Flexible heterogeneous software systems

de Jong, H.A.

Publication date
2007
Document Version
Final published version

[Link to publication](#)

Citation for published version (APA):
de Jong, H. A. (2007). *Flexible heterogeneous software systems*. [Thesis, externally prepared, Universiteit van Amsterdam].

Flexible Heterogeneous Software Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P.F. van der Heijden
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op donderdag 1 februari 2007, te 10.00 uur

door

Hayco Alexander de Jong
geboren te Zaandam

Acknowledgements

First of all I would like to thank Paul Klint for his constructive and inspiring mentorship. You were always there to discuss any research ideas, and helped me (re-)focus when I got lost in my own train of thoughts, without ever giving me the feeling I was forced to see things your way. Thank you for giving me both the guidance and the freedom I needed to complete this thesis!

Next I thank Mark van den Brand who, through his own perseverance, motivated me to go on at times when I was ready to pull the plug. Your ongoing desire to extend and improve the Meta-Environment ultimately sparked my ideas for the current plug-in architecture. Also, you and your wife José made the best *kinderboerderij* stew I ever had! ;-)

I would like to thank the members of the reading committee for reading this thesis and for providing valuable feedback: prof. dr. Uwe Aßmann, prof. dr. Jan Bergstra, prof. dr. Martin Kersten, prof. dr. ir. Koos Rooda, prof. dr. Hans van Vliet.

Many thanks also to Albert Hofkamp for thoroughly reading my thesis, and giving me valuable feedback.

My friendship with Pieter Olivier has grown during my thesis years in many ways. In a professional way, together with Maud Olivier-Weddepohl, we are now real business partners. But I also enjoyed all the social things we shared, such as our interest in gaming and hiking. For the completion of this thesis we have shared many motivating talks while exploring and enjoying the wonderful Veluwe at the same time. Thanks for all your support so far, and may our way of life and view on matters remain as compatible for many years to come, as they are now!

Had I not met Taeke Kooiker at CWI, the computer I used to write this thesis would have been another MyCom product. Instead, I now know exactly how to build (and have built) my own computer from various hardware components. Not only have we spent many hours working together on SEN1 software, we also shared many exciting discussions on other topics, all of which I really enjoyed, and which have enriched my life.

In the past six years at CWI, many people have at some point been part of the SEN1 research group. They have all contributed in one way or another to my thesis. Discussions in the hallway, sharing research views over lunch, having cake and coffee together, (re-)using each others software. The open and harmonious way we worked together in SEN1, with ample room for constructive criticism, has always inspired and motivated me very much.

Contents

Part I

Overview

CHAPTER 1

Introduction

The subject of study in this thesis is the interaction between software components in applications. In particular, I am interested in techniques that help developers build and maintain complex applications built up from individual components, each potentially written in a different programming language.

The overall question is how to make these components work together in a systematic way, such that when individual components of the application are changed, the application as a whole does not collapse. The main technical issues addressed in this thesis are the exchange of structured data between heterogeneous components, systematically addressing compound functionality of an application by coordinating individual components, and centralizing the user interaction of separate components.

The overall result of this research is a framework for the development of complex applications, which offers developers type safe access to a space efficient data exchange layer, a systematic way to describe the cooperation between individual components, and a centralized graphical user interface for the user interaction with individual components and the application as a whole.

In this introductory chapter, we layout the objectives and requirements of the software engineering perspective of developing applications in a component based way. We then introduce the technological background of this thesis: the component coordination architecture called ToolBus [13], and its application in the ASF+SDF Meta-Environment [24]. We focus on two aspects of component coordination: data exchange and component interaction. We identify research questions in both areas, and conclude with a roadmap for the remaining chapters of this thesis.

1.1 Software re-use

Why re-invent the wheel, if there are plenty of wheels for sale, which can be equipped with tires ready for every season? Most engineering disciplines focus their activities around readily available components. Mechanical engineers use standard bolts and nuts, electrical engineers use chips and circuits which have been tried and tested in other systems, etc. Not only small components (bolts, chips) are re-used, the same principle is applied to larger sub-systems, such as an engine. Computer manufacturers

select a case, power supply, mainboard, processor, memory, and handful of storage components to build a desktop.

The same principle of re-use can be applied to software engineering. Substitute functions or methods for the nuts and bolts and you end up with a low-level kind of software re-use. Substitute a text-editing component for the engine, and you have an example of application re-use. Obviously there are benefits to reusing software components, which [110] sums up as follows:

Increased dependability As re-used software has been tried and tested in other working systems, it *should* be more dependable than new software.

Reduced process risk The cost of *existing* software is known, but the costs of *development* are a matter of judgement.

Effective use of specialists Application specialists can develop reusable software encapsulating their knowledge, rather than doing the same work over and over.

Standards compliance By reusing, e.g., user interface elements such as menu items, applications offer a more consistent look and feel to application users, making it less likely they make mistakes when presented with a familiar interface.

Accelerated development Re-use of software *can* speed up system production as both development and validation time should be reduced. In a world where time to market is often more important than overall development cost, development speed is highly relevant.

The goal of software re-use is the reduction of overall development costs, faster delivery of systems, and increased software quality.

However, re-use of almost every (software) component comes at a cost. One immediate concern is always that of *adaptation*. How will we fit the re-used wheel fit our chassis? But also more sociological issues may arise as software engineers may prefer to write their own versions of components thinking they can improve on them (the "Not-invented-here syndrome" [110]). Finally, finding the right software component may not be as easy as selecting the right wheel for a car, as software components often are not as well classified and catalogued as components in other engineering disciplines are.

In this thesis, various chapters are concerned with elements of software re-use. The ATerm-Library presented in Chapter 2 is an example of function re-use in the form of a standard library. Chapter 3 describes how a program generation approach can be used to reduce the adaptation cost of reusing a standard library. In Chapter 5 we study how off-the-shelf components can be re-used in an application at minimal costs. Finally, in Chapter 6 we study software architectures ("plugin frameworks") that allow for later extensions, thereby offering the entire application as reusable component to the extension.

1.1.1 Software Architecture

A software architecture can be defined as follows [9]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

The externally visible properties mentioned in this definition are *assumptions* which other elements can make of an element, e.g., provided services, performance characteristics, fault handling, etc.

In order to reason about the quality of a software architecture, [9] uses six criteria. We summarize these as follows:

Availability Availability is concerned with system failure and its associated consequences. How is system failure detected? What is the mean time to failure, and in case of a failure how long does it take to repair the system?

Modifiability Modifiability is about the cost of change. The main concerns are: *what* can change, *when* is it changed, and *who* makes the change? Changes can occur to, e.g., the functionality of the system, the platform it exists on, protocols used to communicate with the rest of the world, etc. Changes can be made by modifying the source code, during compilation by changing compile-time switches, during configuration, or during execution. Some of these changes (e.g., source code changes) can only be performed by a developer, while others can be made by end-users (e.g., changing the screen saver). Each change, once specified, leads to design, implementation, testing, and deployment costs in terms of time and money, which can be measured.

Performance Performance is all about timing. Events such as interrupts, incoming messages, user requests, occur, and the system must respond to them. Basically, performance is concerned with how long it takes the system to respond when such an event occurs.

Security Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users. Security breaches ("attacks") can take several forms, ranging from attempts to access or modify data or services, to attempts to deny services to legitimate users.

Testability Software testability refers to the ease with which software can be made to demonstrate its faults through testing. In particular, testability refers to the probability (under the assumption that the software has at least one fault), that it will fail on its *next* test.

Usability Usability is concerned with how easy it is for a user to accomplish a desired task and the kind of user support the system provides. What can the system do to make the task of learning the system to a user (who is unfamiliar with the system) easier? What can the system do to make the user more efficient in its operation? What can the system do so that a user error has minimal impact? How can the user (or the system itself) adapt to make the user's task easier? What kind of feedback does the system give to help the user feel confident that the correct action is being taken?

The goals of creating a solid software architecture for a system are reduced development and maintenance cost (in terms of time and money) of the system, and achieving a higher quality of the software.

In this thesis, several chapters are related to *modifiability*. In particular, Chapter 5 deals with the question: How can an existing system be modified to use an off-the-shelf component? And the extensibility framework in Chapter 6 discusses techniques to make a software system more *modifiable* through the use of plug-ins.

1.1.2 Component-based software engineering

The insight that elements of software (routines) could be seen as components, and that these components could be constructed and possibly re-used in a way similar to hardware components already dates back to the late 60's. McIlroy [97] shared his early thoughts on how software components should be available in families arranged according to precision, robustness, generality and time/space performance.

A central problem in component-based software engineering is how to arrange the cooperation between different components. Components today are often available in different forms. For example, one component could come in the form of a software library which needs to be linked against the target application, which has local access to the component. Another component could be available in the form of a webservice. In this case, local requests are handled on a remote machine and some form of network connection is needed to transfer the request and its subsequent result. Also, components are usually implemented in different programming languages. They may differ in (numerical) precision, or in the level of security and robustness they offer. This real world component heterogeneity makes it hard to treat components as basic blocks the way McIlroy sketched. They simply are not as standardized as nuts and bolts are in civil engineering. Building up a system out of software components remains therefore quite a challenge.

From the many interesting issues related to component based software engineering, in this thesis we focus mainly on two specific ones. Namely those of (efficient) data exchange between heterogeneous components, and of coordinating the cooperation between components. In both areas we consider software maintainability to be very important. We are willing to accept a (minor) performance hit if we consider the resulting software to be better maintainable.

1.1.3 Middleware

Merely having a collection of components is not enough to create a working software system. Somehow all these components, whether they are available through re-use, or have been developed specifically for the project, have to be connected. This is the realm of *middleware* [18].

The term middleware is used in general to describe the software that connects various elements of software. The granularity of these elements can range from very small (e.g., objects in a object oriented language) to off-the-shelf components (e.g., a text editor). Also, the implementation language of the components connected by the middleware can vary.

Some examples of middleware are:

Remote Procedure Call (RPC) By using a RPC, a client-server relationship between components can be established. The client component can invoke specific methods in a server component which may be located on another computer. The idea behind RPC (network-based resource sharing) has been around since 1975 [126], but it is still available today, e.g., as `Java` RMI (Remote Method Invocation), and in Microsoft .NET.

Object Request Broker (ORB) A more general approach to distributed system design is to remove the distinction between a client requesting a service, and a server providing it. Instead, each object provides an interface to a set of services it provides. Other objects use these services via the middleware, which is called an object request broker. An example is CORBA [49]: the Common Object Request Broker Architecture.

Message Oriented Middleware (MOM) In RPC and ORB a synchronous request-response mechanism is used, meaning the client blocks and waits for the request to be completed. In MOM asynchronous communication is used between components. Thus, the sending component does not block waiting for the recipient to participate in the communication. If the middleware implements persistence and reliability, the recipient component need not even be up and running when the request is sent. Examples of MOM include IBM's WebSphere MQ [125], the ToolBus Coordination Architecture [13], and Manifold [95].

Structured database access (DBC) A database is often an important component in a software system. Database Connectivity middleware allows scalable, structured data access implementing object persistence and often allowing transactional models for reliable data storage. The Open Database Connectivity (ODBC) provides a standard software API for using database management systems. Sun Microsystems implements ODBC for `Java` (JDBC), and Microsoft designed OLE-DB as an API for accessing various types of data stores in a uniform manner as part of their Component Object Model (COM) architecture.

In this thesis, Chapter 4 addresses various message-based middleware issues focused around the ToolBus Coordination Architecture. The ToolBus is also used as middleware implementation in Chapter 5 to connect off-the-shelf components to an existing software system, and in the plug-in framework described in Chapter 6.

1.1.4 Software Product lines

Not just individual software components can be re-used, but we can also think about the re-use of an entire software architecture across a family of related systems [75]. The idea is that by reusing the same architecture (and elements associated with that architecture), substantial benefits can be enjoyed including a reduction in construction cost and in time to market [9]. We call this a *software product line*, which [46] defines as follows:

7

a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

As with components, product lines are also well known from industry. For example, all major aircraft, car, and computer manufacturers, as well as military industry all use product lines. Based on the same insight that there is such a thing as inter-product commonality which can be exploited, *software* product lines are now a common concept in software engineering as well. With many customers having their own requirements, flexibility on the part of manufacturers is a must. Here, software product lines can help to simplify the creation of systems for (groups of) customers. Several companies, e.g. Nokia, Motorola, and Hewlett-Packard are noted [9] to have experienced significant improvements in cost, time to market and productivity, after successfully using software product lines.

In a product line approach, it is obviously helpful if all components can deal with data (de-)serialization using the same base infrastructure. In this thesis, Chapters 2, and 3 play a role in the context of software product lines, as they enable structured data exchange between components using a central API to access the data. Also, Chapter 6 demonstrates how individual components can be brought together in an all-encompassing user interface. This relieves individual components of the burden of having its own full-blown user interface. It is also useful to be able to connect a range of similar components to the same base architecture, e.g., to facilitate testing these components in a homogeneous way. In Chapter 5 we show how various similar components, text editors in this case, can be used interchangeably in the same base system.

1.1.5   Variability

The software product line approach relies on strategic or planned, rather than opportunistic, re-use [9] of software. Because ultimately not a single, but multiple, though similar products are developed, *variability* of software components [6, 63] has to be taken into account when designing the architecture to be used in a product line setting.

In this thesis, Chapter 5 studies how a variation point can be added to an existing architecture, by removing the fixed text editing facilities in a development environment, and replacing it by a user choice between several off-the-shelf editors.

Chapter 6 describes how variation points can be implemented in a software architecture by means of plug-ins.

In a software product line, there will be several features which can be included or excluded in the various products. Obviously not all features will be able to co-exist. One feature may depend on another, some features might be mutually exclusive, etc. Dealing with large feature and constraint sets becomes a computational problem. Although the ATerm-Library (Chapter 2) by itself does not directly contribute to the domain of variability, its key feature of maximal subterm sharing can be used to keep the memory footprint of feature set representations down [54, 31].

1.2 The Decoupling Paradox

One way to look at a software system is to see it as a single "box" with some externally observable behavior. For example: "a web browser is an application in which a user enters the location of a web page, which is then displayed by the browser."

Another way to look at the same software system, is to recognize that the box is not a single computer program, but that it is in fact made up of several parts. These parts perform some functionality on their own, but they also interact with other parts, and together they form the application. The web browser may contain a user interface part used to enter the location of the desired web page, a download mechanism to fetch the page, a parser to interpret the contents of the page, and a rendering engine which displays the structured document on the user's screen.

As the system grows in complexity (e.g., caching functionality to speed up repeated viewings of the same web page, and security measures needed for electronic shopping are added to the browser), it becomes more and more important to keep all these concerns well separated. Failing to do so leads to a monolithic software system with tangled functionality.

In order to keep these concerns well separated, and to keep the tangling to a minimum, we need to *decouple* [102] the individual parts of the system. Preferably, each of these subcomponents is oblivious of the others, making it easier, for example, to replace the current implementation of security algorithms in our web browser, if a better one becomes available.

The act of *decoupling* two subcomponents in a software system immediately leads to the following question: *When two subcomponents which work together in a software system are decoupled, how will they work together in the new situation?*

Working together means at the very least that the components need to exchange information, and that one component can ask the other to perform some task. So in a way, in order to decouple components, we need to *couple* them again in another way. We call this the *Decoupling Paradox*.

Of course it is possible to come up with some *ad hoc* way to make two decoupled components work together in a way that the system as a whole still functions as it did before the decoupling. But what if the system contains tens or hundreds of components that need to be decoupled? We would then have to come up with numerous of those ad hoc solutions, resulting in a software system that is perhaps even less maintainable than the original, monolithic version.

1.2.1 The UNIX pipeline

The UNIX operating system is an example of a software system which focuses on strong decoupling of components. These components operate together via a standardized communication mechanism. Each component has three channels: *input*, *output*, and *error*. The basic connection operator is an invention by McIlroy from the 1970's, called a *pipe*, which links the output of one component as the input of the next. This architecture is now known as the *pipe-and-filter* architecture [9]. Figure 1.1 (from [127]) shows an example of three cooperating components (programs) running in a text terminal.

9
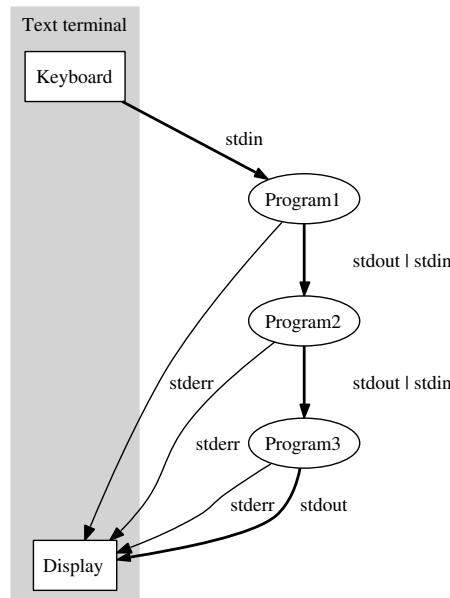
Figure 1.1: A pipeline of three programs run on a text terminal.

Note how each component has its *error* channel redirected to the GUI (text terminal). The components are decoupled, but still need to be coupled. One coupling to get the output of a component linked to the input of the next, and (in this case) one coupling per component to get its errors displayed on the terminal. Finally, by default, if the output of a component is not part of a pipeline to another component, it is also redirected to the terminal.

The data exchanged between components is standardized to be in text format. This means that each component pretty prints its output so that it is human readable in case the output is sent to the terminal. And each component parses any input itself. The only agreement is that data are exchanged in textual format. Other than that, nothing is fixed. Components thus depend on knowledge of the output format of other components.

