found at the increment
      HIDDEN myDocument: Document;  // reference to the root node of document
    END ATTRIBUTES;
    METHODS
      DEFERRED METHOD collapse:STRING;// any increment class will have
      DEFERRED METHOD unparse:STRING; // these implicit methods

      METHOD is_phylum():BOOLEAN;     // returns true if increment is a place holder
      METHOD get_father():Increment;  // returns reference to the father
      METHOD get_doc():Document;      // returns referende to the root node of document

      // Error management in increment
      METHOD has_error():BOOLEAN;
      METHOD get_errors():SET OF STRING;
      METHOD includes_error(errType : ERROR_TYPE) : BOOLEAN;
      METHOD append_error(errType : ERROR_TYPE) : BOOLEAN;
      METHOD clear_error(errType : ERROR_TYPE) : BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE Increment.
```

B.2 Document

Class `Document` is the common superclass for all increments that represent root increments of documents (w.r.t the abstract syntax). As such, it defines a set of attributes for data that all increments of a document have in common. Then this data need not be stored in each separate increment of a document. One of these attributes is `definedDocuments` that is a reference to the common root of all documents. This common root is an instance of class `DocumentPool`. The reference is needed, e.g. to see whether some other document exists. A further attribute, `Owner`, is a string that identifies the owner of the document. Methods `GetOwner` and `ChangeOwner` are defined that are used to query and set this attribute. Furthermore, the attribute `LastModificationDate` stores the date and time of the last document modification. The attribute is implicitly updated by a tool, whenever an increment interaction is successfully completed with a commit. A method `GetDate` is provided that is used to return the last modification date of a document.

```
ABSTRACT INCREMENT INTERFACE Document;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT DocumentPool;
    IMPORT Date;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      definedDocuments: DocumentPool;
      HIDDEN Owner: STRING;
      HIDDEN LastModificationDate : Date;
    END ATTRIBUTES;

    SEMANTIC RELATIONSHIP
      docPool: DocumentPool;
    END SEMANTIC RELATIONSHIP;

    METHODS
```

```
      METHOD init(name:STRING);
      METHOD ChangeOwner(new_owner:STRING);
      METHOD GetOwner():STRING;
      METHOD GetDate:Date;
      METHOD Touch();
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE Document.
```

B.3 DocumentVersion

A subclass of `Document` is class `DocumentVersion`. It is the common super class for all root increments of documents that are versionable. As such it maintains attributes for managing the version history graph of a document. It then contains methods in order to set a version of a document as a default version (`SetDefaultVersion`), to select a particular version (`SelectVersion`), to freeze versions in order to declare them as stable (`FreezeVersion`), to derive a version from the selected version (`DeriveVersion`), to navigate through the version history graph (`GetParents`, `GetChildren`) and to delete a particular version from the graph (`DeleteVersion`). These methods are then used for declaring version management commands in the class specification of `DocumentVersion`. They are therefore hidden. By declaring the root increment class of a tool configuration as a subclass of `DocumentVersion`, a tool is then automatically equipped with all required commands for version management.

`DocumentVersion` has an instance variable `AllUsingIncrements` of type `SET OF UsingIncrement` that maintains references to all instances of subclasses of `UsingIncrement` contained in the document. Before displaying a document at the user-interface, the editor kernel consults this set and selects the document versions identified by the attributes `UsedDoc` and `UsedVer` of the set elements. This assures that all successive commands are performed in the context of the right configuration.

```
ABSTRACT INCREMENT INTERFACE DocumentVersion;

  INHERIT Document;

  IMPORT INTERFACE
    IMPORT DocumentTable;
    IMPORT VersionVectorTable;
    IMPORT UsingIncrement;
    IMPORT UseableIncrement;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      HIDDEN RootVersionName            : STRING;
      HIDDEN Stable                     : BOOLEAN;
      HIDDEN NameOfCurrentVersion       : STRING;
      HIDDEN DefaultVersionName         : STRING;
      HIDDEN AllVersions                : SET OF STRING;
      HIDDEN OutgoingConsistentVersions : VersionVectorTable;
      HIDDEN IncomingConsistentVersions : VersionVectorTable;
      HIDDEN AllUsingIncrements         : SET OF UsingIncrement;
      HIDDEN AllUseableIncrements       : SET OF UseableIncrement;
    END ATTRIBUTES;
    METHODS
      METHOD Init(f:Increment; DocName : STRING; InitialVersionName : STRING);

      // Methods to modify derivation graph
      HIDDEN METHOD DeleteVersion(name : STRING) : BOOLEAN;
      HIDDEN METHOD SetDefaultVersion(name : STRING);
      HIDDEN METHOD SetDefaultVersionName(name : STRING);
      HIDDEN METHOD SelectVersion(name : STRING);
      HIDDEN METHOD SelectDefaultVersion();
      HIDDEN METHOD DeriveVersion(name : STRING);
      HIDDEN METHOD FreezeVersion();

      // methods for enquiries on derivation graph
      METHOD GetChildren(name : STRING) : SET OF STRING;
      METHOD GetParents(name : STRING) : SET OF STRING;
      METHOD GetRootVersionName() : STRING;
      METHOD GetVersionName() : STRING;
      METHOD GetDefaultVersionName() : STRING;
      METHOD AllVersions() : SET OF STRING;
      METHOD IsVersionStable(name : STRING):BOOLEAN;
      METHOD IsStable():BOOLEAN;
      METHOD GetChildrenClosure(name : STRING): SET OF STRING;
      METHOD GetParentsClosure(name : STRING): SET OF STRING;
      METHOD AreOnDifferentPaths(v1 : STRING; v2 : STRING) : BOOLEAN;
      METHOD RealizeNewVersion(v:Version; newName : STRING);

      // methods for enquiries on current configuration
      METHOD IsReferencingDifferentVersions() : BOOLEAN;
      METHOD GetUsedVer(docName:STRING) : STRING;
      METHOD GetUsingVers(docName:STRING) : SET OF STRING;
      METHOD UsedDocs():SET OF STRING;
      METHOD UsingDocs():SET OF STRING;
      METHOD UsedDocClosure():SET OF STRING;
      METHOD UsingDocClosure():SET OF STRING;
      METHOD IsCyclic () : BOOLEAN;
      METHOD IsCompatibleConfiguration () : BOOLEAN;

      // methods for defining inter-document relationships
      // will be used by UseableIncrement and UsingIncrement
      METHOD RealizeOutgoingUsingIncrement(inc : UsingIncrement);
      METHOD UnRealizeOutgoingUsingIncrement(inc : UsingIncrement);
      METHOD RealizeUseableIncrement(i:UseableIncrement);
      METHOD UnRealizeUseableIncrement(i:UseableIncrement);

      // methods for configuration management
      // will be used by CM tool commands
      HIDDEN METHOD SelectReferencedVersions();
      HIDDEN METHOD ChangeVersionDependency(aDoc:STRING; aVer:STRING);
      HIDDEN METHOD IsVersionUsed():BOOLEAN;
      HIDDEN METHOD EstablishConsistencyRelation();
      HIDDEN METHOD RealizeOutgoingConsVer(doc:STRING; VER:STRING);
      HIDDEN METHOD RealizeIncomingConsVer(doc:STRING; VER:STRING);
      HIDDEN METHOD OutgoingConsistentDocs() : SET OF STRING;
      HIDDEN METHOD IncomingConsistentDocs() : SET OF STRING;
      HIDDEN METHOD OutgoingConsistentVers(doc:STRING): SET OF STRING;
      HIDDEN METHOD IncomingConsistentVers(doc:STRING) : SET OF STRING;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE DocumentVersion.
```

B.4 OptionalIncrement

Class `OptionalIncrement` serves as a super class for all increment classes that trace back to optional productions of the syntax view. It defines a hidden attribute `removed` of type `BOOLEAN` that is `TRUE`, iff the increment is associated with the empty string. Otherwise the increment is visible. `OptionalIncrement` also defines two methods `remove` and `unremove` that modify the `removed` attribute in order to remove an increment or to expand it again. These methods are used by the implicit `parse`, `expand` and `unparse` methods of father increments. The `parse` method invokes `remove` or `unremove` of an optional abstract syntax child depending on whether the empty string is chosen or a match with a non-emtpy string is found. The `expand` method changes `removed` to `FALSE` for any removed optional child increment. The `unparse` method in turn only invokes an unparse on an optional child increment, if `removed` is `FALSE`. Class `OptionalIncrement` then defines an interaction for removing a selected increment. The interaction is only available if the increment has not yet been removed. Thus any subclass of `OptionalIncrement` is equipped with a command to remove the increment. A command to expand the child increment again cannot be defined in `OptionalIncrement`, because a removed child increment can no longer be selected at the user interface. It is defined in class `NonterminalIncrement` instead.

```
ABSTRACT INCREMENT INTERFACE OptionalIncrement;

  INHERIT Increment;

  EXPORT INTERFACE
    ATTRIBUTES
      HIDDEN removed:BOOLEAN;
    END ATTRIBUTES;

    METHODS
      METHOD remove();
      METHOD is_removed() : BOOLEAN;
      METHOD unremove();
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE OptionalIncrement.
```

B.5 NonterminalIncrement

Class `NonterminalIncrement` serves as an abstract super class for all increment classes that trace back to productions of a normalised EBNF that create further terminal and non-terminal symbols. It declares two deferred methods that will be redefined by implicit methods of non-terminal increment classes, namely `parse`, `expand`. Moreover, it defines an interaction that implements a command to expand an increment that is still a place holder or has removed optional child increments.

```
ABSTRACT INCREMENT INTERFACE NonterminalIncrement;

  INHERIT Increment;

  EXPORT INTERFACE
    METHODS
```

```
      DEFERRED METHOD parse(Str:STRING):NonterminalIncrement;
      DEFERRED METHOD expand();
    END METHODS;
 END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE NonterminalIncrement.
```

B.6 TerminalIncrement

Class `TerminalIncrement` is the counterpart to `NonterminalIncrement` for all increment classes that trace back to terminal symbols of a normalised EBNF. It defines two deferred methods and two explicit methods. The deferred method `scan` will be redefined by implicit methods of terminal increment classes. The two explicit methods `expand_terminal` and `change_terminal` use the deferred methods. With late binding, the respective implicit methods of subclasses will be invoked at run-time then. Moreover, `TerminalIncrement` defines two interactions that implement commands to expand or change the terminal increment.

```
ABSTRACT INCREMENT INTERFACE TerminalIncrement;

  INHERIT Increment;

  EXPORT INTERFACE
    METHODS
      DEFERRED METHOD scan(Str:STRING):BOOLEAN;
      METHOD ExpandTerminal(Str:STRING):BOOLEAN;
      METHOD ChangeTerminal(Str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE TerminalIncrement.
```

B.7 IncrementList

Class `IncrementList` serves as an abstract increment class for lists of increments. As such it defines an abstract syntax child `elems` whose type is `LIST OF Increment`[1]. Hence, arbitrary increments can be inserted into the list. It then defines a deferred method `expand`. Again, `expand` will be redefined by implicit methods of non-terminal increment classes. Moreover, `IncrementList` defines methods to expand the increment list with an element, to insert an element before the current element, to add an element after the current increment and to delete an element. These methods are then used by interactions that define all structure-oriented commands that are required to edit list increments.

The interactions that are offered for increment lists then investigate the static base type of the multi-valued abstract syntax child `elems`. They then compute the names of all terminal and non-terminal increment classes that are subclasses of this base type. Abstract increment classes are not included, because these must not be instantiated (A.54). We only consider the subclasses, because the subclasses of the base type represent all alternatives in the grammar and the polymorphism rule allows us only to insert these. If there are more than one subclass,

[1]This assures that only abstract and non-terminal increment classes can be defined as subclasses of `IncrementList` (A.13)

the interactions offer the class names in a submenu to the user. After the user has determined a name, the respective methods of this class are invoked with the user-defined class name as string parameter.

Note, that the bodies of these methods cannot be specified in GTSL. This is because GTSL intentionally does not include a concept for meta-classes. Meta-classes are required here since the class of the increment to be included in the list is determined at run-time by name, i.e. as a string. A meta-class is then required that knows about all classes and can lookup the class with the given name in order to instantiate an increment from the class. As this is inherently unsafe (there need not be a class of the given name and this cannot be checked statically) and we do not foresee that a tool builder might want to use meta-classes, we did not include them into GTSL. We rather only use the meta-class provided by the O_2 meta schema[2] in the implementation of this pre-defined class library and hide the mechanism from tool builders.

```
ABSTRACT INCREMENT INTERFACE IncrementList;

  INHERIT Increment;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF Increment;
    END ABSTRACT SYNTAX;

    METHODS
      DEFERRED METHOD expand();
      METHOD ExpandListWithIncrement(TheClass:STRING):SET OF STRING;
      METHOD AddElement(TheClass:STRING; cursor:Increment):SET OF STRING;
      METHOD InsertElement(TheClass:STRING; cursor:Increment):SET OF STRING;
      METHOD DeleteElement(i:Increment);
      METHOD CreateIncrement(TheClass:STRING):Increment;
      METHOD BaseSubClasses():SET OF STRING;
      METHOD BaseClass():STRING;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE IncrementList.
```

B.8 TerminalIncrementList

Class `TerminalIncrementList` specialises `IncrementList` in a way that only terminal increments can be included into `elems`. Therefore `TerminalIncrementList` redefines the type of `elems` to become `LIST OF TerminalIncrement`. Then it can apply method `ExpandTerminal` to list elements in order to define two further interactions for free textual input of new terminal increments to be inserted or added into the list.

```
ABSTRACT INCREMENT INTERFACE TerminalIncrementList;

  INHERIT IncrementList;

  IMPORT INTERFACE
```

[2]Note also, that this class could not be implemented in Eiffel, C++ or any of the C++ based ODBSs due to the lack of meta-classes. In these systems each increment class with a multi-valued abstract syntax child had to duplicate the implementation of the respective methods.

```
    IMPORT TerminalIncrement;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF TerminalIncrement;
    END ABSTRACT SYNTAX;

    METHODS
      METHOD ScanAndAddElement(TheClass:STRING; str:STRING;
                               cursor:TerminalIncrement):SET OF STRING;
      METHOD ScanAndInsertElement(TheClass:STRING; str:STRING;
                               cursor:TerminalIncrement):SET OF STRING;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE TerminalIncrementList.
```

B.9 NonterminalIncrementList

Class `NonterminalIncrementList` is declared as subclass of `IncrementList` in order to restrict the list elements to non-terminal increments into the list. It therefore redefines the type of `elems` to `LIST OF NonterminalIncrement`. Then it can import the deferred methods `unparse`, `parse` and `expand` from `NonterminalIncrement` in order to define various interactions for inserting or adding freely edited non-terminal increments into the list.

```
ABSTRACT INCREMENT INTERFACE NonterminalIncrementList;

  INHERIT IncrementList;

  IMPORT INTERFACE
    IMPORT NonterminalIncrement;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF NonterminalIncrement;
    END ABSTRACT SYNTAX;

    METHODS
      METHOD ExpandListWithIncrement(TheClass:STRING):SET OF STRING;
      METHOD AddElement(TheClass:STRING;
                        cursor:NonterminalIncrement) : SET OF STRING;
      METHOD InsertElement(TheClass:STRING;
                           cursor:NonterminalIncrement) : SET OF STRING;
      METHOD ParseAndAddElement(TheClass:STRING;str:STRING;
                                cursor:NonterminalIncrement):SET OF STRING;
      METHOD ParseAndInsertElement(TheClass:STRING;str:STRING;
                                   cursor:NonterminalIncrement):SET OF STRING;
      METHOD ParseAndExpandElement(str:STRING;
                                   cursor:NonterminalIncrement):SET OF STRING;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE NonterminalIncrementList.
```

B.10 UsingIncrement and UsableIncrement

Class `DocumentVersion` supports version management of documents. In order to support also management of different configurations of these versioned documents, we have to consider semantic relationships between documents. Classes `UsingIncrement` and `UsableIncrement` serve this purpose. `UsingIncrement` declares a semantic relationship with an explicit link to class `UsableIncrement`. `UsableIncrement` in turn declares the corresponding implicit link. In addition, `UsingIncrement` declares two attributes of type `STRING`, namely `UsedDoc` and `UsedVer` that uniquely identify the particular version that is is currently used. The attributes are used e.g. to traverse through the current configuration. Similarly, `UsableIncrement` stores the identifications of all documents and all versions that use the increment. Since this information is more complex, a particular class `VersionVectorTable` is used for this purpose (see below).

Classes `UsingIncrement`, `UsableIncrement` and `DocumentVersion` then define a number of interactions in order to implement configuration management commands. `UsingIncrement` has an interaction that displays the name and version of the currently used document. Likewise, `UsableIncrement` defines an interaction that displays the document and version names of documents that contain increments using the selected increment. On a more coarse grained level, `DocumentVersion` defines an interaction that implements a command for changing the currently used document version for all using increments contained in the current version of the document.

A tool builder might now exploit these configuration management mechanisms, by defining tool specific subclasses of `UsingIncrement` and `UsableIncrement`. In particular all configuration management commands will be available in the tool then. In these classes, the type of the links need to be restricted to the tool-specific increment classes by covariant redefinition. Moreover, semantic rules must be defined for the tool-specific subclasses of `UsingIncrement` that establish the inter-document semantic relationship in a tool-specific way. This is done by assigning a particular value to the explicit link `UsedIncrement`.

```
ABSTRACT INCREMENT INTERFACE UsingIncrement;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT UsableIncrement;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      HIDDEN UsedDoc : STRING;
      HIDDEN UsedVer : STRING;
      HIDDEN OwnerVersion : STRING;
    END ATTRIBUTES;

    SEMANTIC RELATIONSHIPS
      UsedIncrement: UsableIncrement;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      METHOD Init(f:Increment);
      METHOD Collapse();
      METHOD GetUsedDoc() : STRING;
      METHOD GetUsedVer() : STRING;
      METHOD GetAlreadyCreatedClientInc(CheckInc : UsingIncrement) : UsingIncrement;
      METHOD SetOwnerVersion(name:STRING);
      METHOD GetOwnerVersion() : STRING;
      METHOD RealizeVersionDependency(doc:STRING;ver:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE UsingIncrement.

ABSTRACT INCREMENT INTERFACE UsableIncrement;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT UsingIncrement;
    IMPORT VersionVectorTable;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      HIDDEN UsingVersions : VersionVectorTable;
    END ATTRIBUTES;

    METHODS
      METHOD Isolate();
      METHOD Init(f:Increment);
      METHOD RealizeIncomingUsingIncrement(inc:UsingIncrement);
      METHOD UnRealizeIncomingUsingIncrement(inc:UsingIncrement);
      METHOD RealizeDerivedVersion(ver:STRING);
      METHOD UsingDocs() : SET OF STRING;
      METHOD appendVersion(doc:STRING; ver:STRING);
      METHOD removeVersion(doc:STRING; ver:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE UsableIncrement.
```

B.11 DocumentPool

The class `DocumentPool` serves as a super class for classes that define persistent roots for documents. It defines an attribute `definedDocuments` of class `DocumentTable`. All documents that are associated with a name in this table will be persistent. The keys for entries in this table, however, are document names which are tool specific. They might correspond to module names, section titles, names of methods and the like. Therefore subclasses of `DocumentPool` define semantic rules that control associations in the symbol table based on changes of these document type specific name increments.

```
ABSTRACT INCREMENT INTERFACE DocumentPool;

INHERIT Increment;

IMPORT INTERFACE
  IMPORT Increment;
  IMPORT Document;
  IMPORT DocumentTable;
END IMPORT INTERFACE;

EXPORT INTERFACE
```

```
  ATTRIBUTES
    definedDocuments: DocumentTable;
  END ATTRIBUTES;

  SEMANTIC RELATIONSHIPS
    IMPLICIT registeredDocuments: SET OF Document.docPool;
  END SEMANTIC RELATIONSHIPS;

END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE DocumentPool.
```

B.12 Attribute

The only purpose of class `Attribute` is to serve as a super class of all non-syntactic classes. It only declares a deferred `init` methods in order to enforce definition of a method that is executed whenever an attribute is created.