The way individual components interact and cooperate is programmed by means of shell scripts. In its simplest form, such a script is nothing but a sequence of program invocations, separated by pipelines. For example, `cat /etc/passwd | wc -l` is a shell script that reads a file from the filesystem (in this case `/etc/passwd`) and feeds the output into a program `wc` which can count words, lines, and characters in the input. In this case it takes the `-l` parameter directing it to only output the line count. Effectively, this shell script counts the number of entries in the password file.

ToolBus

Coordination

Representation

ATerms   ATerms

Computation

Tools
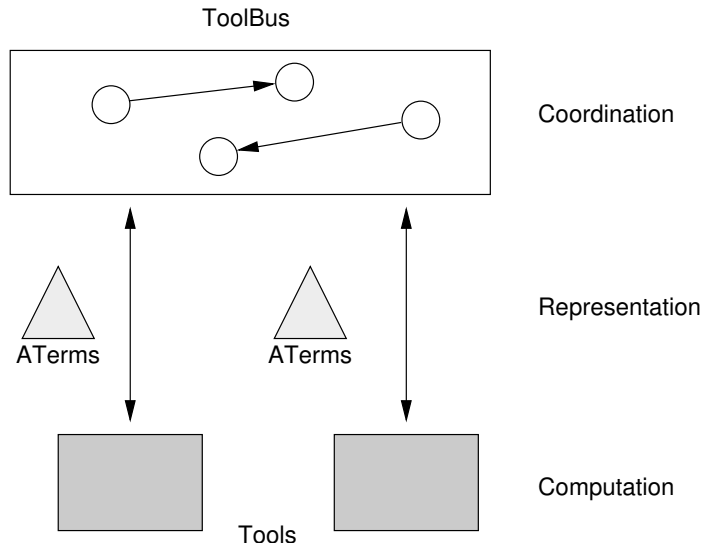
Figure 1.2: The ToolBus architecture

1.2.2   The ToolBus coordination architecture

In the ToolBus coordination architecture [13], a central, programmable software bus coordinates the interaction between connected components. Similar to the unix example above, it also promotes looking at components in a highly decoupled way. Where the unix pipeline system is based on (unstructured) textual data exchange, the ToolBus allows components to exchange *structured* information in the form of terms as well.

Each component in a ToolBus coordinated application uses a generic term library to encode any data it exchanges with other components. This term library is implemented in several programming languages to facilitate interoperability of tools written in various programming languages. Figure 1.2 shows how individual tools are connected to the ToolBus, exchanging data (in the form of ATerms, detailed in Chapter 2) via the ToolBus.

The way these components interact and cooperate is now programmed by means of a ToolBus script. The primitives available in the ToolBus language are based on process algebra, which are detailed in Chapter 4. In this example, we abstract from most of the ToolBus details, and use only the dot (.) operator for sequential composition.

```
process ReadFile is
let
  Filename: str,
  CatTool: cat,
  Contents: str
in
  rec-msg(read-file(Filename?))
```

11

```
   . execute(cat, CatTool?)
   . snd-eval(CatTool, read-file(Filename))
   . rec-value(CatTool, Contents?)
   . snd-msg(file-contents(Contents))
endlet

tool cat is { command = "/path/to/cat-tool" }
```

The ToolBus process `ReadFile` defines the behavior of the external tool `cat-tool`. After receiving the `read-file` message parameterized by a specific filename, the external tool is executed, and told to read the file. The `cat-tool` then passes the result to the ToolBus, where the `ReadFile` process propagates the result of reading the file. The last line in the example tells the ToolBus what the operating system command is to start the tool. In this case it only mentions the location of our tool on the filesystem.

In a similar way we can write a ToolBus process for a tool capable of counting the number of lines in a given string. For this example, we will just assume it exists and that it can be invoked by sending the message `count-lines` and receiving the number of lines via a message `line-count`. With these two processes, we can now mimic our example from Section 1.2.1 which prints out the number of entries in the password file.

```
process CountPasswordEntries is
let
  Contents: str,
  Lines: int
in
  snd-msg(read-file("/etc/passwd"))
  . rec-msg(file-contents(Contents?))
  . snd-msg(count-lines(Contents))
  . rec-msg(line-count(LineCount?))
  . printf("/etc/passwd has %d entries.\n", LineCount)
endlet
```

This process first sends a message via the ToolBus with the request to read the file `/etc/passwd`, handled by the `ReadFile` process. After receiving the contents of the file, it sends out another message requesting a line count of the contents. The result is then printed to the terminal.

1.3   Research Context

The primary case study used to verify and validate the research described in this thesis is the ASF+SDF Meta-Environment. The ASF+SDF Meta-Environment [86] is an interactive development environment for constructing language definitions and generating tools for them. A language definition typically includes a specification for its syntax, as well as for type checking, pretty printing and executing programs in the target language.

The first implementation of the ASF+SDF Meta-Environment was written in LELISP. After several years the application had developed into an amorphous blob of functionality, too inflexible to be properly used for further development. For example, an experiment to externalize the GUI and editor [8] failed due to deadlock issues in the system which were too complex to be solved.

In order to solve these problems, a reimplementation of the ASF+SDF Meta-Environment was started, this time designed to be a compound system of individual functional components, plus a central component coordination architecture to coordinate interaction between the components. To facilitate this, the ToolBus Coordination Architecture [11] was developed. By identifying chunks of functionality in the original application, and isolating them into individual components, a much more open and untangled application results. Each individual component is connected to the Tool-Bus. The interaction between components is described in the ToolBus in a formal way by means of a scripting language which is based on process algebra [7]. The Tool-Bus architecture advocates strict separation between *computation* which is done inside components and *coordination* between components which is performed in the ToolBus itself.

Splitting up a software system such as the Meta-Environment, an interactive software development system, into independent components helps to untangle the software system as a whole, but it also introduces new challenges. First of all, an investment is needed for external components to adapt them to the specific coordination environment used for the application. In particular, we wanted to use existing, off-the-shelf text editors such as GNU Emacs and Vim to implement the main text editing facilities of the Meta-Environment. Of course, when a new version of the external component is released, we want as little maintenance as possible, preferably none at all, on our part to keep using that component. This raises the question how external components can be used in a component coordination driven application, without big investments or high maintenance cost per component.

When components are decoupled, we can coordinate their cooperation in an entirely decoupled way, but *human users* of the system will prefer to interact with it in a centralized way. They want a single graphical user interface, not one separate window per component. The Meta-Environment project also ran into this conflict of interests where we want to have decentralized components, but with a centralized GUI. The GUI was a separate tool, but as new functionality requiring user interaction was added to the system, the interface of the GUI component grew bigger and bigger. This led us to investigate how we could come up with a more modular user interface architecture which itself has a fixed interface to the system, but with the ability to host plug-ins for the various features of the system that require user interaction.

1.4   Research Questions

The work in this thesis is structured around two research questions which are both related to the (re)structuring of the architecture of an interactive software development environment.

With the introduction of a component coordination architecture, and its accompanying software engineering ideas to identify and separate functionality from a monolithic program into individual components, an obvious question is how to facilitate data exchange between the components:

> **Research Question 1:** *How can structured data be exchanged between heterogeneous components in a space efficient, type-safe way?*

Another interesting study is to look at the impact of applying component coordination techniques to the redesign and implementation of an interactive application. With portions of functionality now isolated into individual components, how do we describe the functionality of the application as a whole? How does this affect central user interaction, what does it mean for the software developer? In general, we are interested in the following question:

> **Research Question 2:** *What are the implications of using component coordination techniques on the architecture of interactive software development environments?*

1.5   Related Work

There are several sections discussing related work in this thesis. Most are local discussions about work related to that in the chapter itself. In Section 2.6.1 we give an overview of some of the work related to intermediate representations of tree-like data structures like ATerms.

In Section 3.1.1 we give an overview of techniques similar to our generation of a type-safe access layer on top of a generic, untyped data representation layer.

In the context of possible improvements to the current ToolBus architecture, we briefly discuss several other remote method invocation frameworks in Section 4.5.4. We also relate the call-by-value model currently used in the ToolBus, to models used in other architectures.

In Section 5.1.2 we relate our approach to integrating off-the-shelf components into a ToolBus coordinated IDE to similar projects. Our approach focuses on *functional* integration, where other projects often achieve embedding, or *visual* integration.

In Chapter 6 we present a comprehensive overview of contemporary projects using a plug-in mechanism to achieve some form of application extensibility We categorize software system extensibility patterns found in these systems, and relate them to our approaches in ToolBus based applications.

1.6   Outline and Origin of the Chapters

This thesis is divided into two parts, each dealing with one of the research questions. Each part consists mostly of previously published chapters which are self-contained, and can be read independently.

In the first part of this thesis we study the lower level data management question. Chapter 2 discusses the design and implementation of a generic, space efficient data type for the exchange of structured data between heterogeneous components. Chapter 3 shows how an access layer can be generated that allows developers type-safe access to the untyped generic data layer.

In the second part, we try to answer the higher level architectural question by means of a series of case studies. In Chapter 4 we reflect on our coordination architecture itself. In Chapter 5 we study how existing third party components can be deployed in a component coordinated architecture in an off-the-shelf way. Finally, Chapter 6 studies how to have a centralized graphical user interface in an otherwise decentralized, component based architecture.

For each of the chapters, the following list describes their origin, respective co-authors, and acknowledgments.

Chapter 2: Efficient Annotated Terms was published in Software, Practice and Experience [30] and is joint work with P. Olivier, M.G.J. van den Brand, and P. Klint.

Chapter 3: API Generation from Syntax Definitions was published in The Journal of Logic and Algebraic Programming [79] and is joint work with P. Olivier.

Chapter 4: ToolBus: the Next Generation was presented in Formal Methods for Components and Objects [77] and is joint work with P. Klint.

Chapter 5: My Favorite Editor Anywhere was presented at RISE 2004: First International Workshop on the Rapid Integration of Software Engineering Techniques [78] and is joint work with A.T. Kooiker.

Chapter 6: Software System Extensibility has not yet been submitted for publication. The overview and the initial ideas for our plug-in architecture were contributed by me, the implementation and application in the Meta-Environment is joint work with A.T. Kooiker.

1.7   About the Implementations

The work presented in this thesis is supported by a substantial implementation effort. Initially written with the ASF+SDF Meta-Environment in mind, the software developed during my research has found several uses outside our research group as well. Throughout the six years of research, development and maintenance of this software, the focus of my contribution has shifted from implementation centric to a more architectural one. The openness and simplicity of the implementations are a key contribution, allowing future research to build on the fruits of our research. Implementing a "real" version of an idea (as opposed to stopping at the proof-of-concept level) and using it in your own software and having it used by other researchers immediately both validates and challenges the foundations of the initial idea.

15

The ATerm library (Chapter 2, [30])   The ATerm implementation has been a joint effort of the author of this thesis with Pieter Olivier. The library we initially developed consisted of roughly 12,000 lines of C code and 5,000 lines of `Java` code. The `Java` version was later improved by Pierre-Etienne Moreau, who factored out the maximal subterm sharing concept into a library for sharing any `Java` object (not necessarily ATerms). The original mark-and-sweep garbage collector implemented by Pieter Olivier and me was later replaced by a generational garbage collector developed by Moreau and Zendra [99], resulting in an observed efficiency gain of 20-35%. In [31], an overview can be found of various current applications of the ATerm library.

The type-safe access API generator ApiGen (Chapter 3, [79])   The ApiGen implementation was also a joint effort of Pieter Olivier and me. The C code generator was written in about 1,400 lines of `Java` code. The proof-of-concept `Java` code generator was about 1,000 lines of `Java` code. Both implementations use an ASF+SDF specification of 275 lines of code to translate SDF specifications to the intermediate format used by the generators. The `Java` generator was later improved significantly by Pierre-Etienne Moreau. An overview of projects using ApiGen can again be found in [31].

The ToolBus Next Generation (Chapter 4, [77])   studies have resulted in a re-implementation in `Java` of the original ToolBus (written in C). The new `Java` version currently counts about 11,000 lines of code, and (like the C version) was implemented by Paul Klint. He is currently working towards a first release of the `Java` version to replace the C ToolBus currently in use by the Meta-Environment.

The support for using third-party editors (Chapter 5, [78])   was developed by Taeke Kooiker and me. The core functionality consists of about 2,500 lines of handwritten code, and a C library generated by ApiGen for type-safe access on the data exchanged between the core and the individual editor instances. An 80 line ToolBus script describes how other ToolBus processes can use the editing functionality. The third party editors were in use in the Meta-Environment for several years.

The plug-in architecture (Chapter 6)   was first implemented by me in about 500 lines of `Java` proof-of-concept code. Together with Taeke Kooiker the version currently in use in the Meta-Environment was developed. The plug-in framework now consists of about 1,500 lines of `Java` code and 50 lines of ToolBus code. The migration of existing user interface code from the monolithic Meta-Environment GUI into individual plug-ins was a joint effort by me and Taeke Kooiker. At the same time new plug-ins were added by Taeke Kooiker and Jurgen Vinju.

Part II

Structuring Component Data

C H A P T E R 2

ATerms

2.1 Introduction

Cut and paste operations on complex data structures are standard in most desktop soft-ware environments: one can easily clip a part of a spreadsheet and paste it into a text document. The exchange of complex data is also common in distributed applications: complex queries, transaction records, and more complex data are exchanged between different parts of a distributed application. Compilers and programming environments consist of tools such as editors, parsers, optimizers, and code generators that exchange syntax trees, intermediate code, and the like.

How is this exchange of complex data structures between applications achieved? One solution is Microsoft's Object Linking and Embedding (OLE) [45]. This is a platform-specific, proprietary, set of primitives to construct Windows applications. An-other, language-specific, solution is to use Java's serialization interface [66]. This al-lows writing and reading Java objects as sequential byte streams. Yet another solu-tion is to use OMG's Interface Definition Language (part of the Common Object Bro-ker Architecture [107]) to define data structures in a language-neutral way. Specific language-bindings provide the mapping from IDL data structures to language-specific data structures.

All these solutions have their merits but do not really qualify when looking for an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of complex data structures between distributed applications. To be more specific, we are interested in a solution with the following characteristics:

Open: independent of any specific hardware or software platform.

Simple: the procedural interface should contain 10 rather than 100 functions.

Efficient: operations on data structures should be fast.

Concise: inside an application the storage of data structures should be as small as possible by using compact representations and by exploiting sharing. Between applications the transmission of data structures should be fast by using a com-pressed representation with fast encoding and decoding. Transmission should preserve any sharing of in-memory representation in the data structures.

19

Language-independent: data structures can be created and manipulated in any suitable programming language.

Annotations: applications can transparently extend the main data structures with annotations of their own to represent non-structural information.

In this chapter we describe the data type of *Annotated Terms*, or just *ATerms*, that have the above characteristics. They form a solution for our implementation needs in the areas of interactive programming environments [86, 35] and distributed applications [15] but are more widely applicable. Typically, we want to exchange and process tree-like data structures such as parse trees, abstract syntax trees, parse tables, generated code, and formatted source texts. The applications involved include parsers, type checkers, compilers, formatters, syntax-directed editors, and user-interfaces written in a variety of languages. Typically, a parser may add annotations to nodes in the tree describing the coordinates of their corresponding source text and a formatter may add font or color information to be used by an editor when displaying the textual representation of the tree.

The ATerm data type has been designed to represent such tree-like data structures and it is therefore very natural to use ATerms both for the internal representation of data inside an application and for the exchange of information between applications. Besides function applications that are needed to represent the basic tree structure, a small number of other primitives are provided to make the ATerm data type more generally applicable. These include integer constants, real number constants, binary large data objects ("blobs"), lists of ATerms, and placeholders to represent typed gaps in ATerms. Using the comprehensive set of primitives and operations on ATerms, it is possible to perform operations on an ATerm received from another application without first converting it to an application-specific representation.

First, we will give a quick overview of ATerms (Section 2.2). Next, we discuss implementation issues (Section 2.3) and give some insight in performance issues (Section 2.4). An overview of applications (Section 2.5) and an overview of related work and a discussion (Section 2.6) conclude this chapter.

2.2   ATerms at a Glance

We now describe the constructors of the ATerm data type (Section 2.2.1) and the operations defined on it (Section 2.2.2).

2.2.1   The ATerm Data Type

The data type of ATerms (`ATerm`) is defined as follows:

- `INT`: An integer constant (32-bits integer) is an ATerm.[1]

- `REAL`: A real constant (64-bits real) is an ATerm.

---

[1] We are currently upgrading the ATerm library to support 64-bit architectures as well.

- APPL: A function application consisting of a function symbol and zero or more ATerms (arguments) is an ATerm. The number of arguments of the function is called the *arity* of the function.

- LIST: A list of zero or more ATerms is an ATerm.

- PLACEHOLDER: A placeholder term containing an ATerm representing the type of the placeholder is an ATerm.

- BLOB: A "blob" (Binary Large data OBject) containing a length indication and a byte array of arbitrary (possibly very large) binary data is an ATerm.

- A list of ATerm pairs may be associated with every ATerm representing a list of (*label*, *annotation*) pairs.

Each of these constructs except the last one (i.e., INT, REAL, APPL, LIST, PLACEHOLDER, and BLOB) form subtypes of the data type ATerm. These subtypes are needed when determining the type of an arbitrary ATerm. Depending on the actual implementation language they will be represented as a constant (C, Pascal) or a subclass (C++, Java).

The last construct is the *annotation construct*, which makes it possible to annotate terms with transparent information[2].

Appendix A.1 contains a definition of the concrete syntax of ATerms. The primary reason for having a concrete syntax is to be able to exchange ATerms in a human-readable form. In Section 2.3 we also discuss a compact binary format for the exchange of ATerms in a format that is only suitable for processing by machine. We will now give a number of examples to show some of the features of the textual representation of ATerms.

- Integer and real constants are written conventionally: 1, 3.14, and -0.7E34 are all valid ATerms.

- Function applications are represented by a function name followed by an open parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: f(a,b) and "test!"(1,2.1,"Hello world!"). These examples show that double quotes can be used to delimit function names that are not identifiers.

- Lists are represented by an opening square bracket, a number of list elements separated by commas and a closing square bracket: [1,2,"abc"], [], and [f,g([1,2]),x] are examples.

- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are <int>, <[3]>, and <f(<int>,<real>)>.

---

[2]Transparent in the sense that the result of most operations is independent of the annotations. This makes it easy to completely ignore annotations. Examples of the use of annotations include annotating parse trees with positional or typesetting information, and annotating abstract syntax trees with the results of type checking.

- Blobs do not have a concrete syntax because their human-readable form depends on the actual blob content.

2.2.2 Operations on ATerms

The operations on ATerms fall into three categories: making and matching ATerms (Section 2.2.2), reading and writing ATerms (Section 2.2.2), and annotating ATerms (Section 2.2.2). The total of only 13 functions provide enough functionality for most users to build simple applications with ATerms. We refer to this interface as the *level one* interface of the ATerm data type.

To accommodate "power" users of ATerms we also provide a *level two* interface, which contains a more sophisticated set of data types and functions. It is typically used in generated C code that calls ATerm primitives, or in efficiency-critical applications. These extensions are useful only when more control over the underlying implementation is needed or in situations where some operations that can be implemented using level one constructs can be expressed more concisely and implemented more efficiently using level two constructs. The level two interface is a strict superset of the level one interface (see Appendix A.2 for further details).

Observe that ATerms are a purely functional data type and that no destructive updates are possible, see Section 2.3.2 for more details.

Making and Matching ATerms

The simplicity of the level one interface is achieved by the *make-and-match* paradigm:

- *make* (compose) a new ATerm by providing a pattern for it and filling in the holes in the pattern.

- *match* (decompose) an existing ATerm by comparing it with a pattern and decompose it according to this pattern.

Patterns are just ATerms containing placeholders. These placeholders determine the places where ATerms must be substituted or matched. An example of a pattern is `"and(<int>,<appl>)"`. These patterns appear as string argument of both make and match and are remotely comparable to the format strings in the `printf/scanf` functions in C. The operations for making and matching ATerms are:

- `ATerm ATmake(String` p`, ATerm` a_1`, ..., ATerm` a_n`)`: Create a new term by taking the string pattern p, parsing it as an ATerm and filling the placeholders in the resulting term with values taken from a_1 through a_n. If the parse fails, a message is printed and the program is aborted. The types of the arguments depend on the specific placeholders used in *pattern*. For instance, when the placeholder `<int>` is used an integer is expected as argument and a new integer ATerm is constructed.

- `ATbool ATmatch(ATerm` t`, String` p`, ATerm` $\star a_1$`, ..., ATerm` $\star a_n$`)`:

22

Match term t against pattern p, and bind subterms that match with placeholders in p with the result variables a_1 through a_n. Again, the type of the result variables depend on the placeholders used. If the parse of pattern p fails, a message is printed and the program is aborted. If the term itself contains placeholders these may occur in the resulting substitutions. The function returns `true` when the match succeeds, `false` otherwise.

- `Boolean ATisEqual(ATerm` t_1`, ATerm` t_2`)`: Check whether two ATerms are equal. The annotations of t_1 and t_2 must be equal as well.

- `Integer ATgetType(ATerm` t`)`: Retrieves the type of an ATerm. This operation returns one of the subtypes mentioned before in Section 2.2.1.

Reading and Writing ATerms

For reasons of efficiency and conciseness, reading and writing can take place in two forms: text and binary. The text format uses the textual representation discussed earlier in Section 2.2.1 and Appendix A.1. This format is human-readable, space-inefficient[3], and any sharing of the in-memory representation of terms is lost.

The binary format (Binary ATerm Format, see Section 2.3.5) is portable, machine-readable, very compact, and preserves all in-memory sharing. The operations for reading and writing ATerms are:

- `ATerm ATreadFromString(String` s`)`: Creates a new term by parsing the string s. When a parse error occurs, a message is printed, and a special error value is returned.

- `ATerm ATreadFromTextFile(File` f`)`: Creates a new term by parsing the data from file f. Again, parse errors result in a message being printed and an error value being returned.

- `ATerm ATreadFromBinaryFile(File` f`)`: Creates a new term by reading a binary representation from file f.

- `Boolean ATwriteToTextFile(ATerm` t`, File` f`)`: Write the text representation of term t to file f. Returns `true` for success and `false` for failure.

- `Boolean ATwriteToBinaryFile(ATerm` t`, File` f`)`: Write a binary representation of term t to file f. Returns `true` for success, and `false` for failure.

- `String ATwriteToString(ATerm` t`)`: Return the text representation of term t as a string.

Either format (textual or binary) can be used on any linear stream, including files, sockets, pipes, etc.

---

[3] The unnecessary size explosion could be avoided by extending the textual representation with a mechanism for labeling and referring to terms. Instead of `f(g(a),g(a))`, one could then write `f(1:g(a), #1)`. The first occurrence of `g(a)` is labeled with "`1`", and the second occurrence refers to this label ("`#1`").

Annotating ATerms

Annotations are ($label, annotation$) pairs that may be attached to an ATerm. Recall that ATerms are a completely functional data type and that no destructive updates are possible. This is evident in the following operations for manipulating annotations:

- `ATerm ATsetAnnotation(ATerm t, ATerm l, ATerm a)`: Return a copy of term t in which the annotation labeled with l has been changed into a. If t does not have an annotation with the specified label, it is added.

- `ATerm ATgetAnnotation(ATerm t, ATerm l)`: Retrieve the annotation labeled with l from term t. If t does not have an annotation with the specified label, a special error value is returned.

- `ATerm ATremoveAnnotation(ATerm t, ATerm l)`: Return a copy of term t from which the annotation labeled with l has been removed. If t does not have an annotation with the specified label, it is returned unchanged.

2.3 Implementation

2.3.1 Requirements

In Section 2.1 we have already mentioned our main requirements: openness, simplicity, efficiency, conciseness, language-independence, and capable of dealing with annotations. There are a number of other issues to consider that have a great impact on the implementation, and that make this a fairly unique problem:

- By providing automatic garbage collection ATerm users do not need to deallocate ATerm objects explicitly. This is safe and simple (for the user).

- The expected lifetime of terms in most applications is very short. This means that garbage collection must be fast and should touch a minimal amount of memory locations to improve caching and paging performance.

- The total memory requirements of an application cannot be estimated in advance. It must be possible to allocate more memory incrementally.

- Most applications exhibit a high level of redundancy in the terms being processed. Large terms often have a significant number of identical subterms. Intuitively this can be explained from the fact that most applications process terms with a fixed signature and a limited tree depth. When the amount of terms that is being processed increases, it is plausible that the similarity between terms also increases.

- In typical applications less than 0.1 percent of all terms have an arity higher than 5.

- Many applications will use annotations only sparingly. The implementation should not impose a penalty on applications that do not use them.

- In order to have a portable yet efficient implementation, the implementation language will be C. This poses some special requirements on the garbage collection strategy[4].

With these considerations in mind, we will now discuss maximal (in-memory) sharing of terms (Section 2.3.2), garbage collection (Section 2.3.3), the encoding of terms (Section 2.3.4), and the Binary ATerm Format (Section 2.3.5).

2.3.2   Maximal Sharing

Our strategy to minimize memory usage is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term is reused, ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be built and leads to maximal sharing of subterms. The library functions that construct terms make sure that shared terms are returned whenever possible. The sharing of terms is thus invisible to the library user.

The Effects of Maximal Sharing

Maximal sharing of terms can only be maintained when we check at every term creation whether a particular term already exists or not. This check implies a search through all existing terms but must be fast in order not to impose an unacceptable penalty on term creation. Using a hash function that depends on the internal code of the function symbol and the addresses of its arguments, we can quickly search for a function application before creating it. The terms are stored in a hash table. The hash table does not contain the terms themselves, but pointers to the terms. This provides a flexible mechanism of resizing the table and ensures that all entries in the table are of equal size. Hence the (modest but not negligible) cost at term creation time is one hash table lookup.

Fortunately, we get two returns on this investment. First, the considerably reduced memory usage also leads to reduced execution time. Second, we gain substantially as the equality check on terms (`ATisEqual`) becomes very cheap: it reduces from an operation that is linear in the number of subterms to be compared to a constant operation (pointer equality).

Another consequence of our approach is less fortunate. Because terms can be shared without their creator knowing it, terms cannot be modified without creating unwanted side-effects. This means that terms effectively become *immutable* after creation. Destructive updates on maximally shared terms are not allowed. Especially in list operations, the fact that ATerms are immutable can be expensive. It is often the responsibility of the user of the library to choose algorithms that minimize the effect of this shortcoming.

---

[4]We have implemented the library in Java as well. In this case, many of the issues we discuss in this chapter are irrelevant, either because we can use built-in features of Java (garbage collection), or because we just cannot express these low level concerns in Java.

Searching for Shared Subterms

Maximal sharing of terms requires checking at term creation time whether this term
already exists. This search must be fast in order to ensure efficient term creation. A
hash function based on the *addresses* of the function symbol and the arguments of
a function application allows for a quick lookup in the hash table to find a function
application before creating it.

Collisions   One issue in hash techniques is handling collisions. The simplest tech-
nique is linear chaining [88]. This requires one pointer in each object for hash chaining,
which in our implementation implies a memory overhead of about 25 percent. Other
solutions for collision resolution will either increase the memory requirements, or the
time needed for insertions or deletions (see [88]). We therefore use linear hash chaining
in our implementation.

Direct or Indirect Hashing   Another issue is whether to store all terms directly in
the hash table, or only references. Storing the objects directly in the hash table saves
a memory access when retrieving a term as well as the space needed to store the refer-
ence. However, there are severe drawbacks to this approach:

- We cannot rehash old terms because rehashing means that we have to move the
  objects in memory. When using C as an implementation language, moving ob-
  jects in memory is not allowed because we can only determine a conservative
  root set and therefore are not allowed to change the pointers to roots. This would
  mean that the hash table could not grow beyond its initial size.

- Internal fragmentation is increased, because empty slots in the hash table are as
  large as the object instead of only one machine word.

- We would need a separate hash table for each term size to decrease the internal
  fragmentation.

Because of these problems, we use linear hash chaining combined with indirect
hashing. When the load of the hash table reaches a certain threshold, we rehash into a
larger table.

The user can increase the initial size of the hash table to save on resizing and rehash-
ing operations. The ATerm library provides facilities for defining hash tables as well.
This allows the implementation of a fast lookup mechanism for ATerms. User-defined
hash tables are used, for instance, to implement memo-functions in the ASF+SDF to
C compiler (see Section 2.5.3).

2.3.3   Garbage Collection

Which Technique?

The most common strategies for automatic recycling of unused space are reference
counting, mark-compact collection, and mark-sweep collection. In our case, reference
counting is not a valid alternative, because it takes too much time and space and is very

hard to implement in C. Mark compact garbage collection is also unattractive because it assumes that objects can be relocated. This is not the case in C where we cannot identify *all* references to an object. We can only determine the root set conservatively which is good enough for mark-sweep collection discussed below, but not for mark-compact collection.

Mark-sweep Garbage Collection   Mark-sweep garbage collection works using three phases. In the first phase, all objects on the heap are marked as 'dead'. In the second phase, all objects reachable from the known set of root objects are marked as 'live'. In the third phase, all 'dead' objects are swept into a list of free objects.

Mark-sweep garbage collection can be implemented in C efficiently, and without support from the programmer or compiler [21, 20]. Mark-sweep collection is more efficient, both in time and space than reference counting [76]. A possible drawback is increased memory fragmentation compared to mark-compact collection. The typical space overhead for a mark-sweep garbage collection algorithm is only 1 bit per object, whereas a reference count field would take at least three or four bytes.

Reusing an Existing Garbage Collector

A number of excellent generic garbage collectors for C are freely available, so why do we not reuse an existing implementation?

We have examined a number of alternatives, but none of them fit our needs. The Boehm-Weiser garbage collector [21] came close, but we face a number of unusual circumstances that render existing garbage collectors impractical:

- The hash table always contains references to all objects. It must be possible to instruct the garbage collector not to scan this area for roots.

- After an object becomes garbage, it must also be removed from the hash table. This means that we need very low level control over the garbage collector.

- The ATerm data type has some special characteristics that can be exploited to dramatically increase performance:

  - Destructive updates are not allowed. In garbage collection terminology, this means that there are no pointers from old objects to younger objects. Although we do not exploit it in the current implementation, this characteristic makes the use of a *generational* garbage collector very attractive.

  - The majority of objects have an in-memory representation of 8, 12, or 16 bytes.

  - Practical experience has shown that not many root pointers are kept in static variables or on the generic C heap. Performance can be increased dramatically if we eliminate the expensive scan through the heap and the static data area for root pointers. The only downside is that we require the programmer to explicitly supply the set of roots that is located on the heap or in static variables.

These observations allow us to gain efficiency on several levels, using everything from low level system 'hacks' to high-level optimizations.

Implementing the Garbage Collector

Considering both performance and the maintainability of the code that uses the ATerm library, we have opted for a version of the mark-sweep garbage collector. Every object contains a single bit used by the mark-sweep algorithm to indicate 'live' (marked) objects. At the start of a garbage collection cycle, all objects are unmarked. The garbage collector tries to locate and mark all live objects by traversing all terms that are explicitly protected by the programmer (using the `ATprotect` function), and by scanning the C run-time stack looking for words that could be references to objects. When such a word is found, the object (and the transitive closure of all of the objects it refers to) are marked as 'live'.

This scan of the run-time stack causes all objects referenced from local variables to be protected from being garbage collected. Our garbage collector is a conservative collector in the sense that some of the words on the stack could accidentally have the same bit pattern as object references. Because there is no way to separate these 'fake' bit patterns from 'real' object references, this can cause objects to be marked as 'live' when these are actually garbage. Note that bit patterns on the stack that do not point to valid objects are not traversed at all. Only when a bit pattern represents an address that is a valid object address it is followed to mark the corresponding object.

When all live objects are marked, a single sweep through the heap is used to store all objects that are free in separate lists of free objects, one list for each object size.

As we shall see in Section 2.3.4, most objects consist of only a couple of machine words. By restricting the maximum arity of a function, we can also set an upper bound on the maximum size of objects. This enables us to base the memory management algorithms we use on a small number of block sizes. Allocation of objects is now simply a matter of taking the first element from the appropriate free-list, which is an extremely cheap operation. If garbage collection does not yield enough free objects, new memory blocks will be allocated to satisfy allocation requests.

2.3.4   Term Encoding

An important issue in the implementation of ATerms is how to represent this data type so that all operations can be performed efficiently in time and space.

The very concise encoding of ATerms we use is as follows. Assume that one machine word consists of four bytes. Every ATerm object is stored in two or more machine words. The first byte of the first word is called the *header* of the object, and consists of four fields (see Figure 2.1):

- A field consisting of one bit used as a mark flag by the garbage collector.

- A field consisting of one bit indicating whether or not this term has an annotation.

- A field consisting of three bits that indicate the type of the term.

Figure 2.1: The header layout

- A field consisting of three bits representing the arity (number of pointers to other terms) of this object. When this field contains the maximum value of 7, the term must be a function application and the actual arity can be found by retrieving the arity of the function symbol (see below).

Depending on the type of the node (as determined by the header byte in the first word) the remaining bytes in the first word contain either a function symbol, a length indication, or they are unused.

The *second* word is always used for hashing, and links together all terms in the same hash bucket.

The type of the node determines its exact layout and contents. Figure 2.2 shows the encoding of the different term types which we will now describe in more detail.

INT encoding   In an integer term, the third word contains the integer value. The arity of an integer term is 0.

REAL encoding   In an real term, the third and fourth word contain the real value represented by an 8 byte IEEE floating point number. The arity of a real term is 0.

APPL encoding   The remaining 3 bytes following the header in the first word are used to represent the index in a table containing the function symbols. The words following the second word contain references to the function arguments. In this way, function applications can be encoded in $2 + n$ machine words, with n the arity of the function application.

LIST encoding   The binary list constructor can be seen as a special function application with no function symbol and an arity of 2. The third word points to the first element in the list, this is called the `first` field, the fourth word points to the remainder of the list, and is called the `next` field. The length of the list is stored in the three bytes after the header in the first word. The empty list[5] is represented using a LIST object with empty first and next fields, and a length of 0.

After the function application, the list construct is the second most used ATerm construct. A (memory) efficient representation of lists is therefore very important. Due to the nature of the operations on ATerm lists, there are two obvious list representations: an array of term references or a linked list of term references. Experiments have shown that in typical applications quite varying list sizes are encountered. This renders the

---

[5]Due to the uniqueness of terms, only one instance of the empty list is present at any time.

byte	0	1		4	

INT

Header	Unused	0
Next hash entry		1
Integer value		2
Optional Annotation		3

REAL

Header	Unused	0
Next hash entry		1
Real value		2
		3
Optional Annotation		4

APPL

Header	Function symbol	0
Next hash entry		1
Argument 0		2
Argument 1		3
Argument n		n+1
Optional Annotation		n+2

LIST

Header	Length	0
Next hash entry		1
First		2
Next		3
Optional Annotation		4

PLACEHOLDER

Header	Unused	0
Next hash entry		1
Placeholder type		2
Optional Annotation		3

BLOB

Header	Data size	0
Next hash entry		1
Data pointer		2
Optional Annotation		3

Figure 2.2: Encoding of the different term types

array approach inferior, because adding and deleting elements of a list would become too expensive. Consequently, we have opted for the linked list approach. Lists are constructed using binary list constructors, containing a reference to the first element in the list and to the tail of the list. Each list operation must ensure that the list is "normalized" again. This makes it very easy to perform the most commonly used operations on list, namely adding or removing the first element of a list.

Other operations are more expensive, since we do not allow destructive updates. Adding an element to the tail of a list for instance, requires n list creation operations, where n is the number of elements in the newly created list.