```
INTERFACE Attribute;

EXPORT INTERFACE
  METHODS
    DEFERRED METHOD init();
  END METHODS;
END EXPORT INTERFACE;

END INTERFACE Attribute.
```

B.13 Error

Error descriptors are instances of class `Error`. Error has an integer attribute that stores the error codes and a method `get_string` that returns a formatted error message. This method is used by `get_set_of_errors` in `ErrorSet` in order to format its set of errors.

```
INTERFACE Error;

INHERIT Attribute;

  EXPORT INTERFACE
    CONSTRUCTION
      error: ERROR_TYPE;
    END CONSTRUCTION;

    METHODS
      METHOD init(e: ERROR_TYPE);
      METHOD get_string():STRING;
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE Error.
```

B.14 SyntaxError

Syntax errors are more special error descriptors. They add an attribute `pos` that is used to store two-dimensional coordinates. These coordinates identify where syntax errors occurred in a source text. The method `get_string` is redefined in order to also include the error position into the formatted error message.

```
INTERFACE SyntaxError;

  INHERIT Error;

  EXPORT INTERFACE
    CONSTRUCTION
      column, row: INTEGER;
    END CONSTRUCTION;

    METHODS
      METHOD init(e: ERROR_TYPE; x: INTEGER; y:INTEGER);
      METHOD get_string():STRING;
      // get_string uses get_string of the father class and
      // adds the position
     METHOD set_position(x: INTEGER; y: INTEGER);
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE SyntaxError.
{\footnotesize \begin{verbatim}
```

B.15 SymbolTable

Class `SymbolTable` efficiently manages a set of associations between symbols (represented as character strings) and increments. It exports methods `associate` to enter a new association, `deassociate` to delete an association and `increment_at` to search for the increment that is associated with a symbol.

An increment class where a new scope starts (such as a function in Pascal or a block in Algol) may then declare an attribute of class `SymbolTable`. It can then define semantic rules for all child increments that declare symbols in the scope. Then semantic rules may be defined for child increments that query the symbol table for uniqueness of symbols.

```
INTERFACE SymbolTable;

INHERIT Attribute;

  IMPORT INTERFACE
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    CONSTRUCTION
      st : DICTIONARY OF (KEY:STRING; VALUE:Increment)
    END CONSTRUCTION;

    METHODS
```

```
      METHOD associate(k:STRING; v:Increment);
      METHOD deassociate(k:STRING);
      METHOD includes(k:STRING) : BOOLEAN;
      METHOD increment_at(k:STRING) : Increment;
      METHOD size() : INTEGER;
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE SymbolTable.
```

B.16 DocumentTable

Class `DocumentPool` has an attribute that provides efficient access to all documents. This attribute is a symbol table, but all entries in this table are documents. In order to avoid type casts, we define a subclass of `SymbolTable`, which is `DocumentTable`. It redefines all methods that return entries to return documents rather than increments. In that way, we can access in `DocumentPool` and other classes documents from the table without having to type cast them.

```
INTERFACE DocumentTable;

INHERIT SymbolTable;

  IMPORT INTERFACE
    IMPORT Document;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    METHODS
      METHOD associate(k:STRING; v:Document);
      METHOD increment_at(k:STRING) : Document;
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE DocumentTable.
```

B.17 DocumentVersionTable

```
INTERFACE DocumentTable;

  INHERIT TableTable;

  IMPORT INTERFACE
    IMPORT DocumentVersion;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    METHODS
      METHOD associate(k:STRING; v:DocumentVersion);
      METHOD increment_at(k:STRING) : DocumentVersion;
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE DocumentVersionTable.
```

B.18 VersionVector and VersionVectorTable

A usable Increment may be used in more than one version of a document. For configuration management purposes, it is important to keep track of all these usages. Otherwise, we could not not support queries like "Which versions of a document use this increment". Therefore we need to associate additional information with implicit links of semantic relationships. Class `VersionVector` serves this purpose. It can store a list of version names in a hidden attribute and it provides access operation to enter or delete version names. Each entry in this vector then represents a version of the document that uses the increment.

As an increment may in fact be used from more than one document, we have to define a data structure that can efficiently manage a set of version vectors: one for each using document. Class `VersionVectorTable` is defined for that purpose. It has an attribute whose type is `DICTIONARY OF VersionVector`. Keys in this dictionary are considered as document names. They correspond to entries that are the version vectors of version names of that document that uses the increment.

The use of version vectors and version vector tables is completely transparent for a tool builder. These two classes are used in `UsableIncrement`. Entries are created or deleted, whenever a semantic relationship to a versioned usable increment is established. Its version vector table is queried during execution of a command that displays all documents and their respective versions that use the increment. This command is then inherited by all subclasses of `UsableIncrement`.

```
INTERFACE VersionVector;

  INHERIT Attribute;

  EXPORT INTERFACE
    CONSTRUCTION
      verList  : LIST OF STRING
    END CONSTRUCTION;

    METHODS
      METHOD appendVersion(name: STRING);
      METHOD removeVersion(name: STRING);
      METHOD GetVersionVector() : SET OF STRING;
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE VersionVector.


INTERFACE VersionVectorTable;

  INHERIT Attribute;

  IMPORT INTERFACE
    IMPORT VersionVector;
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    CONSTRUCTION
      TheVersions : DICTIONARY OF (KEY:STRING;VALUE:VersionVector)
    END CONSTRUCTION;
    METHODS
      METHOD GetVersions(docName : STRING) : SET OF STRING;
      METHOD appendVersion(docName : STRING; verName : STRING);
      METHOD removeVersion(docName : STRING; verName : STRING);
    END METHODS;
  END EXPORT INTERFACE;
END INTERFACE VersionVectorTable.
```

B.19 Summary

We have suggested a carefully designed library of pre-defined GTSL classes. This library fulfills three purposes. First of all, it defines the properties that are required from increments by the editor kernel. Most of them such as attribute `errors` in class `Increment` are explicitly defined in the library. Few, however, are defined as deferred (such as `GetDocName`). Second, it defines properties that are common to a high number of increment classes. Tool specification is significantly eased since these properties can be inherited by tool specific classes from library classes. Note, that we have allowed reuse by inheritance here although we do not want to have tool builders inheriting from other configurations. This is because in this way we can safely assure that all properties required by the editor kernel are available. Moreover, the library has been applied in a number of tool specifications and will quite unlikely have to be changed. Therefore, the rationale for not allowing inheritance among configurations does not hold here. Third, we have defined a number of classes (such as `SymbolTable`) that tool-builders may import and then reuse. This also demonstrates the way reuse is supported between different configurations.

Appendix C

GTSL Specification of GENESIS Tools

C.1 ENBNF Tool

C.1.1 Tool Configuration

```
CONFIGURATION ENBNF

   CONSISTS OF
     IMPORT FROM CONFIGURATION INT:
      INCREMENT CLASSES
        InterfacePool,
        Interface,
        TerminalInterface,
        ToolRootClass;
     END IMPORT;

   INCREMENT CLASSES
      ScopingBlock INHERIT Increment;
      NameInST INHERIT Increment;
      UsingNameInST INHERIT Increment;
      TerminalIncrement INHERIT Increment;
      NonterminalIncrement INHERIT Increment;
      IncrementList INHERIT NonterminalIncrement;
      TerminalIncrementList INHERIT IncrementList;
      NonterminalIncrementList INHERIT IncrementList;
      ENBNFPool INHERIT DocumentPool;
      ENBNF INHERIT Document, ScopingBlock;
      ProductionList INHERIT NonterminalIncrementList;
      Production INHERIT NonterminalIncrement;
      Alternative INHERIT Production;
      SymbolList INHERIT TerminalIncrementList;
      Regular INHERIT Production;
      StructureOpt INHERIT Production;
      RegularOpt INHERIT Production;
      Structure INHERIT Production;
      ComponentList INHERIT IncrementList;
      Component INHERIT Increment;
      ListProd INHERIT Component;
      Delimiter INHERIT OptionalIncrement;
      KeywordList INHERIT TerminalIncrementList;
      Keyword INHERIT Component,TerminalIncrement;
      RegExp INHERIT TerminalIncrement;
      DefiningSymbol INHERIT NameInST, TerminalIncrement;
      UsingSymbol INHERIT UsingNameInST, Component, TerminalIncrement;
    END INCREMENT CLASSES;

   ROOT INCREMENT IS ENBNF

   ADDITIONAL ERRORS
     #SymbAlreadyDef : "The given Symbol has already been defined";
     #SymbolNotDef : "The given Symbol has not yet been defined";
     #TooManyNts : "Too many nonterminals in the component list"
   END ADDITIONAL ERRORS
END CONFIGURATION ENBNF.
```

C.1.2 GTSL Class Definitions

C.1.2.1 Increment Class ScopingBlock

```
ABSTRACT INCREMENT INTERFACE ScopingBlock;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT DuplicateSymbolTable;
    IMPORT NameInST;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      DefinedNames: DuplicateSymbolTable;
    END ATTRIBUTES;

    SEMANTIC RELATIONSHIPS
      IMPLICIT envelopedNames: SET OF NameInST.envelopingScope;
    END SEMANTIC RELATIONSHIPS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE ScopingBlock.