PLACEHOLDER encoding   The placeholder term has an arity of 1, where the third word contains a pointer to the placeholder type.

BLOB encoding   The length of the data contained in a BLOB term is stored in the three bytes after the header. This means that up to 16,777,200 bytes can be encoded in a single BLOB term. A pointer to the actual data is stored in the third word.

Annotations   In all cases, annotations are represented using an extra word at the end of the term object. The single annotation bit in the header indicates whether or not an annotation is present. Only when this bit is set, an extra word is allocated that points to a term with type `LIST`, which represents the list of annotations.

2.3.5   ATerm Exchange: the Binary ATerm Format

The efficient exchange of ATerms between tools is very important. The simplest form of exchange is based on the concrete syntax presented in Appendix A.1. This would involve printing the term on one side and parsing it on the other. The concrete syntax is not a very efficient exchange format however, because the sharing of function symbols and subterms cannot be expressed in this way.

A better solution would be to exchange a representation in which sharing (both of function symbols and subterms) can be expressed concisely. A raw memory dump cannot be used, because addresses in the address space of one process have no meaning in the address space of another process.

In order to address these problems, we have developed BAF, the **B**inary **A**Term **F**ormat. Instead of writing addresses, we assign a unique number (index) to each subterm and each symbol occurring in a term that we want to exchange. When referring to this term, we could use its index instead of its address.

When writing a term, we begin by writing a table (in order of increasing indices) of all function symbols used in this term. Each function symbol consists of the string representation of its name followed by its arity.

ATerms are written in prefix order. To write a function application, first the index of the function symbol is written. Then the indices of the arguments are written. When an argument consists of a term that has not been written yet, the index of the argument is first written itself before continuing with the next argument. In this way, every subterm is written exactly once. Every time a parent term wishes to refer to a subterm, it just uses the subterm's index.

Exploiting ATerm Regularities

When sending a large term containing many subterms, the subterm indices can become quite large. Consequently many bits are needed to represent these indices. We can considerably reduce the size of these indices when we take into account some of the regularities in the structure of terms. Empirical study shows that the set of function symbols that can actually occur at each of the argument positions of a function application with a given function symbol is often very small. A explanation for this is that although ATerm applications themselves are not typed, the data types they represent often are. In this case, function applications represent objects and the type of the object is represented by the function symbol. The type hierarchy determines which types can occur at each position in the object.

We exploit this knowledge by grouping all terms according to their top function symbol. Terms that are not function applications are grouped based on dummy function symbols, one for each term type. For each function symbol, we determine which function symbols can occur at each argument position. When writing the table of function symbols at the start of the BAF file, we write this information as well. In most cases this number of function symbol occurrences is very small compared to the number of terms that is to be written. Storing some extra information for every function symbol in order to get better compression is therefore worthwhile.

When writing the argument of a function application, we start by writing the actual symbol of the argument. Because this symbol is taken from a limited set of function symbols (only those symbols that can actually occur at this position), we can use a very small number to represent it. Following this function symbol we write the index of the argument term itself in the table of terms over this function symbol instead of the index of the argument in the total term table.

Example

As an example, we show how the term `mult(s(s(z)),s(z))` is represented in BAF. This term contains three function symbols: `mult` with arity two, `s` with arity one, and `z` with arity zero. When grouping the subterms by function symbol we get:

0: `mult`	1: `s`	2: `z`
`mult(s(s(z)),s(z))`	`s(s(z))`	`z`
	`s(z)`	

When we look at the function symbols that can occur at every argument position (≥ 0) we get:

position	`mult`	`s`	`z`
0	`s`	`s, z`	
1	`s`		

We start by writing this symbol information to file. To do this, we have to write the following bytes[6]:

4 "mult" : The length (4) and ASCII representation of `mult`.
2   : The arity (2) of `mult`.
1 1   : There is only one symbol (1) that can occur at the first argument
                 position of `mult`. This is symbol `s` with index (1)
1 1   : At the second argument position, there is only (1) possible

---

[6]When the value of these numbers used exceeds 127, two or more bytes are used to encode them. Strings are written as strings to improve readability.

top symbol and that is `s` with index (`1`).
1 `"s"`   : The length (`1`) and ASCII representation of `s`.
1   : The arity (`1`) of `s`.
2 1 2   : The single argument of `s` can be either of two (`2`) different top
             function symbols: `s` with index (`1`) or `z` with index (`2`).
1 `"z"`   : The length (`1`) and ASCII representation of `z`.
0   : The arity (`0`) of `z`.

Following this symbol information, the actual term `mult(s(s(z)),s(z))` can be encoded using only a handful of bits. Note that the first function symbol in the symbol table is always the top function symbol of the term (in this case: `mult`):

   : No bits need to be written to identify the function symbol `s`,
     because it is the only possible function symbol at the first
     argument position of `mult`.
0 : One bit indicates which term over the function symbol `s` is
     written (`s(s(z))`). Because this term has not been written yet,
     it is done so now.
0 : The function symbol of the only argument of `s(s(z))` is `s`.
1 : `s(z)` has index 1 in the term table of symbol `s`.
1 : Symbol `z` has index 1 in the symbol table of symbol `s`.
   : Because there is only one term over symbol `z`, no bits are
     needed to encode this term. Now we only need to encode the
     second argument of the input term, `s(z)`.
   : No bits are needed to encode the function symbol `s`, because
     it is the only symbol that can occur as the second argument of mult.
1 : `s(z)` has index 1 in the term table of symbol `s`. Because
     this term has already been written, we are done.

Only five bits are thus needed to encode the term `mult(s(s(z)),s(z))`. As mentioned earlier, the amount of data needed to write the table of function symbols at the start of the BAF file is in most cases negligible compared to the actual term data.

2.4   Performance Measurements

2.4.1   Benchmarks

How concise is the ATerm representation and how fast can BAF files be read and written? Since results highly depend on the actual terms being used, we will base our measurements on a collection of terms that cover most applications we have encountered so far.

Artificial Cases

Two artificial cases are used that have been constructed to act as borderline cases:

Random-unique: a randomly generated term over a signature of 9 fixed function symbols with arities ranging from 1 to 9 and an arbitrary number of constant symbols (functions with arity 0). The terms are generated in such a way that all constants are unique. These terms are the worst case for our implementation: there is no regularity to exploit and there are many subterms with a relatively high arity.

Random: a randomly generated term over a signature of 10 function symbols with arities ranging from 0 to 9. In these terms only a single constant can occur which will be shared, but no other regularities can be exploited and there are many subterms with a relatively high arity.

Real Cases

Several real-life cases are used that are based on actual applications:

COBOL Parse Table: a generated parse table for COBOL including embedded SQL and CICS. The grammar consists of 2,009 productions and the generated automaton has 6,699 states. The parse table contains an action-table (2,0947 non-empty entries) and a goto-table (76527 non-empty entries). This is an example of an abstract data type represented as ATerm.

COBOL System: a COBOL system consisting of 117 programs with a total of 247,548 lines of COBOL source code. It has been parsed with the above parse table. The parse trees constructed for these COBOL programs are represented as ATerms, see Section 2.5.1 for more details.

Risla Library: a parse tree of the component library for the Risla language, a domain specific language for describing financial products [3]. This component library consists of 10,832 lines of code.

LPO: a linear process operator (LPO) describing the "firewire" protocol with 1 bus and 9 links [68, 94]. LPOs are the kernel of the μCRL ToolKit [51] which is a collection of tools for manipulation process and data descriptions in μCRL (micro Common Representation Language) [69]. An LPO is a structured process, where the state consists of an assignment to a sequence of typed data variables and its behaviour is described by condition, action and effect functions. These states are represented as ATerms, and are rather complex.

Casl specifications: a collection of abstract syntax trees represented as ATerms of 98 Casl files, the total number of lines of Casl code is 2,506. For more details on Casl and the abstract syntax tree representation as ATerms we refer to Section 2.5.1.

lcc Parse Forest: a new back-end similar to the ASDL back-end [124] has been added to the lcc compiler [71]. This back-end maps the internal format used by the lcc compiler to ATerms. The ATerm representation and the ASDL representation of a C program contain equivalent information.

Term	# nodes	# unique nodes	Sharing (%)	Memory (bytes)	Bytes/ Node
Artificial Cases					
Random-unique	1,000,000	1,000,000	0.00	15,198,694	15.20
Random	1,000,000	92,246	90.81	2,997,120	3.00
Real Cases					
COBOL Parse Table	961,070	97,516	89.85	2,836,529	2.95
COBOL System	31,332,871	470,872	98.50	12,896,609	0.41
Risla Library	708,838	40,073	94.35	960,170	1.35
LPO	8,894,391	225,229	97.47	3,701,438	0.42
Casl Specifications	34,526	11,699	66.12	235,655	6.83
lcc Parse Forest	360,829	86,589	76.00	1,547,713	4.29
S-expressions	593,874	283,891	52.20	9,111,863	15.34
Real Case Averages			82.07		4.51

Table 2.1: Memory usage of ATerms

Given this back-end the C sources of the lcc compiler itself are mapped to ATerms. The lcc compiler consists of 34 source files, consisting of a total of 13,588 lines of source code.

S-expressions: a simple translator has been developed which transforms an S-expression into an ATerm. This translator has been used to process an arbitrary collection of ".el" files containing S-expressions found within the Emacs source tree under Linux. The total number of ".el" files was 738, these files together contained 286,973 lines of code.

In the cases of the COBOL System, Casl Specifications, lcc Parse Forest, and S-Expressions the set of ATerms are combined into and processed as *one* ATerm. Measurements were performed on an ULTRA SPARC-5 (270 MHz) with 256 Mb of memory. All times measured are the user CPU time for that particular job.

2.4.2 Measurements

In Table 2.1, we give results for the memory usage of our sample terms[7]. The five columns give the total number of nodes in each term, the number of unique nodes in each term, the sharing percentage, the amount of memory (in bytes) used for the storage of the term, and the average number of bytes needed per node. As can be seen in these figures, at least in our applications sharing *does* make a difference. By fully exploiting the redundancies in the input terms, we can store a node using on the average 4.5 bytes, and still perform operations on them efficiently. The worst case behaviour is 15 bytes per node. The amount of sharing is clearly less high in case of abstract syntax trees than in case of parse trees represented as AsFix terms. The AsFix terms contain

---

[7]Since we consider the Random-unique and Random cases to be unrepresentative, we only present the averages for the real cases in this and the following tables.
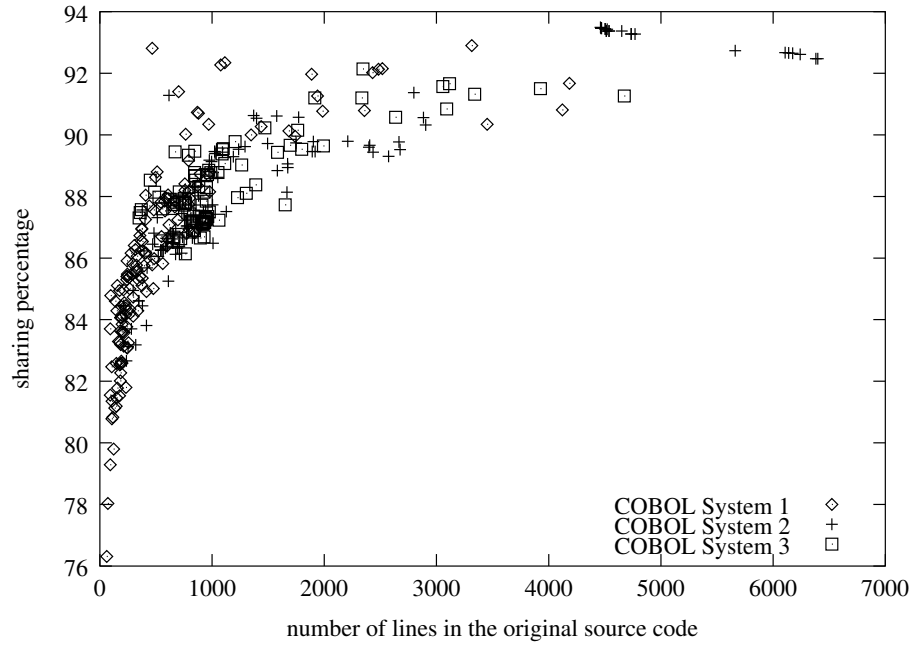
Figure 2.3: Sharing of a large number of COBOL parse trees

much redundant information which can be optimally shared. The amount of sharing in the abstract syntax trees for Casl is lower, but this is due to the fact that the set of Casl specifications is small and each specification tests another feature of the Casl language, so not much sharing was to be expected. The S-expressions have the lowest ratio of sharing, but this was to be expected: they represent ad hoc *hand-written* Lisp programs while in the other cases the ATerms are obtained by a systematic translation from source code. In the latter case, recurring patterns in the translation scheme result in higher levels of sharing.

Figure 2.3 shows the amount of sharing with respect to the size of a large number of COBOL programs. Three different sets of COBOL programs were considered. The first system consists of 151 files, the second of 116 files, and the last of 98 files. From this figure it can be concluded that the amount of sharing increases with the size of the COBOL system. In all three systems, the percentage of sharing converges to slightly over 90%. We find this high percentage in combination with the strong correlation between size and sharing very remarkable and will analyze its causes and consequences in further detail in a separate paper.

In Table 2.2 we give results for reading and writing our sample terms as ASCII text files. The six columns give the size of the text representation of the test term in bytes, the average number of bytes per node, the time needed to read the text file, the average time needed to read a node, the time needed to write the text file, and the average time needed to write a node. On the average, a node requires 6.2 bytes and reading and writing requires 10.5 μs and 2.7 μs, respectively.

Term	ASCII (bytes)	Bytes/ Node	Read (s)	Read/ Node (μs)	Write (s)	Write/ Node (μs)
Artificial Cases						
Random-unique	6,888,889	6.89	34.76	34.76	4.06	4.06
Random	6,200,251	6.20	15.90	15.90	3.67	3.67
Real Cases						
COBOL Parse Table	4,211,366	4.38	6.33	6.95	2.30	2.29
COBOL System	135,350,005	4.32	199.43	6.36	65.02	2.08
Risla Library	2,955,964	4.17	4.25	6.00	1.40	1.98
LPO	41,227,481	4.64	81.90	9.21	29.16	3.28
Casl Specifications	217,958	6.31	0.36	10.43	0.08	2.32
lcc Parse Fores	2,132,245	6.22	3.13	9.14	0.86	2.51
S-expressions	7,954,550	13.39	15.09	25.41	2.49	4.19
Real Case Averages		6.20		10.50		2.66

Table 2.2: Reading and writing ATerms as ASCII text

In Table 2.3 we give results for reading and writing BAF files for the same set of sample terms. The columns give in order: the size of the BAF files in bytes, the average number of bytes needed per node, the time to read the BAF representation, the average read time per node, the time to write the BAF representation, and the average write time per node. Typically, we can read a node in 1.3 μs and write it in 2.4 μs.

Note that reading a BAF term is faster than writing the same term, whereas in case of ASCII the writing is faster than reading. This is caused by the fact that reading the ASCII representation of an ATerm involves numerous matching operations, whereas reading the BAF representation can be done with less matching. On the other hand, writing the BAF representation involves more calculations to encode the sharing of terms, whereas writing the ASCII representation involves a straightforward term traversal.

In Table 2.4 we show how the compression in BAF files compares to the compression of the standard Unix utility `gzip`. Considering the same set of examples, we give figures for a straightforward dump of each term as ASCII text (column 1), the size of the BAF version of the same term (column 2) and percentage of compression achieved (column 3). Next, we give the results of compressing the ASCII version of each term with `gzip` (column 4), and compression achieved (column 5). The compression factors are 85% for BAF and 92% for `gzip`. The worst case compression of `gzip` (66%) is considerably better than the worst case compression using BAF (12%). No gains are to be expected from using `gzip` instead of BAF, since this would imply first writing the ATerm in textual format (an expensive operation which looses sharing) and then compressing it with `gzip`.

Term	BAF	Bytes/ Node	Read	Read/ Node	Write	Write/ Node
	(bytes)		(s)	(μs)	(s)	(μs)
Artificial Cases						
Random-unique	6,073,795	6.07	8.85	8.85	11.57	11.57
Random	567,419	0.57	2.06	2.06	2.76	2.76
Real Cases						
COBOL Parse Table	370,450	0.39	0.63	0.66	1.75	1.82
COBOL System	2,279,066	0.07	4.88	0.16	20.76	0.66
Risla Library	141,946	0.20	0.22	0.31	0.75	1.06
LPO	1,106,661	0.12	1.86	0.21	9.40	1.06
Casl Specifications	32,083	0.93	0.05	1.45	0.15	4.34
lcc Parse Forest	358,318	0.99	0.34	0.99	0.95	2.77
S-expressions	4,438,229	7.47	3.31	5.57	10.49	6.23
Real Case Averages		1.45		1.32		2.42

Table 2.3: Reading and writing ATerms as BAF

Term	ASCII	BAF	Comp.	gzip	Comp.
	(bytes)	(bytes)	(%)	(bytes)	(%)
Artificial Cases					
Random-unique	6,888,889	6,073,795	11.8	2,324,804	66.3
Random	6,199,981	567,419	90.9	439,293	92.9
Real Cases					
COBOL Parse Table	4,211,366	370,450	91.2	230,297	94.5
COBOL System	135,350,005	2,279,066	98.3	3,072,774	97.7
Risla Library	2,955,964	141,946	95.2	80,009	97.3
LPO	41,227,481	1,106,661	97.3	804,521	98.0
Casl Specifications	217,958	32,083	85.3	20,767	90.5
lcc Parse Forest	2,244,691	358,318	84.0	244,502	89.1
S-expressions	7,954,550	4,438,229	44.2	1,858,366	76.6
Real Case Averages			85.1		92.0

Table 2.4: BAF *versus* gzip

	Memory	ASCII	BAF
Size per node (bytes)	4.51	6.20	1.45
Read node (μs)		10.50	1.32
Write node (μs)		2.66	2.42

Table 2.5: Summary of measurements (based on Real Case averages)

2.4.3 Summary of Measurements

These measurements are summarized in Table 2.5. For in-memory storage, 4.5 bytes are needed per node. Using BAF, only 1.54 bytes are needed to represent a node. Also observe that reading BAF is an order of magnitude faster than reading terms in textual form. In case of parse trees represented as AsFix (COBOL System and Risla Library) less than 2 bytes are needed to represent a node in memory and less than 2 bits (0.20 bytes) are needed to represent it in binary format.

2.5 Applications

ATerms have already been used in applications ranging from development tools for domain specific languages [53] to factories for the renovation of COBOL programs [40]. The ATerm data type is also the basic data type to represent the terms manipulated by the rewrite engines generated by the ASF+SDF compiler [33] and they play a central role in the development of the new ASF+SDF Meta-Environment [35].

2.5.1 Representing Syntax Trees: AsFix and CasFix

The ATerm data type proves to be a powerful and flexible mechanism to represent syntax trees. By defining an appropriate set of function symbols parse trees and abstract syntax trees can be represented for any language or formalism. We describe two examples: AsFix (a parse tree format for ASF+SDF, Section 2.5.1) and CasFix (an abstract syntax tree format for Casl, Section 2.5.1).

AsFix

AsFix (ASF+SDF Fixed format) is an incarnation of ATerms for representing ASF+SDF [72, 10, 52]. ASF+SDF is a modular algebraic specification formalism for describing the syntax and semantics of (programming) languages. SDF (Syntax Definition Formalism) allows the definition of the concrete and abstract syntax of a language and is comparable to (E)BNF. ASF (Algebraic Specification Formalism) allows the definition of the semantics in terms of equations, which are interpreted as rewrite rules. The development of ASF+SDF specifications is supported by an integrated programming environment, the ASF+SDF Meta-Environment [86].

Using AsFix, each module or term is represented by its parse tree which contains both the syntax rules used and all original layout and comments. In this way, the original source text can be reconstructed from the AsFix representation, thus enabling transformation tools to access and transform comments in the source text. Since the AsFix representation is self-contained (all grammar information needed to interpret the term is also included), one can easily develop tools for processing AsFix terms which do not have to consult a common database with grammar information. Examples of such tools are a (structure) editor or a rewrite engine.

AsFix is defined by an appropriate set of function symbols for representing common constructs in a parse tree. These function symbols include the following:

- `prod`(T) represents production rule T.

- `appl`(T_1, T_2) represents applying production rule T_1 to the arguments T_2.

- `l`(T) represents literal T.

- `sort`(T) represents sort T.

- `lex`(T_1, T_2) represents (lexical) token T_1 of sort T_2.

- `w`(T) represents white space T.

- `attr`(T) represents a single attribute.

- `attrs`(T) represents a list of attributes.

- `no-attrs` represents an empty list of attributes.

The following context-free syntax rules (in SDF [72]) are necessary to parse the input sentence `true or false`.

```
sort Bool
context-free syntax
  true          -> Bool
  false         -> Bool
  Bool or Bool -> Bool {left}
```

The parse tree below gives the AsFix representation for the input sentence `true or false`.

```
appl(prod([sort("Bool"),l("or"),sort("Bool")],sort("Bool"),
        attrs([attr("left")])),
    [appl(prod([l("true")],sort("Bool"),no-attrs),[l("true")]),
     w(" "),l("or"),w(" "),
     appl(prod([l("false")],sort("Bool"),no-attrs),[l("false")])
    ])
```

Two observations can be made about this parse tree. First, this parse tree is an ordinary ATerm, and can be manipulated by all ATerm utilities in a completely generic way.

Second, this parse tree is completely self-contained and does not depend on a separate grammar definition. It is clear that this way of representing parse trees contains much redundant information. Therefore, both maximal sharing and BAF are essential to reduce their size. In our measurements, AsFix only plays a role in the cases COBOL System and Risla Library.

The annotations provided by the ATerm data type can be used to store auxiliary information like position information derived by the parser or font and/or color information needed by a (structure) editor. This information is globally available but can be ignored by tools that are not interested in it.

CasFix

Casl (Common Algebraic Specification Language) is a new algebraic specification formalism [48] developed as part of the CoFI initiative. It is a general algebraic specification formalism incorporating common features of most existing algebraic specification languages. In addition to the language itself, a set of tools is planned for supporting the development of Casl specifications. Existing tools will be reused as much as possible.

In order to let the various tools, like parsers, editors, rewriters, and proof checkers, communicate with each other an intermediate format was needed for Casl. ATerms have been selected as intermediate format and a specialized version for representing the *abstract* syntax trees of Casl has been designed (CasFix [32]). Contrast this with the approach taken for AsFix, where the more concrete *parse trees* are used as intermediate representation.

CasFix is obtained by defining an appropriate set of function symbols for representing Casl's abstract syntax [48] and by defining a mapping from Casl's concrete syntax to its abstract syntax. For each abstract syntax rule an equivalent CasFix construct is defined as in:

```
ALTERNATIVE ::= "total-construct" OP-NAME COMPONENTS*
```

$\Longrightarrow$

```
total-construct(<OP-NAME>,COMPONENTS*([<COMPONENTS>]))
```

In this example `"total-construct"` and `"COMPONENTS*"` are function symbols and `<OP-NAME>` and `<COMPONENTS>` represent the subtrees of the corresponding sort.

2.5.2   ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [86] is an interactive development environment for writing language specifications in ASF+SDF. A new generation of this environment is being developed based on separate components connected via the ToolBus [15]. A description of this new architecture can be found in [35]. The new Meta-Environment provides tools for parsing, compilation, rewriting, debugging, and formatting. ATerms and AsFix play an important role in the new Meta-Environment:

- The parser generator [118] produces a parse table represented as ATerm.

- The parser uses this parse table and transforms an input string into a parse tree which is represented as AsFix term.

- After parsing, the modules of an ASF+SDF specification are stored as AsFix terms. Information concerning the specification such as the rewrite rules that must be compiled are exchanged as AsFix terms.

- The ASF+SDF compiler (see next section) reads and writes AsFix terms.

41

Specification	ASF+SDF (equations)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF compiler (sec)	C compiler (sec)
ASF+SDF compiler	1,876	8,699	85,185	216	323

Table 2.6: Some figures on the ASF+SDF compiler.

Application	Time (sec)	Memory (Mb)
ASF+SDF compiler (with sharing)	216	16
ASF+SDF compiler (without sharing)	661	117

Table 2.7: Performance with and without maximal sharing.

2.5.3   ASF+SDF to C compiler

The ASF+SDF to C compiler [33] is a compiler for ASF+SDF. It generates ANSI-C code and depends on the ATerm library as run-time environment. All terms manipulated by the generated C code are represented as ATerms thus taking advantage of maximal subterm sharing and automatic garbage collection.

The optimized memory usage of ATerms has already been exploited in various industrial projects [25, 34] where memory usage is a critical success factor. This ASF+SDF compiler has, for instance, been applied successfully in projects such as the development of a domain-specific language for describing interest products (in the financial domain) [3] and a renovation factory for restructuring COBOL code [40].

The ASF+SDF compiler is an ASF+SDF specification and has been bootstrapped. Table 2.6 gives some figures on the size of this specification and the time needed to compile it. Table 2.7 gives an impression of the effect of compiling the ASF+SDF compiler with and without sharing. More information on the compiler itself and on performance issues can be found in [33].

2.5.4   Other Applications

Other applications are still under development and include:

- A tool for protocol verification [68]. The ATerms are used to represent the states in the state space of the protocol. Because of the huge amount of states ($\geq 1,000,000$) it is necessary to share as many states as possible.

- A tool for the detection of code clones in legacy code.

- The Stratego compiler [120].

2.6 Discussion

2.6.1 Related Work

S-expressions in LISP   Many intermediate representations are derived in some form or another from the S-expressions in LISP. ATerms are no exception to this rule. The main improvements of ATerms over S-expressions are

- ATerms support arbitrary binary data (Blobs, see Section 2.2.1).

- ATerms support annotations.

- ATerms support maximal sharing in a systematic way.

- ATerms support a concise, sharing preserving, exchange format that exploits the implicit signature of terms.

- The ATerm library provides a comprehensive collection of access functions based on the *match-and-make* paradigm.

Intermediate representations in compiler frameworks   There exist numerous frameworks for compilers and programming environments that provide facilities for representing intermediate data. Examples are Centaur's VTP [23], Eli [67], Cocktail's Ast [70], SUIF [128], ASDL [124], and Montana [83]. These systems either provide an explicit intermediate format (Eli, Ast, SUIF) or they provide a programmable interface to the intermediate data (VTP, Montana, ASDL). Lamb's IDL [91] and OMG's IDL [107] are frameworks for representing intermediate data that are not tied to a specific compiler construction paradigm but have objectives similar to the systems already mentioned.

These approaches typically use a grammar-like definition of the abstract syntax (including attributes) and provide (generated) access functions as well as readers and writers for these intermediate data. In most cases support exists for accessing the intermediate data from a small collection of source languages.

The major difference between these approaches and ATerms is that they operate at different levels of abstraction. ATerms just provide the lower-level representation for terms (or more precisely directed acyclic graphs), while intermediate representations for compilers are more specialized and give a higher-level view on the intermediate data. They provide primitives for representing program constructs, symbol tables, flow graphs and other derived information. In most cases they also provide a fixed format for representing programs at different levels of abstraction ranging from call graphs to machine-like instructions. ATerms are thus simpler and more general and they can be used to represent each of these compiler's intermediate formats.

Another difference is that most compiler frameworks use a statically typed intermediate representation. The major advantage is early error-detection. The disadvantages are, however, less flexibility and the need to generate different access functions for each different intermediate format. In the case of ATerms, a dynamic check may be necessary on the intermediate data but only a single, generic, set of access functions is needed.

Term	ASCII	BAF	ASDL pickle
COBOL Parse Table	4,211,366	370,450	5,262,426

Table 2.8: Sizes of the COBOL parse table (in bytes)

Term	ASCII	BAF	ASDL pickle
lcc Parse Forest	2,246,436	624,091	1,290,595

Table 2.9: Sizes of abstract syntax trees (in bytes)

ASDL   The abstract syntax definition language (ASDL) [124] is a language for describing tree data structures and is used as intermediate representation language between the various phases of a compiler [71]. We consider ASDL in more detail, because of its public availability and the fact that the goals of ASDL and ATerms are quite similar as they are both used to exchange of syntax trees between tools, although ATerms are more general in the sense that other types of information, such as unstructured binary objects and annotations, can also be represented as an ATerm. Everything that can be represented by a grammar can be represented in ATerms as well as ASDL.

ASDL pickles and the BAF format for ATerms are comparable with respect to functionality, both are binary representations of (among others) syntax trees. The pickle and unpickle functions are generated from the ASDL description and are thus application specific (this may be more efficient) whereas the reading and writing of BAF is entirely generic (this avoids the proliferation of versions).

ASDL and ATerms can be compared at two different levels:

- *Low level:* ASDL pickle versus plain ATerms. By providing an ASDL definition of ATerms we can compare the size of the same object as ATerm (ASCII and BAF) and as ASDL pickle. This is done in Table 2.8 for the COBOL Parse Table. In this case, the representation in BAF is an order of magnitude smaller than the ASDL pickle.

- *High level*: compare at the level of parse trees or abstract syntax trees. ASDL is typically used to represent abstract syntax trees while ATerms can be used to represent both as we have discussed in Section 2.5.1. To make a meaningful comparison, we compare the abstract syntax trees generated by the lcc back-end in ATerm format (both in ASCII and BAF) and the corresponding ASDL pickles. These figures are presented in Table 2.9 for the abstract syntax trees generated for the lcc source files. In this case the BAF representation is 2 times smaller than the ASDL pickle. Note that the figure for the BAF representation differs from the figure in Table 2.3, this is caused by the fact that in Table 2.3 all files are combined into one BAF term whereas in Table 2.9 each file is a separate BAF term and their sizes are added.

XML   The Extensible Markup Language [129] is a recently standardized format for Web documents. Unlike HTML, XML makes a strict distinction between *content* and

presentation. XML can be *extended* by adding user-defined *tags* to parts of a document and by defining the overall structure of the document thus enabling well-formedness checks on documents. Although the original objectives are completely different, there are striking similarities between ATerms and XML: both serve the representation of hierarchically structured data and both allow arbitrary extensions (adding tags *versus* adding function symbols). There is also a straightforward translation possible between ATerms and XML.

The main difference between the two is that XML is more verbose and does not provide a simple mechanism to represent sharing, whereas ATerms provide the BAF format. This may not be a problem for Web documents like catalogs and database records, but it does present a major obstacle in our case when we need to exchange huge terms between tools. We are currently considering whether some link between ATerms and XML may be advantageous.

Data encodings As described in Section 2.3.5, we use a form of data encoding to compress ATerms when they are exchanged between tools. Of course, encoding and data compression techniques are in common use in telecommunications. For instance, the ASN.1 standard gives detailed rules for data encoding [4].

In an earlier project in our group, the Graph Exchange Language (GEL) [82] has been developed. It is similar in goals to BAF, but BAF can only represent acyclic directed graphs, whereas GEL can represent arbitrary (potentially cyclic) graphs. The technical approaches are different as well. GEL uses a binary-encoded postfix format to represent the nodes in the graph and introduces explicit labels to reuse previously constructed parts of the graph. BAF uses a prefix format augmented by generated symbol tables.

A final difference is in the *usage* of both approaches. GEL was used as a separate library that could be used in applications and the graph encoding was therefore visible to the programmer using it. BAF is, on the other hand, completely integrated in the ATerm implementation and is only used by the standard read and write functions for ATerms. The BAF format is therefore never visible to programmers.

Hash consing In LISP, the success of hash consing [1] has been limited by the existence of the functions `rplaca` and `rplacd` that can destructively modify a list structure. To support destructive updates, one has to support two kinds of list structures "mono copy" lists with maximal sharing and "multi copy" lists without maximal sharing. Before destructively changing a mono copy list, it has to be converted to a multi copy list. In the 1970's, E. Goto has experimented with a Lisp dialect (HLisp) supporting hash consing and list types as just sketched. See [115] for a recent overview of this work and its applications.

A striking observation can be made in the context of SML [2] where sharing resulted in slightly increased execution speed and only marginal space savings. On closer inspection, we come to the conclusion that both methods for term sharing are different and can not be compared easily. We share terms immediately when they are created: the costs are a table lookup and the storage needed for the table while the benefits are space savings due to sharing and a fast equality test (one pointer comparison). In [2]

sharing of subterms is *only* determined during garbage collection in order to minimize the overhead of a table lookup at term creation. This implies that local terms that have not yet survived one garbage collection are not yet shared thus loosing most of the benefits (space savings and fast equality test) as well.

2.6.2 History

Terms are so simple that most programmers prefer to write their own implementation rather than using (or even *looking for*) an existing implementation. This is all right, except when this happens in a group of cooperating developers as in our case.

A very first version of the ATerm library was developed as part of the ToolBus coordination architecture [15]. It was used to represent data which were transported between tools written in different languages running on different machines. Simultaneously, we were developing a formalism for representing parse trees [26]. In addition, incompatible term formats were in use in various of our compiler projects [61]. Observing the similarities between all these incompatible term data types triggered the work on ATerms as described here. The benefits are twofold. First, a common term data type is used in more applications and investments in it are well rewarded. Second, the mere existence of a common data type leads to new, unanticipated, applications. For instance, we now use ATerms for representing parse tables as well.

2.6.3 Conclusions

As stated in the introduction, ATerms are intended to form an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of (tree-like) data structures between distributed applications.

ATerms *are* open and language independent since they do not depend on any specific hardware or software platform. ATerms *are* simple: the level one interface consists of only 13 functions. ATerms *are* efficient and concise as shown by the measurements in Section 2.4. Last but not least, ATerms are also *useful* as shown on Section 2.5.

The ATerm format is supported by a binary exchange format (BAF) which provides a mechanism to exchange ATerms in a concise way. This BAF format maintains the in-memory sharing of terms and uses a minimal amount of bits to represent the nodes, in case of AsFix terms only 2 bits are needed per node.

The most innovative aspects of ATerms are the simple procedural interface based on the *make-and-match* paradigm, term annotations, maximal subterm sharing, and the concise binary encoding of terms that is completely hidden behind high-level read and write operations.

46

C H A P T E R 3

Generation of Abstract Programming Interfaces from Syntax Definitions

3.1 Introduction

Since the development of the ATerm-Library [30] in 1999, its use for the implementation of tree-like data structures has become quite popular among developers of scanners, parsers, rewrite engines and model checkers. Apart from its inevitable deployment in the tools of the Meta-Environment [86, 35, 24] for which it was specifically designed, the ATerm-Library is used amongst others in: the ELAN system [38], the XT Program Transformation Tools [80] which are based on the Stratego Language [119], the CoFI Algebraic Specification Language CASL [48, 47], Strafunski [92]: a Haskell-centered software bundle for generic programming and language processing, and the μCRL ToolSet for Analyzing Algebraic Specifications [19]. ATerms include several nice features: they are easy to manipulate yet very efficient; they come with a built-in garbage collector (in the C library), and they have persistence support in the form of a compact, sharing preserving serialization in both textual and binary representations.

As more and more tools in the Meta-Environment were converted to work with the ATerm-Library, it became apparent that the tools had become inflexible with respect to changes in the parse tree format (called AsFix), and were hard to maintain. The reason behind this inflexibility was the fact that all tools used manually encoded structural knowledge about the signature of the data types, i.e. the location of data elements inside their ATerm representation. Hard-wiring such knowledge into the tools without an explicit signature definition makes it difficult, if not impossible, to change the ATerm representation of the data type.