INCREMENT SPECIFICATION ScopingBlock;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT NameInST INCLUDING envelopingScope,value;
    IMPORT DuplicateSymbolTable INCLUDING associate,deassociate;
  END IMPORT INTERFACE;

  INITIALIZATION
    DefinedNames := NEW DuplicateSymbolTable;
  END INITIALIZATION;

  SEMANTIC RULES
    // Associate/deassociate NameInST in Symboltable
    ON EXISTS(name : NameInST IN SELF.envelopedNames):
       CHANGED(name.value)
    ACTION
      SELF.DefinedNames.associate(name,name.value);
    END ACTION;

    ON EXISTS(name : NameInST IN SELF.envelopedNames):
       DELETED(name)
    ACTION
      SELF.DefinedNames.deassociate(name);
    END ACTION;
  END SEMANTIC RULES;
END INCREMENT SPECIFICATION ScopingBlock.
```

C.1.2.2 Increment Class `NameInST`

```
ABSTRACT INCREMENT INTERFACE NameInST;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ScopingBlock;
    IMPORT UsingNameInST;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      value: STRING;
      notDefined_error : ERROR_TYPE;
    END ATTRIBUTES;

    SEMANTIC RELATIONSHIPS
      envelopingScope: ScopingBlock;
      IMPLICIT UsedBy : SET OF UsingNameInST.DefinedIn;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      DEFERRED METHOD unparse():STRING;
      DEFERRED METHOD ErrorId():ERROR_TYPE;
      METHOD compute_envelopingScope():ScopingBlock;
    END METHODS;
  END EXPORT INTERFACE;



INCREMENT SPECIFICATION NameInST;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ScopingBlock INCLUDING DefinedNames;
    IMPORT DuplicateSymbolTable INCLUDING is_duplicate_incr;
    IMPORT UsingNameInST;
  END IMPORT INTERFACE;

  INITIALIZATION
    envelopingScope := SELF.compute_envelopingScope();
    notDefined_error := SELF.ErrorId();
  END INITIALIZATION;
  SEMANTIC RULES
    ON CHANGED(SELF.envelopingScope.DefinedNames)
    ACTION
      IF(SELF.envelopingScope.DefinedNames.is_duplicate_incr(SELF)) THEN
        SELF.Errors.append_error(notDefined_error);
      ELSE
        SELF.Errors.clear_error(notDefined_error);
      ENDIF
    END ACTION;
  END SEMANTIC RULES;

  METHODS
    METHOD compute_envelopingScope():ScopingBlock;
    VAR i: Increment;
    BEGIN
      i:=SELF.father;
      WHILE ((i!= NIL) AND (NOT i.IS_KIND_OF("ScopingBlock"))) DO
        i := i.father;
      ENDDO;
      RETURN(<ScopingBlock>i);
    END compute_envelopingScope;
  END METHODS;
END INCREMENT SPECIFICATION NameInST.
```

C.1.2.3 Increment Class `UsingNameInST`

```
ABSTRACT INCREMENT INTERFACE UsingNameInST;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ScopingBlock;
    IMPORT NameInST;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ATTRIBUTES
      value : STRING;
      usedClass : STRING;
      notDefined_error : ERROR_TYPE;
    END ATTRIBUTES;

    SEMANTIC RELATIONSHIPS
      envelopingScope : ScopingBlock;
      DefinedIn : NameInST;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      DEFERRED METHOD unparse():STRING;
      DEFERRED METHOD EnvelopingScope():ScopingBlock;
      DEFERRED METHOD DeclClassName(): STRING;
      DEFERRED METHOD ErrorId():ERROR_TYPE;
      DEFERRED METHOD react_on_change(Str:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE UsingNameInST.
```

```
INCREMENT SPECIFICATION UsingNameInST;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ScopingBlock INCLUDING DefinedNames;
    IMPORT DuplicateSymbolTable  INCLUDING increment_at;
    IMPORT NameInST;
  END IMPORT INTERFACE;

  INITIALIZATION
    envelopingScope := SELF.EnvelopingScope();
    usedClass := SELF.DeclClassName();
    notDefined_error := SELF.ErrorId();
  END INITIALIZATION;

  SEMANTIC RULES
    ON CHANGED(SELF.envelopingScope.DefinedNames)
    VAR inc: Increment;
    ACTION
      inc:=SELF.envelopingScope.DefinedNames.increment_at(SELF.value);
      IF inc != NIL THEN
        IF(inc.IS_KIND_OF(SELF.usedClass)) THEN
          DefinedIn := <NameInST>inc;
          Errors.clear_error(SELF.notDefined_error);
        ELSE
          DefinedIn:=<NameInST>NIL;
          Errors.append_error(SELF.notDefined_error);
        ENDIF;
      ELSE
        DefinedIn:=<NameInST>NIL;
        Errors.append_error(SELF.notDefined_error);
      ENDIF;
    END ACTION;

    ON CHANGED(SELF.value)
    VAR inc: Increment;
    ACTION
      inc:=SELF.envelopingScope.DefinedNames.increment_at(SELF.value);
      IF inc != NIL THEN
        IF(inc.IS_KIND_OF(SELF.usedClass)) THEN
          DefinedIn := <NameInST>inc;
          Errors.clear_error(SELF.notDefined_error);
        ELSE
          DefinedIn:=<NameInST>NIL;
          Errors.append_error(SELF.notDefined_error);
         ENDIF;
      ELSE
         DefinedIn:=<NameInST>NIL;
         Errors.append_error(SELF.notDefined_error);
      ENDIF;
    END ACTION;
  END SEMANTIC RULES;
END INCREMENT SPECIFICATION UsingNameInST.
```

C.1.2.4 Increment Class ENBNFPool

```
TERMINAL INCREMENT INTERFACE ENBNFPool;

  INHERIT DocumentPool;
  IMPORT INTERFACE
    IMPORT ENBNF;
    IMPORT Increment;
    IMPORT DocumentPool;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    SEMANTIC RELATIONSHIPS
      IMPLICIT registeredDocuments: SET OF ENBNF.docPool;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END TERMINAL INCREMENT INTERFACE ENBNFPool.


INCREMENT SPECIFICATION ENBNFPool;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ENBNF;
    IMPORT DocumentTable;
    IMPORT DocumentPool;
  END IMPORT INTERFACE;

  INITIALIZATION
    definedDocuments := NEW DocumentTable(SELF);
  END INITIALIZATION;
END INCREMENT SPECIFICATION ENBNFPool.
```

C.1.2.5 Increment Class ENBNF

```
NONTERMINAL INCREMENT INTERFACE ENBNF;

  INHERIT Document,ScopingBlock;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ProductionList;
    IMPORT ENBNFPool;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      pl: ProductionList;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      pl, (NL)
    END UNPARSING SCHEME;
    SEMANTIC RELATIONSHIPS
      docPool: ENBNFPool;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ENBNF;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ENBNF.



INCREMENT SPECIFICATION ENBNF;
END INCREMENT SPECIFICATION ENBNF.
```

C.1.2.6 Increment Class `ProductionList`

```
NONTERMINAL INCREMENT INTERFACE ProductionList;

  INHERIT NonterminalIncrementList;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Production;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF Production;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheList DELIMITED BY (NL), (NL) END
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ProductionList;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE ProductionList.



INCREMENT SPECIFICATION ProductionList;
END INCREMENT SPECIFICATION ProductionList.
```

C.1.2.7 Increment Class `Production`

```
ABSTRACT INCREMENT INTERFACE Production;

  INHERIT NonterminalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT DefiningSymbol;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      lhs : DefiningSymbol;
    END ABSTRACT SYNTAX;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE Production.



INCREMENT SPECIFICATION Production;
END INCREMENT SPECIFICATION Production.
```

C.1.2.8 Increment Class `Alternative`

```
NONTERMINAL INCREMENT INTERFACE Alternative;

  INHERIT Production;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT SymbolList;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      sl : SymbolList;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      lhs, (" "), "::=", (" ") , sl, (" "), "."
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Alternative;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE Alternative.



INCREMENT SPECIFICATION Alternative;
END INCREMENT SPECIFICATION Alternative.
```

C.1.2.9 Increment Class SymbolList

```
NONTERMINAL INCREMENT INTERFACE SymbolList;

  INHERIT TerminalIncrementList;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT TerminalIncrement;
    IMPORT UsingSymbol;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF UsingSymbol;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheList DELIMITED BY (" "), "|", (" ") END
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):SymbolList;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      // redefine two methods inherited from TerminalIncrementList
      // so as to propagate the insertion of an symbol into an alternative
      // production into the class interface tool and adjust the
      // inheritance hierarchy accordingly there
      METHOD ScanAndAddElement(TheClass:STRING;
                               str:STRING;
                               cursor:TerminalIncrement):SET OF STRING;
      METHOD ScanAndInsertElement(TheClass:STRING;
                                  str:STRING;
                                  cursor:TerminalIncrement):SET OF STRING;
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE SymbolList.



INCREMENT SPECIFICATION SymbolList;
  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT TerminalIncrement;
    IMPORT UsingSymbol;
    IMPORT UsingSymbol INCLUDING scan, has_error;
    IMPORT DefiningSymbol INCLUDING Class, unparse;
    IMPORT Interface INCLUDING AddInheritance;
    IMPORT Production INCLUDING lhs;
    IMPORT ENBNF INCLUDING DefinedNames;
    IMPORT DuplicateSymbolTable INCLUDING increment_at;
  END IMPORT INTERFACE;

  METHODS
    METHOD ScanAndAddElement(TheClass:STRING;
                             str:STRING;
                             cursor:TerminalIncrement):SET OF STRING;
    VAR anInc : UsingSymbol;
        Definition : DefiningSymbol;
    BEGIN
      anInc := NEW UsingSymbol(SELF);
      anInc.scan(str);
      IF ( NOT anInc.has_error()) THEN
        // anInc has not yet been defined or
        // has a lexical error
        RETURN( anInc.get_set_of_errors() );
      ELSE
        TheList.ADD_AFTER((<UsingSymbol> cursor), anInc );
        Definition:=<DefiningSymbol>(<ENBNF>myDocument).DefinedNames.increment_at(str);
        Definition.Class.AddInheritance((<Production>father).lhs.unparse());
        RETURN( anInc.get_set_of_errors() );
      ENDIF;
    END ScanAndAddElement;

    METHOD ScanAndInsertElement(TheClass:STRING;
                                str:STRING;
                                cursor:TerminalIncrement):SET OF STRING;
    VAR anInc : UsingSymbol;
        Definition : DefiningSymbol;
    BEGIN
      anInc := NEW UsingSymbol(SELF);
      anInc.scan(str);
      IF ( NOT anInc.has_error()) THEN
        RETURN( anInc.get_set_of_errors() );
      ELSE
        TheList.ADD_BEFORE((<UsingSymbol> cursor), anInc );
        Definition:=<DefiningSymbol>(<ENBNF>myDocument).DefinedNames.increment_at(str);
        Definition.Class.AddInheritance((<Production>father).lhs.unparse());
        RETURN( anInc.get_set_of_errors() );
      ENDIF;
    END ScanAndInsertElement;
  END METHODS;
END INCREMENT SPECIFICATION SymbolList.
```

C.1.2.10 Increment Class `Regular`

```
NONTERMINAL INCREMENT INTERFACE Regular;

  INHERIT Production;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT RegExp;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      re : RegExp;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      lhs, (" "), "::=" , (" "), re, (" "), "."
    END UNPARSING SCHEME;

    METHODS
```

```
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Regular;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE Regular.