The coding practice that uses such structural knowledge is not in any way restricted to the realm of parse trees. In fact, anyone who has programmed with the ATerm-Library, will probably be familiar with patterns such as `and(<bool>,<bool>)`. And given such a pattern, what could be easier than writing a function that extracts the arguments of the expression? But as these patterns

become longer and more intricate with a liberal sprinkling of quoted strings containing backslash-escaped quotes, and as they begin to contain lists and annotations, the once so simple *make-and-match* paradigm becomes a developer's nightmare.

Shielding ATerm representation knowledge in access macros somewhat improves the legibility of code that uses them, but it does not in any way remove the maintenance issue. It restricts the knowledge to a specific set of macros, but these still need manual maintenance. As a result, this approach only *looks like* representation hiding, but in fact all programmers of different tools still need to know the exact ATerm representation of the data being exchanged.

Motivated by the need to change AsFix and to avoid the herculean maintenance task this operation would impose on our toolset, we decided to remove as much "ATerm-handicraft" from the tools as possible by developing an API-generator that creates both an interface and an implementation of data structures represented by ATerms.

While maintaining the advantages of the ATerm-Library (in our case most notably its efficiency due to *maximal subterm sharing*[1]), applications built with this generated API benefit from improved simplicity and readability, they are easier to maintain, and they are more robust against changes in the underlying AsFix representation.

This chapter describes how an annotated grammar or syntax definition can be used to generate a library of functions that provide access to the parse trees of terms over this grammar. Such a library effectively turns a parse tree into an abstract data type, providing a type-safe and systematic API to manipulate terms. In particular we describe how a SDF-specification commonly found when using the Meta-Environment is used to collect the information into an *annotated data type* (ADT), necessary to build a mapping between grammar productions and their ATerm-pattern in the underlying AsFix parse tree, and how this mapping is subsequently used to generate C functions that provide an API to these parse trees. A schematic overview is shown in Figure 3.1.

Although this chapter uses tools from the Meta-Environment as a running example, the maintenance issues addressed here are not specific to parse trees at all. The issues are fundamental to all applications that use ATerms as its data structure representation. Moreover, many of these issues are also found in applications based on other generic data representation formalisms like for instance XML.

We first relate our approach to other work in Section 3.1.1, and continue with some introductory sections on the specification formalism ASF+SDF (Section 3.1.2), the syntax of ATerms (Section 3.1.3), and AsFix (Section 3.1.4). Section 3.2 explains how ATerm-based data types are typically accessed in tools and applications and we show how this approach causes development and maintenance problems. We then describe the actual generation scheme from SDF to an intermediate representation (Section 3.3) and the subsequent generation into the target language (Section 3.4). The results of the application of our generation technique on tools in the Meta-Environment are shown in Section 3.5, followed by some conclusions (Section 3.6), a discussion (Section 3.7) and future work (Section 3.8).

---

[1] Our strategy to minimize memory usage is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term is reused, ensuring maximal (sub)term sharing.
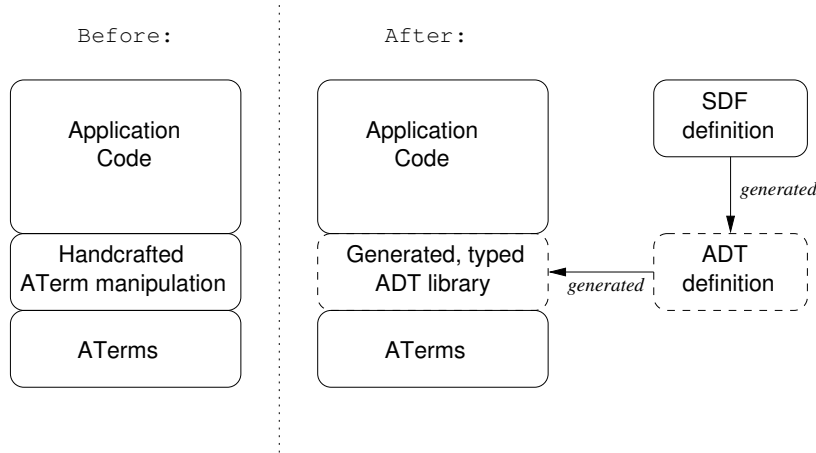
Figure 3.1: Overview of an application before and after introduction of API-generation.

3.1.1   Related Work

The techniques described in this chapter both use the Meta-Environment and *are used* to improve the tools therein and as such our work can be seen as one step towards the implementation of [73]. We now briefly discuss some related work that deals with the aspects we address in this chapter: applying *generational* techniques to create an *abstraction layer* on top of a *generic* data exchange formalism.

Grammars as Contracts   In [81] a generic framework is presented that includes the generation of libraries from concrete syntax definitions. These libraries can then be used to manipulate both parse trees and abstract syntax trees. Just like our work, the instantiations are based on SDF as syntax definition formalism, in combination with tool support from the Meta-Environment. Instantiations are described for generating libraries in different languages including C, Java, Stratego, and Haskell.

The work described in our chapter can be seen as a refined instantiation of this generic framework. Among the instantiations described in [81] generation of a C library for concrete syntax manipulation is missing, and our approach remedies this situation. We also focus on generating more intuitive and readable API's, at the cost of extra annotation effort on the original syntax definition.

Zephyr ASDL   The abstract syntax definition language (ASDL) [124] is a language for describing tree data structures much like ATerms, and is used as intermediate representation language between the various phases of a compiler [71].

The ASDL tools support the generation of accessor and serialization code. The main differences with our approach are:

- ASDL works on abstract syntax definitions. The link between parser and ASDL must be programmed manually;

- ASDL supports a wide variety of languages including C, C++, Java, Standard ML, and Haskell;

- ASDL offers a graphical browser and editor for data described in ASDL;

- ASDL does not support maximal subterm sharing;

- There is no garbage collection support for languages like C and C++.

(D)COM/Corba IDL compiler The two major commercial component architectures, Microsoft's (D)COM and OMG's CORBA [49, 113], both provide IDL compilers that take an interface definition written in their respective *interface definition languages* (IDLs), and generate communication scaffolding code. The generated code includes *stubs* and *skeletons* to make it easy to write clients and servers respectively.

The biggest difference with our work is that the target of these systems is to make it easy for programmers to build components in a distributed setting while we focus on providing an abstraction layer on top of a generic data exchange format. This means that the IDL compilers generate code for marshaling arguments when calling remote procedures and for unmarshaling their return values, while in our approach we keep the data in the original "marshaled" form until it is actually used. In this sense, our approach could be characterized as lazy and the DCOM/CORBA approach as eager.

XML data binding in Java A comparable approach to provide an abstraction layer on top of a generic data exchange formalism is used in `jaxb` [74]. This is a tool that generates a Java class hierarchy from an XML DTD. Besides accessor functions and constructors, (de)serialization functions to and from XML are generated. The actual code generation can be steered using a specification in XML. This makes it possible to add e.g. interfaces and extra code to the generated classes.

In general this approach is called *data binding*, and several other initiatives in this area are currently under way, including some open source initiatives [58, 59] and the commercial initiative [42]. All these approaches offer tool support for generating `Java` code from an XML Schema. The generated code can *marshal* and *unmarshal* XML terms to Java objects with accessors to retrieve type safe (sub)elements.

Generative Programming Generative programming focuses on using domain engineering to retrieve domain specific knowledge that can be incorporated into component generators [50]. At first glance this "high level" view on generating software components seems to be far removed from the low level view on source code generation we have taken in this chapter. If we take a closer look, the two approaches are not as disjoint as one might think. We believe any successful generic approach to generative programming must be based on some abstract data type definition augmented with domain specific knowledge. In our case, the data type definitions are written in SDF, and the domain specific knowledge consists of the mapping of such data types to concrete AsFix representations.

JJForester   Another approach to generating code from an SDF definition is taken in JJForester [90]. JJForester combines the parser generator and parser from the Meta-Environment with a tree builder, and visitor generator for Java. The focus lies on the generation of tree manipulation and especially tree traversal support. Especially nice is the integration with JJTraveler [121], providing generic visitor combinator support.

3.1.2   ASF+SDF in a nutshell

The specification formalism ASF+SDF [10, 72] is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. An overview can be found in [52]. As an illustration, Figure 3.2 presents the definition of the Boolean data type in ASF+SDF [2]. ASF+SDF specifications consist of modules, where each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations).

SDF

The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation but instead arbitrary context-free grammars can be defined. SDF contains some interesting features that make it possible to give concise definitions of context-free grammars:

- Both context-free and lexical syntax can be specified.

- Lexical syntax can be described using regular expressions.

- Associativity can be specified using *attributes* (left, right, non-assoc).

- Priority relations between productions can be specified in priority sections.

- Grammar specifications can be modular.

- Modules and sorts can be parameterized.

- A number of heavily used constructs are built-in including lists, separated lists, alternatives, tuples, and function application.

The syntax defined in the SDF-part of a module can be used immediately when defining equations, thus making the syntax used in equations *user-defined*.

The technology behind SDF is based on *scannerless generalized LR parsing* [39]. The term *scannerless* indicates that there is no separate scanning phase before parsing: each character is a token. This approach has the advantage that the class of languages that can be handled by the parser is not restricted by local tokenization decisions taken by the scanner.

The term *generalized* means that the parser can handle ambiguous constructs and in general yields a parse *forest* instead of a single parse tree.

---

[2]Note how in SDF left-hand and right-hand sides of a production have opposite meaning compared to BNF notation. In SDF the elements of the LHS produce the RHS, in BNF notation the LHS is produced by the elements of the RHS.

```
module Bool
imports Layout
exports
  sorts Bool
  context-free syntax
    "true"          -> Bool
    "false"         -> Bool
    "not" Bool      -> Bool
    Bool "and" Bool -> Bool {left}
    Bool "or" Bool  -> Bool {left}

hiddens variables
    "Bool" -> Bool

equations
  [not-1] not true = false
  [not-2] not false = true

  [and-1] Bool and false = false
  [and-2] Bool and true = Bool

  [or-1] Bool or true = true
  [or-2] Bool or false = Bool
```

Figure 3.2: ASF+SDF specification of the Booleans

To implement scannerless parsing for SDF the SDF *normalizer* is used to transform a SDF grammar into a simple character level grammar. One of the tasks of the normalizer is to explicitly insert *layout* symbols between all symbols in context-free syntax sections. For the syntax defined in Figure 3.2 this means that whitespace can be inserted between keywords, for instance between `not` and `true` in equation `not-1`. In this example, the actual definition of what constitutes whitespace is defined in the module Layout that is not shown in the example.

ASF

The equations appearing in the ASF-part of a specification have the following distinctive features:

- Conditional equations with positive and negative conditions.

- Non left-linear equations.

- List matching.

- Default equations.

It is possible to execute specifications by interpreting the equations as conditional rewrite rules. The semantics of ASF+SDF are based on innermost rewriting. Default

equations are tried when all other applicable equations have failed, either because the arguments did not match or because one of the conditions failed.

The development of ASF+SDF specifications is supported by an interactive programming environment, the Meta-Environment [24]. In this environment specifications can be developed and tested. It provides syntax-directed editors, a parser generator, and a rewrite engine. Given this rewrite engine terms can be reduced by interpreting the equations as rewrite rules. For instance, the term

```
true or false
```

reduces to `true` when applying the equations of Figure 3.2.

3.1.3 Annotated Terms: the ATerm syntax

The definition of the concrete syntax of ATerms is given in Appendix A.1. Here are a number of examples to (re-)familiarize the reader with some of the features of the textual representation of ATerms:

- Integer and real constants are written conventionally: `1`, `3.14`, and `-0.7E34` are all valid ATerms.

- Function applications are represented by a function name followed by an open parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: `f(a,b)` and `"test!"(1,2.1,"Hello world!")`. These examples show that double quotes can be used to delimit function names that are not identifiers.

- Lists are represented by an opening square bracket, a number of list elements separated by commas and a closing square bracket: `[1,2,"abc"]`, `[]`, and `[f,g([1,2],x)]` are examples.

- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are: `<int>`, `<[3]>`, and `<f(<int>,<real>)>`.

3.1.4 ASF+SDF Parse Trees for Dummies: AsFix explained

From a SDF-specification, a parse table can be generated using the `pgen` tool from the Meta-Environment. `pgen` consists of the normalizer discussed earlier combined with a parse table generator. The resulting parse table can subsequently be used by `sglr`: the scannerless, generalized LR parser to parse input terms over the syntax described by the SDF-specification. The result of a successful parse is a parse forest, containing parse trees. The data structure used to represent parse trees is called AsFix, and is implemented using the ATerm-Library to exploit the maximal subterm sharing that is commonly present in parse trees.

Because AsFix is a parse tree format (as opposed to an abstract syntax tree), layout in the input term is preserved, and other syntax-derived facts such as associativity

and constructor information is made available to any tool that has access to the AsFix representation of the input term.

The definition of the concrete syntax of AsFix is given in Appendix B.1, but to quickly familiarize the reader with AsFix, we show some of its idiosyncrasies by means of some real life examples.

Example: grammar production `"true" -> Bool` The AsFix representation of the SDF production

```
"true" -> Bool
```

is:

```
prod([lit("true")],cf(sort("Bool")),no-attrs)
```

The `prod` symbol declares this to be a grammar production. It has three arguments: the first is a list of terminals and non-terminals that occur in the left-hand side of the production, the second argument is the non-terminal of the right-hand side, and the third argument contains the attributes (e.g. left associativity) of the production.

In this example, the literal (denoted by the symbol `lit`) `true` is the only element in the left-hand side of the production. It is injected into the context-free (denoted by the symbol `cf`) non-terminal `Bool`. The production has no specific attributes (`no-attrs`).

Example: grammar production `Bool "and" Bool -> Bool {left}` The AsFix representation of the following grammar production:

```
Bool "and" Bool -> Bool {left}
```

is:

```
1  prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),
2        cf(opt(layout)),cf(sort("Bool"))],
3     cf(sort("Bool")),
4     attrs([assoc("left")]))
```

- Lines 1 and 2 declare this to be a grammar production (`prod`), containing all the elements of the left-hand side of the production. The SDF-normalizer has inserted the context-free sort `opt(layout)` subterms at every location where optional layout in the input term is allowed.

- Line 3 tells us that the result sort of this production is `Bool`.

- Line 4 shows the attributes associated with this production. In this case the only attribute is `left` for left-associativity.

54

Example: parsed term `true and false`   When the input term `true and false` is parsed, the resulting parse tree is the grammar production from the previous example, applied to the actual argument `true and false`. The layout in the input term consists of exactly one space immediately before and after the keyword `and`.

```
1 appl(
2  prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),cf(opt(layout)),
3         cf(sort("Bool"))],cf(sort("Bool")),attrs([assoc("left")])),
4 [appl(prod([lit("true")],cf(sort("Bool")),no-attrs),[lit("true")]),
5  layout([" "]), lit("and"), layout([" "]),
6  appl(prod([lit("false")],cf(sort("Bool")),no-attrs),
7        [lit("false")])])
```

- Line 1 states that this tree is the application of a grammar production to a specific term.

- Lines 2–3 show the representation of `Bool "and" Bool -> Bool {left}` from the previous example.

- Line 4 shows the application of the production `"true" -> Bool` to the literal `true`.

- Line 5 contains the instantiated optional layout terms. In this case the input term contained exactly one space immediately before and just after the keyword `and`.

- Similar to line 4, lines 6–7 represent the literal `"false"`.

The fact that many tools in the Meta-Environment need to operate on such parse trees, raises the question of how best to access this ATerm representation of a data type.

3.2   Accessing ATerm Data Types

The ATerm-Library provides two levels of access to ATerms. We briefly discuss both of them (Sections 3.2.1 and 3.2.2) by showing some examples using the C implementation of the ATerm-Library. Similar statements are needed when using the `Java` implementation.

Section 3.2.3 shows the typical way tools in the Meta-Environment used to access AsFix parse trees. As AsFix terms are of impressive complexity to the human eye, the code needed to access them becomes equally complex if it has to be written down manually.

3.2.1   Accessing ATerms using the Level One interface

The first level of access functions is through the easy-to-learn *make and match* paradigm which allows construction of terms by parsing their string representation. Placeholders in these patterns are used to designate "holes" in the term which are to be filled in by other variables, including other ATerms as well as native types (`int`, `string`, etc.). Terms are constructed using `ATmake`, for example:

```
ATerm t = ATmake("person(name(<str>),age(<int>))", "Anthony", 7);
```

will result in term `t` being assigned the value:

```
person(name("Anthony"),age(7))
```

Note how the placeholders `<str>` and `<int>` are substituted by the values `Anthony` and 7, respectively.
Elements from terms can be extracted using `ATmatch`, for example:

```
char *name;
int age;
if (ATmatch(t, "person(name(<str>),age(<int>))", &name, &age)) {
  printf("name = %s, age = %d\n", name, age);
}
```

will result in the variables `name` and `age` being assigned the values `Anthony` and 7, respectively. The output of this fragment would thus be:

```
name = Anthony, age = 7
```

In case we are only interested in extracting the `age` field and we do not care about the actual value of `name`, we can pass `NULL` instead of the address of a local variable. In this case, that particular subterm is still used during matching, but its actual value is never assigned. This allows us to test if a specific term matches a given pattern, without having to bind every placeholder in the pattern.

3.2.2   Accessing ATerms using the Level Two interface

The second level of access allows more direct manipulation of ATerms by means of access-functions which operate directly on a term or its subterms. This way of access is more efficient than using the level one interface, because there is no need to parse a string pattern to find out which part of the (sub-)term is needed.

For example, consider the term from the previous section:

```
t = person(name("Anthony"),age(7))
```

We can get `Anthony`'s age by first extracting the `age` subterm from `t`, and subsequently getting the actual 7 from this `age` term. Arguments in an ATerm function application are numbered, starting at zero. So, to get to the actual value of 7 which is embedded in the `age` function application, we need to extract argument number 1 from the `person` application, and then extract argument number 0 from this:

```
int age = ATgetInt(ATgetArgument(ATgetArgument(t, 1), 0));
```

Note that the exact *location* of the `age` field in the ATerm representation of the `person` record is used. If the structure of the record were to change, e.g. a field for the person's last name is inserted between the `name` and the `age` fields, the example code would be broken.

Also note that this code does not even check if the term `t` is of the right form, i.e. if `t` satisfies the pattern `person(name(<str>),age(<int>))`. On an arbitrary input term, the age-extraction code will most likely fail and dump core. But if only correct input terms are given, it is the most efficient way to encode the extraction of the age subterm in this ATerm representation of the `person` record.

3.2.3 Accessing AsFix parse trees

This Section shows several ways in which AsFix terms can be accessed. The code fragments are typical for the way parse trees are manipulated in the Meta-Environment.

First, we show the C code necessary to construct the boolean term `true` which, when yielded by the parser, looks like this:

```
appl(prod([lit("true")],cf(sort("Bool")),no-attrs), [lit("true")])
```

Even for such a simple input term, its ATerm representation written as a C (or `Java`) string is already quite complex. This is because we have to escape all the double quotes (the `"` characters) from interpretation by the compiler. Also, because the string representation of the match-pattern is long enough that it does not legibly fit on a single line anymore, we have to resort to ANSI C string concatenation[3] to span the string over multiple lines.

```
ATerm true = ATparse(
  "appl(prod([lit(\"true\")],cf(sort(\"Bool\")),no-attrs),"
  "[lit(\"true\")])");
```

As another example, consider a C function that extracts the left-hand side from a boolean conjunction. It needs to match the parse tree of the incoming term against the pattern for the syntax production:

```
Bool "and" Bool -> Bool {left}
```

An implementation using the level one interface would need the pattern written as a string, with a `<term>` placeholder at the correct spot. Because the pattern is written inside a string, we once again need to escape all quotes.

```
ATerm extract_bool_lhs(ATerm t) {
  ATerm lhs;
  char *bool_and_lhs_pattern =
    "appl(prod([cf(sort(\"Bool\")),cf(opt(layout)),"
    "lit(\"and\"),cf(opt(layout)),cf(sort(\"Bool\"))],"
    "cf(sort(\"Bool\")),attrs([assoc(\"left\")])),"
    "[<term>,<term>,lit("and"),<term>,<term>])";

  if (ATmatch(t, bool_and_lhs_pattern, &lhs, NULL, NULL, NULL)) {
    return lhs;
  }

  return NULL;
}
```

Could there be a quote missing in the pattern? Are all the `)`, `]`, and `}` characters where they should be? Did you expect four `<term>` placeholders in the pattern (to account for the lhs, the rhs, as well as the optional layout before and after the literal `and`)?

---

[3]Strings can be split over multiple lines by ending one line with a `"` and starting the next line with another `"`.

Keep in mind that:

- as long as it is a valid C string, the C compiler is not going to warn you if you make a mistake (e.g. you wrote `lit(and)` instead of `lit(\"and\")`);

- as long as it is a valid ATerm-pattern, the ATerm parser is not going to warn you if you make a mistake (e.g. you forgot to add `<term>` placeholders for the optional layout);

- if you made any mistakes, your only hope to fix them lies in visually inspecting the incoming term and the expected matching pattern, and figuring out why they do not match!

An implementation using the level two interface encodes structural knowledge about the exact location of the `lhs` in terms of direct ATerm access functions. In particular, recalling that in AsFix we are dealing with `appl(prod, [args])` patterns, the `args` are always the second argument of the `appl`. If we look closely at the AsFix pattern for our `and`-terms, we notice that the `lhs` is the first element in this list of `args`. The extraction function can thus be simplified to the more efficient, but very type-unsafe and obfuscated:

```
ATerm extract_bool_lhs(ATerm t) {
  /* get arguments from AsFix "appl" */
  ATermList args = ATgetArgument(t, 1);

  /* lhs is the first of these args. */
  return ATgetFirst(args);
}
```

After all, this function would work on any ATerm function application that has (at least) two arguments, the first of which is a list with (at least) one element.

3.2.4 Maintenance issues

There are several fundamental maintenance issues inherent in the use of ATerms as a data structure implementation in hand-crafted tools.

- The esoteric art of writing down multi-line, quote-escaped string patterns and the subsequent substitution of parts of these patterns to contain the desired placeholders at the correct locations, is so error prone that it is almost guaranteed to go wrong at some point. Practical experience in the Meta-Environment has proven this many times over. Handcrafted ATerm-patterns proliferate through numerous versions of various tools, and after a while all sorts of "mysterious" bugs creep up where one tool cannot handle the output of another tool, or simply bails out reporting that deep down some part of an input term does not satisfy a particular assertion. Obviously, these errors are often due to pattern mismatches, misplaced placeholders, or ill-escaped quotes.

- Even if the patterns are written down correctly, or when the Level Two interface is used (which doesn't use ATerm-patterns), there is much work to be done when the application syntax changes.

  Suppose for example that we want to change the syntax of our boolean conjunction from infix notation:

  ```
  Bool "and" Bool -> Bool
  ```

  into prefix notation:

  ```
  "and" "(" Bool "," Bool ")" -> Bool
  ```

  Conceptually nothing has changed: we mean exactly the same arguments when we address them as `lhs`, `rhs`, and `result` terms in both productions. However, in the underlying parse tree the location of *all three* subterms has changed! This in turn means that all tools that manipulate, e.g. the `lhs` of boolean terms, have to be updated to reflect this structural change.

  In fact, there is hardly any room for flexibility with respect to changes in the syntax, unless the arguments happen to remain at their original position. Every tool based on the modified *application syntax* has to be updated.

- With such inflexibility with respect to the application syntax in mind, imagine what would happen if the structure of the parse trees (AsFix) *itself* were to change. Every tool based on the *representation* of parse trees would have to be updated to reflect the structural changes in the format. In our practical case of the Meta-Environment where we wanted to rid AsFix of some legacy constructs, this meant modification of virtually *every* tool — an arduous task indeed.

3.3   From syntax to API

Abstracting from implementation details about the facts that there is such a thing as a parse tree format and that this format in turn is implemented using ATerms, it is easy to name several operations a tool-builder would like, given a syntax definition.

As an example we consider the booleans again. Some of the typical things a tool-builder would like to be able to do given the boolean syntax are:

- Use a type definition for booleans (it is better to have a specific type `Bool` than to use the generic `ATerm` type);

- Create the basic booleans: `true` and `false`;

- Create a compound boolean term using basic and other compound boolean terms;

- Given an arbitrary term, test if it is a valid boolean term;

- Given an arbitrary boolean term, distinguish between a basic term and a compound term, e.g. by testing if it has a `lhs` or `rhs`;

- Extract the `lhs` and `rhs` of a given boolean term;

- Replace the `lhs` and `rhs` of a compound boolean term by another boolean term;

Obviously, this list is not exhaustive, but it does form a nice starting point. Fortunately, all the necessary information can be extracted from an SDF-definition of the grammar. In order to separate some concerns and simplify the generation framework, we split the process into two steps (see Figure 3.3). First, we extract all the necessary information from the SDF-definition, and store it in a convenient format. This step takes care of the parsing and analysis of the grammar. The second step takes the intermediate format and does the actual generation for a specific target language.
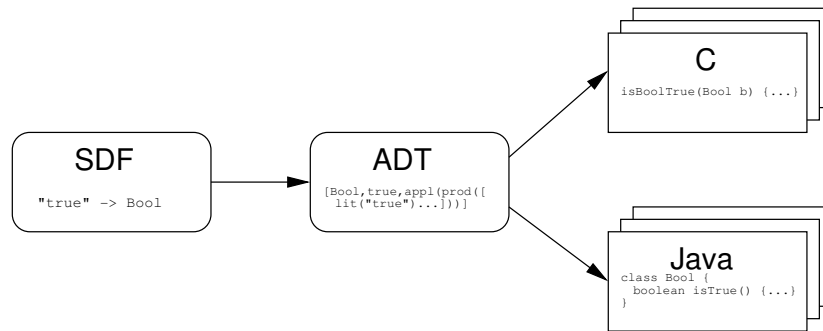
Figure 3.3: Generation scheme: from SDF to ADT to code

We call the intermediate format *annotated data type*, or ADT for short.

It holds the minimal amount of information for each syntax rule in the original SDF specification. In particular, for each rule we need:

- The *sortname* of the production. In our boolean syntax this is `Bool`;

- The *alternative* of the production. Our boolean syntax (from Figure 3.2) has five alternatives: `true`, `false`, `not`, `and`, `or`.

- The actual ATerm-*pattern* representation of the rule. In this pattern, each *field* (non-terminal in the syntax rule) is replaced by a typed placeholder containing the *sort* of the non-terminal and a *descriptive name*. For the `and` rule we could use `lhs`, and `rhs`, both of type `Bool`.

Since we are solving the maintenance problem of using ATerms as a data type representation, we decided we could very well use an ATerm to represent the elements of an ADT. The obvious advantage is that we get persistence (saving and loading of an ADT) for free, and we do not need to construct a domain specific language (with its own parser etc.) which would introduce undesired development-time overhead. Each entry in the ADT consists of the three elements *sortname*, *alternative*, and *term-pattern*, which we can easily represent as an ATerm-list. An entire ADT consists of nothing more than a list of such lists. Instead of using a list, each single entry could also have been

represented as a function with three arguments, but we opted for as little syntactic sugar in the entries as possible, to simplify development-time debugging. Remember that an ADT entry contains an ATerm pattern and they are hard enough to read, without the introduction of an extra function-symbol around them.

As an example of the concrete representation of an ADT entry, let us look at the boolean `and`. In this example, we know that the sortname of the production is `Bool`, the alternative is called `and`. There are two operands, `lhs` and `rhs`, both of type `Bool`. In the pattern we put typed placeholders `<lhs(Bool)>` and `<rhs(Bool)>` at the location of the corresponding non-terminals. Also, because this is a parse tree pattern, we have to allow layout (whitespace), which in this case can occur both after the non-terminal `lhs`, and after the literal `and`. The ADT entry thus becomes:

```
1  [Bool,
2   and,
3   appl(prod(
4    [cf(sort("Bool")),cf(opt(layout)),lit("and"),cf(opt(layout)),
5     cf(sort("Bool"))],cf(sort("Bool")),attrs([assoc(left)])),
6    [<lhs(Bool)>,<ws-after-lhs(Layout)>,lit("and"),
7     <ws-after-and(Layout)>,<rhs(Bool)>])]
```

- Line 1 contains the sortname: `Bool`

- Line 2 shows the alternative: `and`

- Lines 3–5 show the `prod` of the AsFix function application.

- Lines 6–7 show the `args` part. Clearly visible are the typed placeholders for `lhs` and `rhs`.

The two placeholders matching optional layout have the somewhat arbitrary names `ws-after-lhs`, and `ws-after-and`. Section 3.3.1 elaborates on the naming schemes used to generate legible, understandable names.

Given an ADT, which is generated from an SDF definition, but which could also come from any other source, we no longer need to worry about any SDF peculiarities, or parse tree specifics. Instead, we can concentrate on generating the desired functionality for a given target language. In this chapter we concentrate on describing the steps needed to produce legible, type-safe C code. Optimizations to the generated code can easily be obtained by removing type-safety checks, resulting in a more efficient production version of the code.

3.3.1   Deriving the ADT from a SDF specification

Now that we know what specific information we need in the ADT, how do we get it from the SDF definition? If we look back at our SDF definition of the booleans, we can derive two of the necessary elements immediately:

- The result *sort* of a syntax rule. It is explicitly mentioned at the end of each rule.

- The ATerm pattern. It can be constructed by following the exact same rules for constructing AsFix terms that the SDF normalizer uses.

This leaves us with the issue of coming up with a decent name for each *alternative* production of the same sort, and we still need to figure out a way to give *descriptive* names to the non-terminals in the grammar rule.

Naming the non-terminals   Given our SDF rule for the boolean `and`, can we derive a sensible name for each of the `Bool` non-terminals? The only information we have is our syntax rule:

```
Bool "and" Bool -> Bool {left}
```

If we use heuristics to call them e.g. `lhs` and `rhs`, what do we do when we find another syntax rule that has three, four or even more arguments? In syntax rules with only one non-terminal, we could default to using the sort name of that non-terminal. But in general, it is hard to come up with any kind of descriptive naming scheme. Keep in mind that most tool-builders will not really be happy if they are confronted with access functions that have arbitrarily complex names, or numbered arguments.

Instead of coming up with any kind of heuristic at all, we opted to use the *labeling* mechanism present in SDF, which allows grammar writers to label each non-terminal. This eliminates the need to invent a descriptive name altogether and provides an understandable link between items in a grammar rule and their generated access functions. Suppose we like the abbreviations `lhs`, and `rhs`, we could label the syntax rule for `and` to become:

```
lhs:Bool "and" rhs:Bool -> Bool {left}
```

Naming the alternatives   Similarly, we need a solution for the *alternative* name. In this case the literal `and` happens to be a name we could use. But what if there is no literal at all? Or if there are multiple literals in a production, which one should we pick? Should they be concatenated? What if the literal is some sort of baroque lexical expression (think of the C and `Java` symbols `&&` for conjunction). Again we are saved by SDF, which provides a way to annotate syntax rules. In fact, we re-use an annotation which is quite commonly used by SDF syntax writers to annotate the name of the *abstract syntax* node that corresponds to this particular syntax rule. Traditionally the `cons` annotation is used for this purpose. So, finally our `and` syntax rule becomes:

```
lhs:Bool "and" rhs:Bool -> Bool {left, cons("and")}
```

From which we can subsequently generate (e.g. C) type and function names as shown in Figure 3.4.

3.4   Code generation from ADT to C

3.4.1   Generated types and functions

For each sortname in an ADT, we generate the following items (which are further explained in Subsection 3.4.2):

```
            "true" -> Bool {cons("true")}
           "false" -> Bool {cons("false")}
       "not" arg:Bool -> Bool {cons("not")}
  lhs:Bool "or" rhs:Bool -> Bool {left, cons("or")}
 lhs:Bool "and" rhs:Bool -> Bool {left, cons("and")}



 Bool getBoolLhs(Bool arg) {
    if (isBoolAnd(arg)) {
       ...
    }
 }
```

| argument type | sortname | alternative name | argument name |
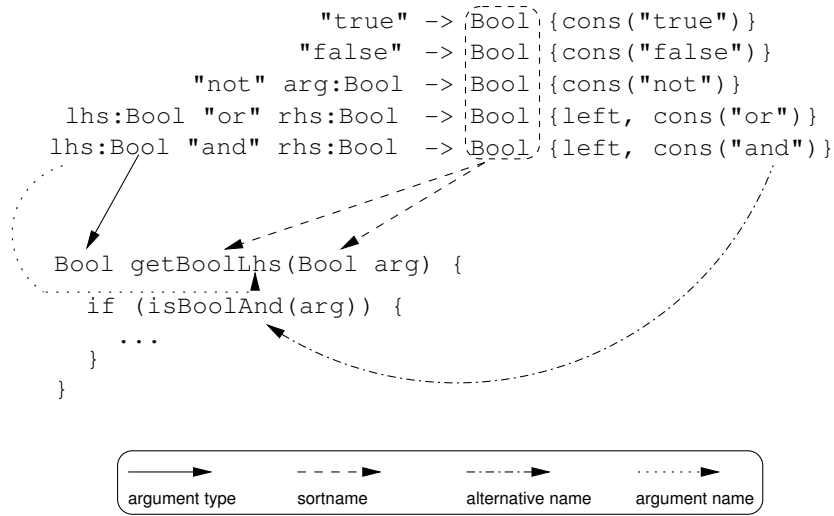
Figure 3.4: Using SDF elements to derive legible names.

- An opaque type definition to distinguish instances of this particular sort from other ATerms.

- Conversion functions `fromTerm` and `toTerm` to interface with generic ATerm functions, such as `ATreadFromFile`. These functions perform a type cast, and as such they form the entry and exit points to type-safety.

- A validity function to test whether an instance of a sort is indeed valid, i.e. that it indeed matches one of the ATerm-patterns defined as an alternative of this sort. This is useful to assert the validity of an externally acquired instance of this sort, e.g. if it has just been read from file.

- Constructor functions for each possible alternative for this sort to create instances from scratch.

- An equality function to test equality with another instance of this sort.

- For each alternative of the sort, an `isAlternative` function that checks if the current object is an instance of that particular alternative.

- For each field used in any of the alternatives of the sort, a `hasField` function that checks if the current object is an instance of an alternative that has that non-terminal.

- Similarly, a `getField` and `setField` method for each of the fields in a sort.

3.4.2 Implementation

In order to address the maintenance issues associated with the proliferation of ATerm-patterns, we decided it was a good idea to isolate them as much as possible from the actual code. This is achieved by generating a separate *dictionary* file containing all the ATerm patterns used by the library. This *dictionary* file declares a separate AFun variable for each ATerm function symbol, and an ATerm variable for each possible pattern. An initialization function is also generated which takes care of the proper initialization of all these variables, and the necessary calls to `ATprotect` to shield them from the built-in garbage collector. A verbatim dump of all the patterns is included in a comment section in the generated code, to provide debugging feedback when necessary. An example of a dictionary file can be found in Appendix B.2.

The actual implementation of the API functions is generated in its own C file, accompanied by a header file containing the signatures of all exported API functions. We show abridged snippets of the generated code. The header file is straightforward, containing merely the opaque type definition, and the declarations of the functions contained in the C file.