INCREMENT SPECIFICATION Regular;
END INCREMENT SPECIFICATION Regular.
```

C.1.2.11 Increment Class `StructureOpt`

```
NONTERMINAL INCREMENT INTERFACE StructureOpt;

  INHERIT Production;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ComponentList;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      cl : ComponentList;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      lhs, (" "), "::=", (" "),
      "|",(" "),
      cl, (" "), "."
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):StructureOpt;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE StructureOpt.



INCREMENT SPECIFICATION StructureOpt;
END INCREMENT SPECIFICATION StructureOpt.
```

C.1.2.12 Increment Class `RegularOpt`

```
NONTERMINAL INCREMENT INTERFACE RegularOpt;

  INHERIT Production;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT RegExp;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      re : RegExp;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      lhs, (" "), "::=", (" "),
      "|",
      re, "."
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):RegularOpt;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE RegularOpt.



INCREMENT SPECIFICATION RegularOpt;
END INCREMENT SPECIFICATION RegularOpt.
```

C.1.2.13 Increment Class `Structure`

```
NONTERMINAL INCREMENT INTERFACE Structure;

  INHERIT Production;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ComponentList;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      cl : ComponentList;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      lhs, (" "), "::=", (" "), cl, (" "), "."
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Structure;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE Structure.



INCREMENT SPECIFICATION Structure;
END INCREMENT SPECIFICATION Structure.
```

C.1.2.14 Increment Class `ComponentList`

```
NONTERMINAL INCREMENT INTERFACE ComponentList;

  INHERIT IncrementList;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Component;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF Component;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheList DELIMITED BY (" ") END
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ComponentList;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ComponentList.



INCREMENT SPECIFICATION ComponentList;
END INCREMENT SPECIFICATION ComponentList.
```

C.1.2.15 Increment Class `Component`

```
ABSTRACT INCREMENT INTERFACE Component;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    METHODS
      DEFERRED METHOD isolate();
      DEFERRED METHOD collapse();
      DEFERRED METHOD unparse():STRING;
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE Component.


INCREMENT SPECIFICATION Component;
END INCREMENT SPECIFICATION Component.
```

C.1.2.16 Increment Class `ListProd`

```
NONTERMINAL INCREMENT INTERFACE ListProd;

  INHERIT Component;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT UsingSymbol;
    IMPORT Delimiter;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheSymbol : UsingSymbol;
      TheDelimiter : Delimiter;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "{", (" "),
      TheSymbol, (" "),
      "}",
      TheDelimiter
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ListProd;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ListProd.
```

```
INCREMENT SPECIFICATION ListProd;
END INCREMENT SPECIFICATION ListProd.
```

C.1.2.17 Increment Class `Delimiter`

```
NONTERMINAL INCREMENT INTERFACE Delimiter;

  INHERIT OptionalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT KeywordList;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : KeywordList;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "(",
      TheList,
      ")"
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Delimiter;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Delimiter.



INCREMENT SPECIFICATION Delimiter;
END INCREMENT SPECIFICATION Delimiter.
```

C.1.2.18 Increment Class `KeywordList`

```
NONTERMINAL INCREMENT INTERFACE KeywordList;

  INHERIT TerminalIncrementList;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Keyword;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      TheList : LIST OF Keyword;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheList DELIMITED BY (" ") END
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):KeywordList;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE KeywordList.


INCREMENT SPECIFICATION KeywordList;
END INCREMENT SPECIFICATION KeywordList.
```

C.1.2.19 Increment Class `Keyword`

```
TERMINAL INCREMENT INTERFACE Keyword;

  INHERIT Component,TerminalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    REGULAR EXPRESSION
      {["][^"]*["]}
    END REGULAR EXPRESSION;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
 END EXPORT INTERFACE;

END TERMINAL INCREMENT INTERFACE Keyword.


INCREMENT SPECIFICATION Keyword;
END INCREMENT SPECIFICATION Keyword.
```

C.1.2.20 Increment Class `RegExp`

```
TERMINAL INCREMENT INTERFACE RegExp;

  INHERIT TerminalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    REGULAR EXPRESSION
     {['][^']*[']}
    END REGULAR EXPRESSION;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      // Redefine the two methods inherited from TerminalIncrement
      // so as to inform the terminal increment interface that corresponds
      // to the definition about the regular expression as well.
      METHOD ExpandIdentifier(Str:STRING):BOOLEAN;
      METHOD ChangeIdentifier(Str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;
END TERMINAL INCREMENT INTERFACE RegExp.


INCREMENT SPECIFICATION RegExp;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Production INCLUDING lhs;
    IMPORT TerminalInterface INCLUDING CreateRegexp, ChangeRegexp;
    IMPORT DefiningSymbol INCLUDING Class;
  END IMPORT INTERFACE;

  METHODS
    METHOD ExpandIdentifier(Str:STRING):BOOLEAN;
    BEGIN
      IF SELF.scan(Str) THEN
        (<TerminalInterface>(<Production>father).lhs.Class).CreateRegexp(Str);
        RETURN(TRUE);
      ELSE
        RETURN(FALSE);
      ENDIF
    END ExpandIdentifier;

    METHOD ChangeIdentifier(Str:STRING):BOOLEAN;
    BEGIN
      IF SELF.scan(Str) THEN
        (<TerminalInterface>(<Production>father).lhs.Class).ChangeRegexp(Str);
        RETURN(TRUE);
      ELSE
        RETURN(FALSE);
      ENDIF
    END ChangeIdentifier;
  END METHODS;
END INCREMENT SPECIFICATION RegExp.
```

C.1.2.21 Increment Class `DefiningSymbol`

```
TERMINAL INCREMENT INTERFACE DefiningSymbol;

  INHERIT NameInST,TerminalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Interface;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    REGULAR EXPRESSION
      {[A-Za-z][A-Za-z0-9$_-]*}
    END REGULAR EXPRESSION;

    SEMANTIC RELATIONSHIPS
      Class: Interface;
      // This is an inter-document reference edge to the GTSL class
      // interface document that corresponds to the production that has
      // this defining symbol on the left hand side
      // The reference is established by methods expand_symbol and
      // change symbol.
    END SEMANTIC RELATIONSHIPS;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      METHOD ErrorId(): ERROR_TYPE;
      METHOD expand_symbol(new_symbol:STRING):BOOLEAN;
      METHOD change_symbol(new_symbol:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END TERMINAL INCREMENT INTERFACE DefiningSymbol.



INCREMENT SPECIFICATION DefiningSymbol;

  IMPORT INTERFACE
    IMPORT UsingNameInST;
    IMPORT Increment;
    IMPORT UsingSymbol INCLUDING react_on_change;
    IMPORT Interface INCLUDING AddInheritance, ChangeClassName;
    IMPORT ToolRootClass INCLUDING create_document;
  END IMPORT INTERFACE;

  METHODS
    METHOD ErrorId(): ERROR_TYPE;
    BEGIN
      RETURN (#SymbAlreadyDef);
    END ErrorId;

    METHOD expand_symbol(new:STRING):BOOLEAN;
    BEGIN
      IF (SELF.scan(new)) THEN
        IF father.IS_OF_CLASS("Alternative") THEN
          Class:=<Interface> INT.create_document("AbstractInterface",new,"RootVersion");
          Class.AddInheritance("Increment");
        ENDIF;
        IF father.IS_OF_CLASS("Structure") THEN
          Class:=<Interface> INT.create_document("NonterminalInterface",new,"RootVersion");
          Class.AddInheritance("NonterminalIncrement");
        ENDIF;
        IF father.IS_OF_CLASS("Regular") THEN
          Class:=<Interface> INT.create_document("TerminalInterface",new,"RootVersion");
          Class.AddInheritance("TerminalIncrement");
        ENDIF;
        IF father.IS_OF_CLASS("StructureOpt") THEN
          Class:=<Interface> INT.create_document("NonterminalInterface",new,"RootVersion");
          Class.AddInheritance("OptionalIncrement");
          Class.AddInheritance("NonterminalIncrement");
        ENDIF;
        IF father.IS_OF_CLASS("RegularOpt") THEN
          Class:=<Interface> INT.create_document("TerminalInterface",new,"RootVersion");
          Class.AddInheritance("OptionalIncrement");
          Class.AddInheritance("TerminalIncrement");
        ENDIF;
        RETURN(TRUE);
      ELSE
        RETURN(FALSE);
      ENDIF
    END expand_symbol;

    METHOD change_symbol(new:STRING):BOOLEAN;
    BEGIN
      FOREACH sym:UsingNameInST IN SELF.UsedBy DO
        (<UsingSymbol>sym).react_on_change(new)
      ENDDO;
      IF SELF.scan(new) THEN
        Class.ChangeClassName(new);
        RETURN(TRUE);
      ELSE
        RETURN(FALSE);
      ENDIF;
    END change_symbol;
  END METHODS;

  INTERACTIONS
    INTERACTION ExpandTheIdentifier;
    NAME "Expand Symbol"
    SELECTED IS SELF
    ON ( NOT SELF.is_phylum() )
    VAR new_name:TEXT;
        errors:TEXT_SET;
    BEGIN
      new_name:= NEW TEXT(SELF.unparse());
      IF new_name.LINE_EDIT("Change symbol:") THEN
        IF NOT SELF.expand_symbol(new_name.CONTENTS()) THEN
          errors:=NEW TEXT_SET(SELF.get_set_of_errors());
          errors.DISPLAY();
          ABORT
        ENDIF
      ENDIF
    END ExpandTheIdentifier;

    INTERACTION ChangeTheIdentifier;
    NAME "Change Symbol"
    SELECTED IS SELF
    ON ( NOT SELF.is_phylum() )
    VAR new_name:TEXT;
        errors:TEXT_SET;
    BEGIN
      new_name:= NEW TEXT(SELF.unparse());
      IF new_name.LINE_EDIT("Change symbol:") THEN
        IF NOT SELF.change_symbol(new_name.CONTENTS()) THEN
          errors:=NEW TEXT_SET(SELF.get_set_of_errors());
          errors.DISPLAY();
          ABORT
        ENDIF
      ENDIF
    END ChangeTheIdentifier;
  END INTERACTIONS;
END INCREMENT SPECIFICATION DefiningSymbol.
```

C.1.2.22 Increment Class UsingSymbol

```
TERMINAL INCREMENT INTERFACE UsingSymbol;
  INHERIT UsingNameInST, Component, TerminalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ScopingBlock;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    REGULAR EXPRESSION
      {[A-Za-z][A-Za-z0-9$_-]*}
    END REGULAR EXPRESSION;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD scan(Str:STRING):BOOLEAN;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      METHOD EnvelopingScope():ScopingBlock;
      METHOD ErrorId():ERROR_TYPE;
      METHOD DeclClassName():STRING;
      METHOD react_on_change(Str:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END TERMINAL INCREMENT INTERFACE UsingSymbol.



INCREMENT SPECIFICATION UsingSymbol;
  IMPORT INTERFACE
    IMPORT ScopingBlock;
    IMPORT Increment;
  END IMPORT INTERFACE;
  METHODS
    METHOD ErrorId(): ERROR_TYPE;
    BEGIN
      RETURN (#SymbolNotDef);
    END ErrorId;

    METHOD DeclClassName():STRING;
    BEGIN
      RETURN ("DefiningSymbol");
    END DeclClassName;

    METHOD react_on_change(Str:STRING);
    BEGIN
      SELF.scan(Str);
    END react_on_change;

    METHOD EnvelopingScope():ScopingBlock;
    VAR i: Increment;
    BEGIN
      i:=SELF.father;
      WHILE ((i!= NIL) AND (NOT i.IS_KIND_OF("ScopingBlock"))) DO
        i := i.father;
      ENDDO;
      RETURN(<ScopingBlock>i);
    END EnvelopingScope;
  END METHODS;
END INCREMENT SPECIFICATION UsingSymbol.
```

C.2 Class Interface Tool

C.2.1 Inheritance Diagram

Figure C.1 depicts the inheritance hierarchy of the subclasses of `NonterminalIncrement` in the class interface editor specification. The inheritance hierarchy of the other classes that are contained in the class interface editor specification were depicted on Page 222.

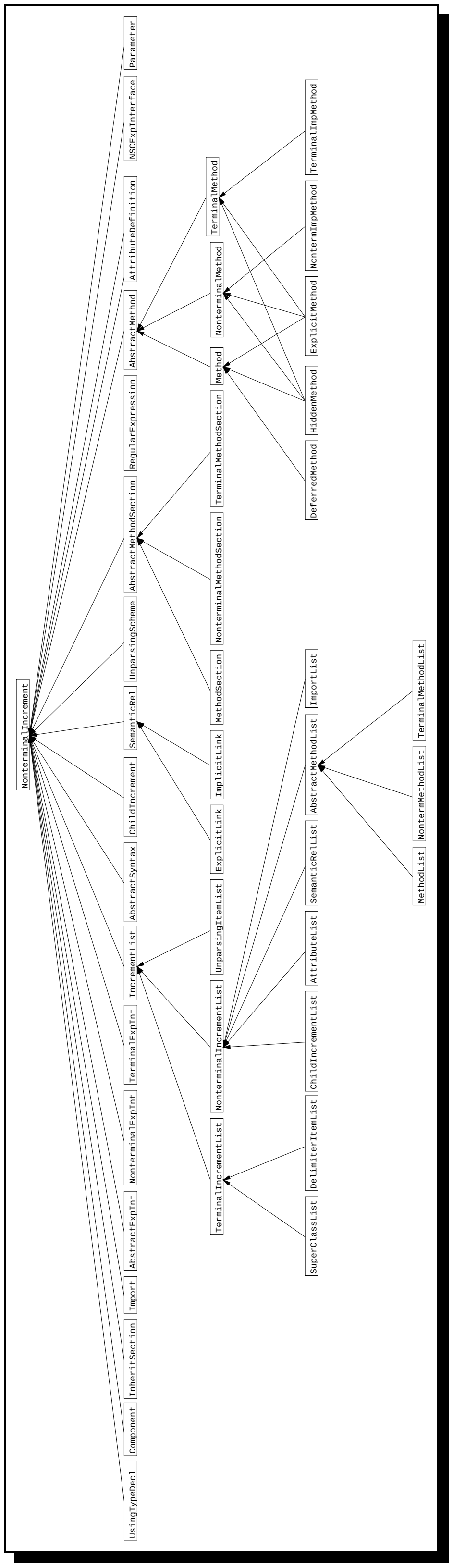


Figure C.1: Inheritance Hierarchy of Non-Terminal Increment Classes

C.2.2 Entity Relationship Diagrams

Figure C.2 displays the refinement of the subsystem `Interfaces` of the top-level entity relation diagram displayed in Figure 9.10 on Page 223.

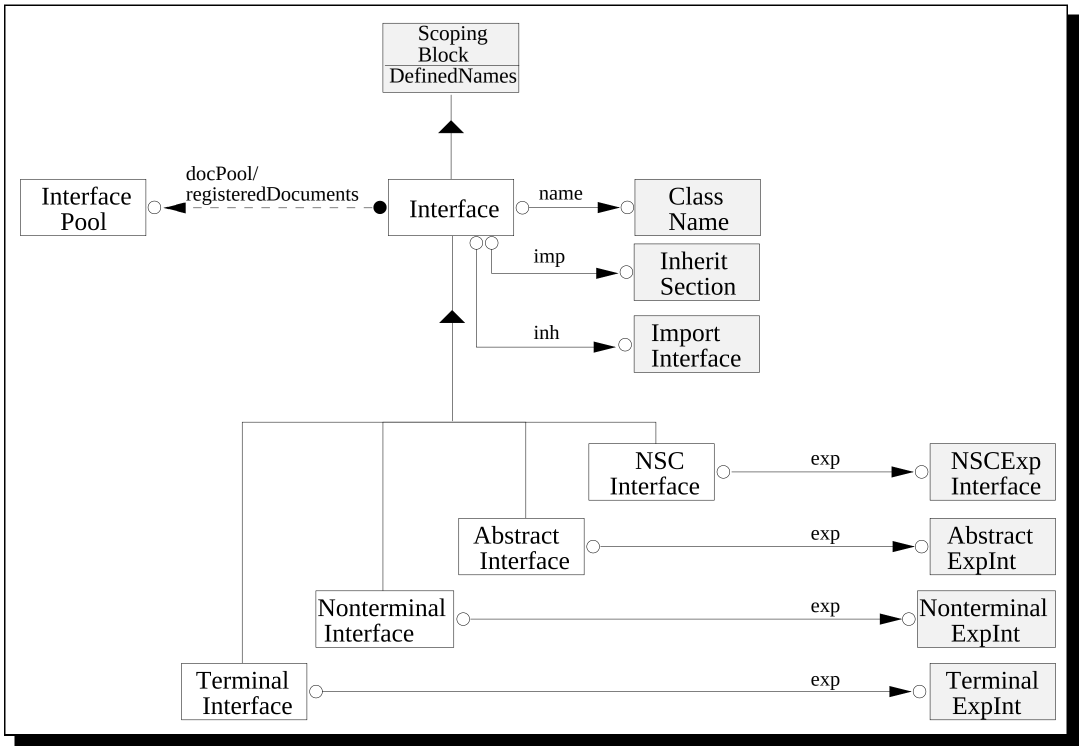


Figure C.2: Entity-Relationship Diagram `Interfaces`

Figures C.3-C.5 include the refinements of subsystems `RelationDefinitions`,`UnparsingItems` and `MethodDefinitions` that were included in the entity relationship diagram displayed in Figure 9.11 on Page 224.

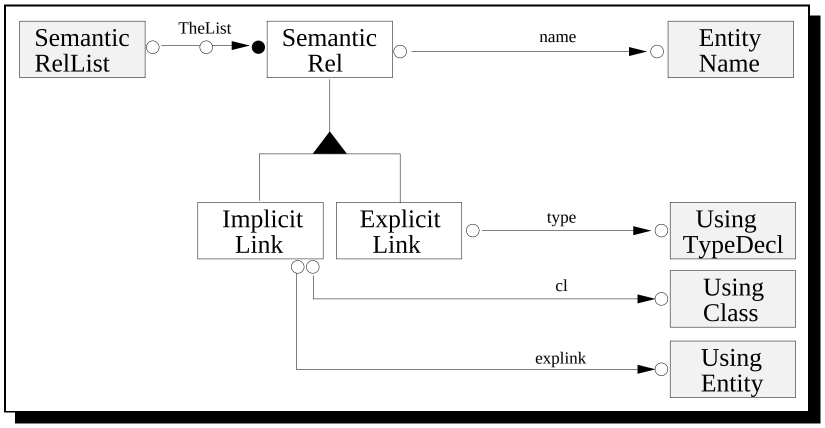


Figure C.3: Entity-Relationship Diagram `RelationDefinitions`

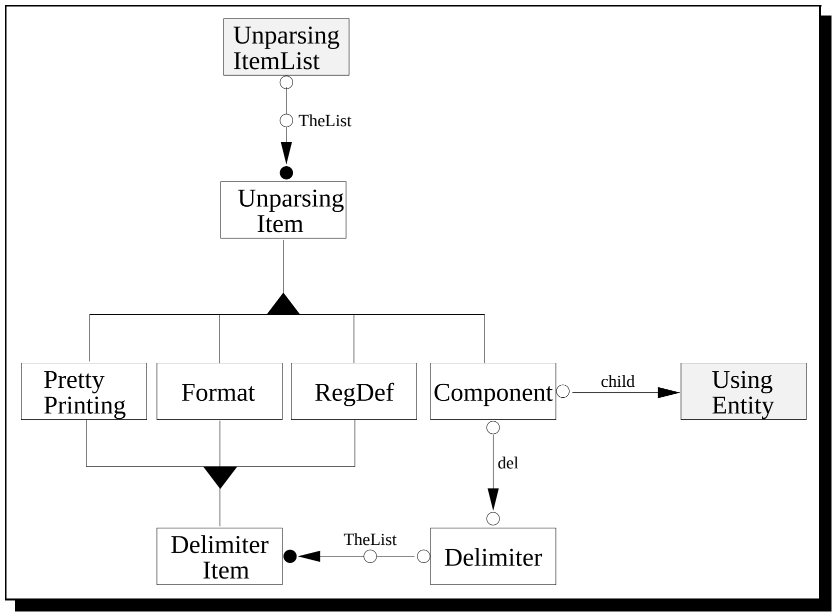


Figure C.4: Entity-Relationship Diagram `UnparsingItems`

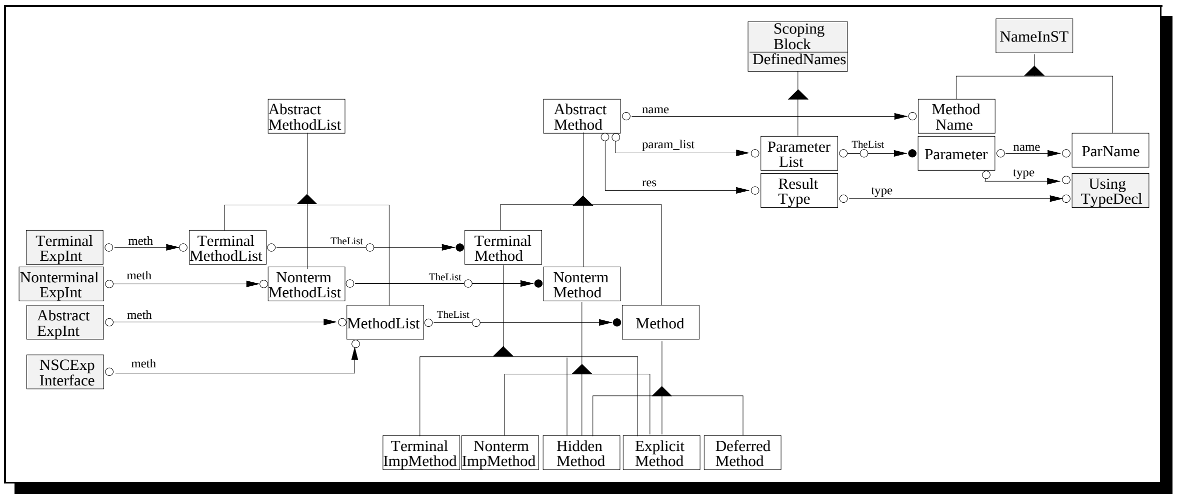


Figure C.5: Entity-Relationship Diagram `MethodDefinitions`

C.2.3 Tool Configuration

```
CONFIGURATION INT
  CONSISTS OF
    INCREMENT CLASSES
      ScopingBlock INHERIT Increment;
      TerminalIncrement INHERIT Increment;
      NameInST INHERIT TerminalIncrement;
      UsingNameInST INHERIT TerminalIncrement;
      NonterminalIncrement INHERIT Increment;
      OptionalNontermIncrement INHERIT  OptionalIncrement,NonterminalIncrement;
      IncrementList INHERIT NonterminalIncrement;
      TerminalIncrementList INHERIT IncrementList;
      NonterminalIncrementList INHERIT IncrementList;
      InterfacePool INHERIT DocumentPool;
```

```
Interface INHERIT DocumentVersion, ScopingBlock;
AbstractInterface INHERIT Interface;
NonterminalInterface INHERIT Interface;
TerminalInterface INHERIT Interface;
NSCInterface INHERIT Interface;
InheritSection INHERIT NonterminalIncrement;
SuperClassList INHERIT TerminalIncrementList;
ImportInterface INHERIT OptionalNontermIncrement;
ImportList INHERIT NonterminalIncrementList;
Import INHERIT NonterminalIncrement;
AbstractExpInt INHERIT NonterminalIncrement;
NonterminalExpInt INHERIT NonterminalIncrement;
TerminalExpInt INHERIT NonterminalIncrement;
NSCExpInterface INHERIT NonterminalIncrement;
Construction INHERIT OptionalNontermIncrement;
AbstractSyntax INHERIT OptionalNontermIncrement;
ChildIncrementList INHERIT NonterminalIncrementList;
ChildIncrement INHERIT NonterminalIncrement;
UnparsingScheme INHERIT NonterminalIncrement;
UnparsingItemList INHERIT IncrementList;
UnparsingItem INHERIT Increment;
Component INHERIT UnparsingItem,NonterminalIncrement;
DelimiterItem INHERIT TerminalIncrement;
RegDef INHERIT UnparsingItem,DelimiterItem;
Format INHERIT UnparsingItem,DelimiterItem;
PrettyPrinting INHERIT UnparsingItem,DelimiterItem;
Delimiter INHERIT OptionalNontermIncrement;
DelimiterItemList INHERIT TerminalIncrementList;
RegularExpression INHERIT NonterminalIncrement;
Attributes INHERIT OptionalNontermIncrement;
AttributeList INHERIT NonterminalIncrementList;
AttributeDefinition INHERIT NonterminalIncrement;
AttributeCategory INHERIT OptionalNontermIncrement;
SemanticRelations INHERIT OptionalNontermIncrement;
SemanticRelList INHERIT NonterminalIncrementList;
SemanticRel INHERIT NonterminalIncrement;
ExplicitLink INHERIT SemanticRel;
ImplicitLink INHERIT SemanticRel;
AbstractMethodSection INHERIT NonterminalIncrement;
MethodSection INHERIT AbstractMethodSection;
NonterminalMethodSection INHERIT AbstractMethodSection;
TerminalMethodSection INHERIT AbstractMethodSection;
AbstractMethodList INHERIT NonterminalIncrementList;
MethodList INHERIT AbstractMethodList;
NontermMethodList INHERIT AbstractMethodList;
TerminalMethodList INHERIT AbstractMethodList;
AbstractMethod INHERIT NonterminalIncrement;
Method INHERIT AbstractMethod;
NonterminalMethod INHERIT AbstractMethod;
TerminalMethod INHERIT AbstractMethod;
NontermImpMethod INHERIT NonterminalMethod;
DeferredMethod INHERIT Method;
TerminalImpMethod INHERIT TerminalMethod;
HiddenMethod INHERIT Method, NonterminalMethod, TerminalMethod;
ExplicitMethod INHERIT Method, NonterminalMethod, TerminalMethod;
ParameterList INHERIT ScopingBlock, OptionalIncrement, NonterminalIncrementList;
Parameter INHERIT NonterminalIncrement;
ResultType INHERIT OptionalNontermIncrement;
UsingTypeDecl INHERIT NonterminalIncrement;
MultiValue INHERIT OptionalIncrement, TerminalIncrement;
UsingType INHERIT UsingNameInST;
```

```
      RegExp INHERIT TerminalIncrement;
      ClassDecl INHERIT NameInST;
      ClassName INHERIT ClassDecl;
      SuperClass INHERIT ClassDecl;
      ImportClass INHERIT ClassDecl;
      EntityName INHERIT NameInST;
      ParName INHERIT NameInST;
      MethName INHERIT NameInST;
      UsingEntity INHERIT UsingNameInST;
      UsingClass INHERIT UsingNameInST;
   END INCREMENT CLASSES;

   ROOT INCREMENT IS Interface

   EXPORT INCREMENT CLASSES:
      InterfacePool,
      Interface,
      TerminalInterface,
      NonterminalInterface,
      AbstractInterface;
   END EXPORT;

   ADDITIONAL ERRORS
     #IdAlreadyDefined : "The given name has already been defined";
     #NotAnAbstractSyntaxChild : "Component must denote abstract syntax child";
     #UnknownType : "The given type is not known";
     #UnknownClass : "The given class is not known";
     #CyclicInheritance : "The inheritance relationship is cyclic";
     #IncorrectMultipleInh : "Incorrect multiple inheritance";
     #IncorrectRepeatedInh : "Incorrect repeated inheritance"
   END ADDITIONAL ERRORS
END CONFIGURATION INT.
```

C.2.4 GTSL Class Definitions

In this section, we list the GTSL class interface and specification definitions of the three classes that model GTSL increment classes, namely `AbstractInterface`, `NonterminalInterface` and `TerminalInterface`. They inherit from class `Interface` which is also included. These classes are used by the ENBNF Editor. `Interface` is the target class of a semantic relationship defined in the ENBNF class `DefiningSymbols`. It is exploited for propagating changes of a production to the respective GTSL class interface. They are executed with methods that are defined in these classes and that were discussed in Subsection 9.3.3. For reasons of brevity, the other 70 classes have been omitted.

C.2.4.1 Increment Class `Interface`

```
ABSTRACT INCREMENT INTERFACE Interface;

  INHERIT DocumentVersion, ScopingBlock;

  IMPORT INTERFACE
    IMPORT ClassName;
    IMPORT InheritSection;
    IMPORT DuplicateSymbolTable;
    IMPORT InterfacePool;
    IMPORT ImportInterface;
    IMPORT Increment;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      name:ClassName;
      inh:InheritSection;
      imp:ImportInterface;
    END ABSTRACT SYNTAX;

    SEMANTIC RELATIONSHIPS
      docPool: InterfacePool;
    END SEMANTIC RELATIONSHIPS;

    METHODS
      DEFERRED METHOD isolate();
      DEFERRED METHOD collapse();
      DEFERRED METHOD parse(Str:STRING):Interface;
      DEFERRED METHOD check();
      DEFERRED METHOD unparse():STRING;
      METHOD AddInheritance(Str:STRING):BOOLEAN;
      // Adds <Str> to the inheritance section
      METHOD ExpandClassName(Str:STRING):BOOLEAN;
      // expands the class name to <Str>
      METHOD ChangeClassName(Str:STRING):BOOLEAN;
      // changes the class name to <Str> and propagates change
      // to all using increments
      METHOD CreateAtomicTypes():DuplicateSymbolTable;
      // creates a new symbol table with all pre-defined atomic types.
    END METHODS;
  END EXPORT INTERFACE;
END ABSTRACT INCREMENT INTERFACE Interface.


INCREMENT SPECIFICATION Interface;
  IMPORT INTERFACE
    IMPORT ClassName;
    IMPORT InheritSection;
    IMPORT InterfacePool;
    IMPORT ImportInterface;
    IMPORT Increment;
    IMPORT DuplicateSymbolTable INCLUDING associate;
    IMPORT EntityName INCLUDING scan;
    IMPORT UsingNameInST INCLUDING react_on_change;
    IMPORT InheritSection INCLUDING scl, is_phylum, expand;
    IMPORT SuperClassList INCLUDING expand, TheList;
    IMPORT SuperClass INCLUDING scan, collapse;
    IMPORT ClassName INCLUDING is_phylum, scan, UsedBy;
  END IMPORT INTERFACE;

  INITIALIZATION
    DefinedNames:=SELF.CreateAtomicTypes();
  END INITIALIZATION;

  METHODS
    METHOD AddInheritance(Str:STRING):BOOLEAN;
    VAR sc:SuperClass;
    BEGIN
      IF inh.is_phylum() THEN
        inh.expand();
      ENDIF;
      IF inh.scl.is_phylum() THEN
        inh.scl.expand();
      ENDIF;
      sc:=NEW SuperClass(inh.scl);
      IF sc.scan(Str) THEN
        inh.scl.TheList.ADD_LAST(sc);
        RETURN(TRUE);
      ELSE
        sc.collapse();
        RETURN(FALSE);
      ENDIF;
    END AddInheritance;

    METHOD ExpandClassName(Str:STRING):BOOLEAN;
    BEGIN
      IF name.is_phylum() THEN
        RETURN(name.scan(Str));
      ENDIF;
    END ExpandClassName;

    METHOD ChangeClassName(Str:STRING):BOOLEAN;
    BEGIN
      FOREACH i:UsingNameInST IN name.UsedBy DO
        i.react_on_change(Str);
      ENDDO;
      RETURN(name.scan(Str));
    END ChangeClassName;

    METHOD CreateAtomicTypes():DuplicateSymbolTable;
    VAR i:EntityName;
        st:DuplicateSymbolTable;
    BEGIN
      st:=NEW DuplicateSymbolTable;
      st.associate(SELF,"BOOLEAN");
      st.associate(SELF,"INTEGER");
      st.associate(SELF,"STRING");
      st.associate(SELF,"TEXT");
      st.associate(SELF,"TEXT_SET");
      RETURN(st);
    END CreateAtomicTypes;
  END METHODS;
END INCREMENT SPECIFICATION Interface.
```

C.2.4.2 Increment Class `AbstractInterface`

```
NONTERMINAL INCREMENT INTERFACE AbstractInterface;

  INHERIT Interface;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT AbstractExpInt;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
```

```
      exp : AbstractExpInt;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "ABSTRACT", WS, "INCREMENT", WS, "INTERFACE", WS, name, ";", (NL),(NL),
      ("  "), inh, (NL),
      ("  "), imp, (NL),
      ("  "), exp, (NL),
      "END", WS, "ABSTRACT", WS, "INCREMENT", WS, "INTERFACE", WS, name, ".", (NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):AbstractInterface;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE AbstractInterface.



INCREMENT SPECIFICATION AbstractInterface;
END INCREMENT SPECIFICATION AbstractInterface.
```

C.2.4.3 Increment Class `NonterminalInterface`

```
NONTERMINAL INCREMENT INTERFACE NonterminalInterface;

  INHERIT Interface;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT NonterminalExpInt;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
      exp : NonterminalExpInt;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "NONTERMINAL", WS, "INCREMENT", WS, "INTERFACE", WS, name, ";",(NL),(NL),
      ("  "), inh, (NL),
      ("  "), imp, (NL),
      ("  "), exp, (NL),
      "END", WS, "NONTERMINAL", WS, "INCREMENT", WS, "INTERFACE", WS, name, ".", (NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):NonterminalInterface;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      METHOD CreateChild(After:STRING;Str:STRING; IsList:BOOLEAN):BOOLEAN;
      // Adds child increment after child increment of class <After>
      // If After is emtpy string child becomes last string
      // expands list elements in unparsing scheme if <IsList>=TRUE
      METHOD DeleteChild(Str:STRING):BOOLEAN;
      // Deletes Child <Str>
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE NonterminalInterface.



INCREMENT SPECIFICATION NonterminalInterface;
  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT NonterminalExpInt INCLUDING as, unp;
    IMPORT AbstractSyntax INCLUDING childlst, expand;
    IMPORT ChildIncrementList INCLUDING expand,TheList;
    IMPORT UnparsingScheme INCLUDING uilst, expand;
    IMPORT UnparsingItemList INCLUDING TheList, expand;
    IMPORT UnparsingItem INCLUDING collapse;
    IMPORT ChildIncrement INCLUDING name,expand,unparse,collapse;
    IMPORT EntityName INCLUDING scan;
    IMPORT Component INCLUDING child,del,expand,unparse,collapse;
    IMPORT Delimiter INCLUDING expand, remove;
    IMPORT UsingEntity INCLUDING scan, unparse;
  END IMPORT INTERFACE;

  METHODS
    METHOD CreateChild(After:STRING;Str:STRING; IsList:BOOLEAN):BOOLEAN;
    VAR ci, cicursor:ChildIncrement;
        ui: Component;
        cd: Interface;
    BEGIN
      IF exp.is_phylum() THEN // ExporInterface is still placeholder
        exp.expand();
      ENDIF;
      IF exp.as.is_phylum() THEN // abstract syntax section is still placeholder
        exp.as.expand();
      ENDIF;
      IF exp.as.childlst.is_phylum() THEN // childlst is still placeholder
        exp.as.childlst.expand();
      ENDIF;
      ci:=NEW ChildIncrement(exp.as.childlst); // create new child increment
      ci.expand();
      IF exp.unp.is_phylum() THEN    // unparsing section is still placeholder
        exp.unp.expand();
      ENDIF;
      IF exp.unp.uilst.is_phylum() THEN  // unparsing item list still placeholder
        exp.unp.uilst.expand()
      ENDIF;
      IF (ci.name.scan(Str)) THEN         // Str is lexically correct
        ui:=NEW Component(exp.unp.uilst); // create a new unparsing
        ui.expand();                             // item for component
        ui.child.scan(Str);                      // store the value as well
        IF IsList THEN                           // if component is a list
          ui.del.expand();                       // expand list place holder
        ELSE                                     // otherwise
          ui.del.remove();                         // remove it
        ENDIF;
        IF After="" THEN                         // append elements to list
```

```
          exp.as.childlst.TheList.ADD_LAST(ci);
          exp.unp.uilst.TheList.ADD_LAST(ui);
        ELSE                                        // search for insertion position
          cicursor:=<ChildIncrement>DefinedNames.increment_at(After);
          exp.as.childlst.TheList.ADD_AFTER(cicursor,ci);
          FOREACH cur:UnparsingItem IN exp.unp.uilst.TheList DO
            IF cur.IS_OF_CLASS("Component") THEN
              IF NOT cur.is_phylum() AND
                 (<Component>cur).child.unparse()=After THEN
                exp.unp.uilst.TheList.ADD_AFTER(cur,ui);
              ENDIF
            ENDIF
          ENDDO;
        ENDIF;
        RETURN(TRUE);
      ELSE
        ci.collapse();
        RETURN(FALSE);
      ENDIF;
    END CreateChild;

    METHOD DeleteChild(Str:STRING):BOOLEAN;
    BEGIN
      // search for component in unparsing item list to delete
      FOREACH cur:UnparsingItem IN exp.unp.uilst.TheList DO
        IF cur.IS_OF_CLASS("Component") THEN
          IF NOT cur.is_phylum() AND
             (<Component>cur).child.unparse()=Str THEN
            exp.unp.uilst.TheList.DELETE(cur);
            cur.collapse();
          ENDIF
        ENDIF
      ENDDO;
      // search for abstract syntax child to delete
      FOREACH cur:ChildIncrement IN exp.as.childlst.TheList DO
        IF cur.unparse()=Str THEN
          exp.as.childlst.TheList.DELETE(cur);
          cur.collapse();
        ENDIF
      ENDDO;
      RETURN(TRUE);
    END DeleteChild;
  END METHODS;
END INCREMENT SPECIFICATION NonterminalInterface.
```

C.2.4.4 Increment Class `TerminalInterface`

```
NONTERMINAL INCREMENT INTERFACE TerminalInterface;

  INHERIT Interface;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT TerminalExpInt;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
```

```
    ABSTRACT SYNTAX
      exp : TerminalExpInt;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "TERMINAL", WS, "INCREMENT", WS, "INTERFACE", WS, name, ";",(NL), (NL),
      ("  "), inh, (NL),
      ("  "), imp, (NL),
      ("  "),exp, (NL),
      "END", WS, "TERMINAL", WS, "INCREMENT", WS, "INTERFACE", WS, name, ".", (NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):TerminalInterface;
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      METHOD CreateRegexp(Str:STRING):BOOLEAN;
      // expands regular expression of regular expression section to <Str>
      // returns false if <Str> is not a correct regular expression or
      // if it has already been expanded, otherwise it returns TRUE
      METHOD ChangeRegexp(Str:STRING):BOOLEAN;
      // changes regular expression of regular expression section to <Str>
      // returns false if <Str> is not a correct regular expression or
      // has not yet been expnaded, otherwise it returns TRUE
    END METHODS;
  END EXPORT INTERFACE;
END NONTERMINAL INCREMENT INTERFACE TerminalInterface.


INCREMENT SPECIFICATION TerminalInterface;
  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT TerminalExpInt INCLUDING is_phylum, expand, re, is_phylum;
    IMPORT RegularExpression INCLUDING is_phylum, expand, re, is_phylum;
    IMPORT RegExp INCLUDING scan, is_phylum;
  END IMPORT INTERFACE;

  METHODS
    METHOD CreateRegexp(Str:STRING):BOOLEAN;
    BEGIN
      IF exp.is_phylum() THEN
        exp.expand();
      ENDIF;
      IF exp.re.is_phylum() THEN
        exp.re.expand();
      ENDIF;
      IF exp.re.re.is_phylum() THEN
        RETURN(exp.re.re.scan(Str));
      ELSE
        RETURN(FALSE);
      ENDIF;
    END CreateRegexp;

    METHOD ChangeRegexp(Str:STRING):BOOLEAN;
    BEGIN
      IF NOT exp.is_phylum() AND
         NOT exp.re.is_phylum() AND
         NOT exp.re.re.is_phylum() THEN
        RETURN(exp.re.re.scan(Str))
      ELSE
        RETURN(FALSE);
      ENDIF
    END ChangeRegexp;
  END METHODS;
END INCREMENT SPECIFICATION TerminalInterface.
```