Opaque type definition Defining `Bool` to be a pointer to a non-existent type (in this case `struct _Bool`, hides the underlying ATerm representation from the point of view of API users. Instances of `Bool` can safely be passed around by functions, but any attempt to dereference such a pointer results in a compile time error.

```
typedef struct _Bool *Bool;
```

Term convertors These functions perform no real operation, but take care of the type casting between the generic ATerm type and the more specific `Bool`. They are needed as entry and exit points to type-safety when ATerm-Library functions such as `ATreadFromFile` are used, which yield an ATerm.

```
Bool BoolFromTerm(ATerm t) { return (Bool)t; }
ATerm BoolToTerm(Bool arg) { return (ATerm)arg; }
```

For improved efficiency, these functions could easily be replaced by macros which perform the exact same type cast. Unfortunately, this irrevocably kills type-safety, because macros are expanded during the pre-processor phase, without any form of type checking on the arguments of the macro.

Equality test Because ATerms are used as implementation, we get the trivial equality check based on memory address comparison for free. We only need to provide a type-safe wrapper around `ATisEqual`.

```
ATbool isEqualBool(Bool arg0, Bool arg1) {
  return ATisEqual((ATerm)arg0, (ATerm)arg1);
}
```

As with the convertor functions, the equality function can be replaced by a macro definition (with the same concerns about the loss of type-safety) for improved efficiency.

Validity test   Whenever an ATerm is acquired from an external source (e.g. by reading it from file) and is converted to `Bool`, programmers might like to assert that the term satisfies one of the alternatives for `Bool`. After all, any valid ATerm will happily be parsed by `ATreadFromFile` and subsequent conversion by `TermToBool` is done without any verification. The `isValidBool` function checks whether a given `Bool` argument is indeed an instance of one of the correct alternatives.

```
ATbool isValidBool(Bool arg) {
   if (isBoolTrue(arg)) { return ATtrue; }
   else if (isBoolFalse(arg)) { return ATtrue; }
   else if (isBoolNot(arg)) { return ATtrue; }
   else if (isBoolAnd(arg)) { return ATtrue; }
   else if (isBoolOr(arg)) { return ATtrue; }
   return ATfalse;
}
```

As checking the alternatives is expensive, the conversion functions themselves do not directly invoke `isValidBool`. Efficiency of the `BoolToTerm` function could be traded for even more robustness, by making it refuse to convert any ATerm that does not satisfy `isValidBool`.

Also note that the `isBoolX` functions perform a *shallow* match: they do not check the *arguments* of the alternative they test. For example, `isBoolAnd` does not check if its `lhs` and `rhs` are actually valid booleans. It merely tests if the term is an instance of the pattern for the `and` alternative. It would be possible to generate code that performs a *deep* match, again at the cost of a considerable efficiency hit.

Inspector   Inspecting a `Bool` to see if it is an instance of a specific alternative involves matching the argument against the pattern for that particular alternative. Because matching is expensive, the result of the most recent match is cached. This caching approach seems limited, but is useful when multiple subterms of the *same argument* are accessed. In these cases, sequences of `getBoolX` and `setBoolY` all reuse (cached) inspection results.

```
ATbool isBoolTrue(Bool arg) {
   static ATerm cached_arg = NULL;
   static int last_gc = -1;
   static ATbool cached_result;

   assert(arg != NULL);

   if (last_gc != ATgetGCCount() || (ATerm)arg != cached_arg) {
     cached_arg = (ATerm)arg;
     cached_result = ATmatchTerm((ATerm)arg, patternBoolTrue);
     last_gc = ATgetGCCount();
   }

   return cached_result;
}
```

Note that the cached ATerm is deliberately *not* protected from garbage collection. Doing so would result in all inspector functions holding on to references of ATerms

65

that could not be collected. These terms are potentially very large and the memory behaviour would become extremely unpredictable. We therefore opted for a solution where caching results are only valid *until the next garbage collection*. This is done by comparing the current garbage collection count with the same count at the time the cached result was calculated.

Query accessor   The query accessor checks if a given argument has a specific field. It is implemented by checking if the argument is an instance of any of the alternatives which has the required field.

```
ATbool hasBoolLhs(Bool arg) {
  if (isBoolAnd(arg)) { return ATtrue; }
  else if (isBoolOr(arg)) { return ATtrue; }
  return ATfalse;
}
```

Get accessor   The `get`ter is implemented much like the query accessor. It inspects the incoming argument to find out of which alternative it is an instance. Upon finding the right alternative, it returns the intended subterm by directly peeking into the ATerm representation.

If the production has but a single alternative, no testing is needed and the requested subterm can be returned immediately.

```
Bool getBoolArg(Bool arg) {
  return (Bool)
    ATelementAt((ATermList)ATgetArgument((ATermAppl)arg, 1), 2);
}
```

If there are multiple alternatives, each is tested in turn, until a single alternative remains, which must be the right one (since none of the other alternatives matched, and we assume a valid instance of one of the alternative productions).

```
Bool getBoolLhs(Bool arg) {
  if (isBoolAnd(arg)) {
    return (Bool)
      ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
  }
  else
    return (Bool)
      ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
}
```

An obvious optimization would be to detect if there are alternatives that have the requested field at the same location in the underlying ATerm representation. In this case, the `isBoolAnd` test is redundant, because both alternatives of `Bool` that have a `lhs`, have it at the exact same position. The condensed version would look much like the previous `getBoolArg` and would be much cheaper since it does not have to do any matching:

```
Bool getBoolLhs(Bool arg) {
  return (Bool)
    ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
}
```

Set accessor   The implementation of the `set`ter is again along the same path as the `get`ter and the inspector. The main issue here stems from the fact that ATerms are immutable. Consequently, all `set`ters need to be of a functional nature. This means that they cannot update the ATerm *in situ*, but instead need to construct a *new* ATerm, reflecting the desired change.

```
Bool setBoolLhs(Bool arg, Bool lhs) {
  if (isBoolAnd(arg)) {
    return (Bool)
      ATsetArgument((ATermAppl)arg, (ATerm)
                     ATreplace((ATermList)
                                ATgetArgument((ATermAppl)arg, 1),
                                (ATerm)lhs, 0), 1);
  }
  else if (isBoolOr(arg)) {
    return (Bool)
      ATsetArgument((ATermAppl)arg, (ATerm)
                     ATreplace((ATermList)
                                ATgetArgument((ATermAppl)arg, 1),
                                (ATerm)lhs, 0), 1);
  }

  ATabort("Bool has no Lhs: %t\n", arg);
  return (Bool)NULL;
}
```

As the construction of a new ATerm is expensive to begin with, the gain of eliminating the test for one of the alternatives (as implemented in the `get`ters) is minimal, which is why that particular optimization is omitted here. If no match was found after exhaustively testing all the alternatives, the operation is aborted.

3.5   Software engineering benefits in the Meta-Environment

Our main motivation to start this work has been the desire to make changes to AsFix, the parse tree format used by our tools in the Meta-Environment. Of particular interest is the dramatic size reduction (in terms of lines of code) of the various tools after refactoring them to use the new APIs.

This *apification* process consisted of the following stages:

- Reverse engineering the actual interfaces (and the corresponding term representations) that were needed in the Meta-Environment. This resulted in three ADTs:

  - A handwritten ADT for our parse tree format AsFix, closely matching the structure of the parse trees as they were produced by our parser;
  - An ADT for SDF, generated from our SDF definition of SDF;
  - An ADT for ASF, generated from a SDF definition of ASF.

As a result, we ended up with a specification for the main three formats used in the Meta-Environment. The main purpose of these specifications is to provide an authoritative description of the formats used, and to generate a consistent API as described in this chapter.

- Replacing all direct, untyped ATerm-manipulations by typed calls to the generated API. In practice, this often amounted to replacing large sections of code by concise snippets or even a single invocation of the API, clearly demonstrating the transition to a higher level of abstraction. The fact that we now operate in a typed domain means that we are able to effectively track type-related problems using the C compiler. We were even able to locate and fix a number of severe bugs in the *original* code that had not yet manifested themselves.

After the apification process was complete, we achieved a significantly higher level of maintainability of the code. We are now able to implement changes in the term representation, which was one of our major goals. Moreover the higher level of abstraction allows us to implement new functionality which would otherwise be much more time consuming and error prone. For example, it allowed us to quickly write an ASF-checker which traverses ASF-equations looking for occurences of uninstantiated variables.

In accordance with these subjective observations that the code has improved, is the Lines of Code (LOC) metric. Comparing versions just before and immediately after *apification*, we found out that we had been able to eliminate almost half of the (manually written) code. The LOC metrics have been summarized in Table 3.1.

The components are: the runtime environment of the ASF+SDF compiler (`asc-runtime`), the parse tree library (`libasfix`) and utilities (`asfix-tools`), the actual ASF+SDF compiler (`asf+sdf compiler`), a collection of ASF manipulation utilities (`asf-tools`), a `structure editor` for editing ASF+SDF specifications, an `evaluator` for evaluating (rather than compiling) ASF equations, and finally a repository for parse tables and parsed ASF+SDF specifications (`module-db`).

Component	Before (LOC)	After (LOC)	Reduction (%)
asc-runtime	2207	1752	21
libasfix	10419	2077	80
asfix-tools	466	603	-29
asf+sdf compiler	1866	1138	39
asf-tools	1303	589	55
structure editor	2861	1946	32
evaluator	4241	4009	5
module-db	1809	1244	31
Total	**25172**	**13358**	**47**

Table 3.1: Code Reduction

Understandably the biggest gain was achieved in `libasfix`, because most of this library is now generated from the SDF definition of SDF itself. Only some high level functionality that could not be generated remains in this library.

3.6   Conclusions

Generating access libraries from SDF definitions offers a simple, consistent way of developing and maintaining type-safe, efficient data types.

The application in the Meta-Environment of the techniques described in this chapter resulted in the elimination of a significant portion of handcrafted code. The effect of this elimination is amplified by the inherent nature of the affected code portions: hard to read and write, difficult to maintain, and in general very error prone to handle at all.

The result of our generational approach is a type-safe replacement for manually crafted libraries that provide access to compound data types implemented by ATerms. Even though several optimization opportunities have not yet been fully explored, the efficiency of the generated library is already comparable to its manually written predecessor.

More generically speaking, the approach presented in this chapter is applicable in situations where type safety is needed at a different (higher) abstraction level than is offered by the underlying data format. This is especially true in situations where representing the data at the higher abstraction level directly is unfeasible, e.g. due to performance issues.

3.7   Discussion

Although this chapter shows how API generation was used in a very specific context (generating ATerm manipulation code from an SDF specification), many of the issues encountered are not ATerm or SDF specific at all. In fact any generic data exchange formalism potentially suffers from the problem that generic manipulation functions are inherently type unsafe. For instance if we look at XML, DOM based libraries for manipulating XML in a generic way suffer from many of the same problems as the ATerm library. Recent techniques like XML data binding (discussed in Section 3.1.1) take the same approach as we do in this chapter by generating accessor and manipulation functions based on a signature description like XML schemas or DTDs.

In retrospect, one might ask why we ever developed code using direct ATerm manipulations in the first place. The answer lies partially in the power and attractiveness of working with ATerms. Because it is so easy to write a tool that uses simple ATerm patterns, several developers quickly started writing their own applications. Later, when some of the tools demonstrated a need for speed, parts of the now grown-but-not-restructured tools were rewritten to use the more efficient level two interface instead of the matching interface, mostly in the form of ad hoc restructuring, driven by the output of the `gprof` profiler. When prompted to implement changes in our parse tree format, we realized the era of direct ATerm manipulation had to end, and we had to find a structural solution to representing data types by ATerms, or we would be unable to maintain our toolset. Fortunately, the road of generating the access library as described in this chapter works very well in the Meta-Environment. Since the introduction of what has become known as APIGEN, we have been able to effectuate considerable changes in AsFix, and we have gained the ability to experiment with the format, and quickly see the results working in our tools.

3.8   Future work

The future work of this project falls into two categories: increasing the efficiency of the generated code and generalizing our approach to a wider application area.

Obvious optimizations include *inlining* (some of) the generated functions. By rewriting the functions as C macros, the overhead of a function call is removed. As noted before, the cost of this efficiency gain is that some type-safety is lost. This is due to the fact that C macro expansion is performed by a pre-compiler, which does not have access to type information and thus performs no type checks on macro arguments. A typical approach would be to generate type-safe functions in the development stage, and switch to the use of generated efficient macros for production code. This approach is comparable to the use of `assert` macro's that are completely eliminated in production versions of the software.

More interesting, however, are optimizations that take into account information about the structure of the underlying ATerm representation. During the generation phase information about common subterms and similarity between alternatives is assembled, which could be exploited to generate more efficient matching and selection code than the current `ATmatch` call, which is rather inefficient.

In a way the project described in this chapter can be seen as a case study in generative programming as described in Section 3.1.1. We want to extend this case study into a more generic approach. This approach will be based on a modular generic generator that takes a set of abstract data definitions and generates code for them. The code generator must be extensible with domain specific "modules" to generate extra functionality. These modules should not only be able to add extra functions when needed, but they should also be able to use *Aspect Oriented Programming* [84] to add functionality to functions that are generated by other modules.

For example, one of the more basic modules (the "accessor" module) could generate the actual data representation (for instance simple attributes in an object oriented setting) and accessors on this representation, while another module could add transparent persistency using a standard relational database by instrumenting all accessors.

The most important challenge in such an approach would be to create an environment where the threshold to create new generator modules is extremely low. In the ideal situation software developers could add new modules to the generator just as easy as to add new modules directly to the software system they are building.

Availability

Users interested in the more technical details (i.e. the actual implementation) or who would like to deploy the tools we described, are encouraged to download the latest distribution from:

```
http://www.cwi.nl/projects/MetaEnv/apigen
```

Part III

Structuring Component Interaction

C H A P T E R 4

ToolBus: the Next Generation

4.1 Generic Language Technology

Our primary interest is *generic language technology* that aims at the rapid construction of tools for a wide variety of programming and application languages. Its central notion is a *language definition* of some programming or application language.

The common methodology is that a language is identified in a given domain, that relevant aspects of that language are formally defined and that desired tools are generated on the basis of this language definition. This generative approach is illustrated in Fig. 4.1. Using a definition for some language L as starting point, a generator can produce a range of tools for editing, manipulating, checking or executing L programs.

Language aspects have to be defined, analyzed, and used to generate appropriate tooling such as compilers, interpreters, type checkers, syntax-directed editors, debuggers, partial evaluators, test case generators, documentation generators, and more.

Language definitions are used, on a daily basis, in application areas as disparate as Cobol renovation, Java refactoring, smart card verification and in application generation for domains including finance, industrial automation and software engineering. In the case of Cobol renovation, the language in question is Cobol and those aspects that are relevant for renovation have to be formalized. In the case of application generation, the language in question is probably new and has to be designed from scratch.

4.1.1 One Realization: the ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [86, 24] is an incarnation of the approach just described and covers both the interactive development of language definitions and the generation of tools based on these language definitions.

In this chapter we are primarily interested in the *software engineering aspects* of building such a system. Starting point is the ASF+SDF Meta-Environment as we had completed it in the beginning of the 1990's. This was a monolithic 200 KLOC Lisp program that was hard to maintain. It had all the traits of a legacy system and was the primary motivation to enter the area of system and software renovation.
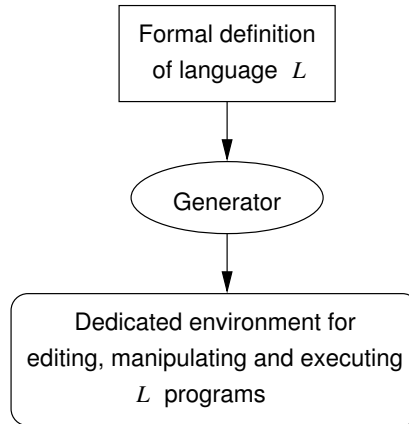
Figure 4.1: From language definition to generated programming environment

4.1.2   Towards a Component Based Architecture

We give a brief time line of the efforts to transform the old, monolithic, implementation of the Meta-Environment into a well-structured, component-based, implementation.

In 1992, first, unsuccessful, experiments were carried out to decompose the system into separate parts [8]. The idea was to separate the user-interface and the text editor from the rest of the system. The user-interface was completely re-implemented as a separate component and as text editor we re-used Emacs. In hindsight, we were unaware of the fact that we made the transition from a completely sequential system to a system with several concurrent components. Unavoidably, we encountered hard to explain deadlocks and race conditions.

In 1993, a next step was to write a formal specification of the desired system behavior [122] using PSF, a specification language based on process algebra and algebraic specifications [96]. Simulation of this specification unveiled other, not yet observed, deadlocks. Although this was clearly an improvement over the existing situation, this specification approach also had its limitations and drawbacks:

- The specification lacked generality. It would, for instance, have been a major change to add the description of a new component.

- The effort to write the PSF specification was significant and there was no way to derive an actual implementation from it.

In 1994, the first version of the ToolBus was completed [11, 13]. The key idea was to organize a system along the lines of a software bus and to make this bus programmable by way of a scripting language (TSCRIPT) that was based on ACP (Algebra of Communicating Processes, [16]). Another idea was to use a uniform data format (called ToolBus terms) to exchange data between ToolBus and tools. At the implementation level, TSCRIPTs were executed by an interpreter and communication between

tools and ToolBus took place using TCP/IP sockets. In this way, multi-language, distributed, applications could be built with significantly less effort than using plain C and sockets.

Based on various experiments [104, 51, 93, 56], in 1995 a new version of the Tool-Bus was designed and implemented: the Discrete Time ToolBus [12, 14, 15]. Its main innovations were primitives for expressing timing considerations (delay, timeout) and for operating on a limited set of built-in data-types (booleans, integers, reals, lists). The Discrete Time ToolBus has been used for the restructuring of the ASF+SDF Meta-Environment [27]. A first version was released in 2001 [24].

In the meantime, the exchange format has also evolved from the ToolBus terms mentioned above to ATerms [30]: a term format that supports maximal subterm sharing and a very concise, sharing preserving, binary exchange format. ATerms decrease memory usage thanks to sharing and they permit a very fast equality test since structural equality can be replaced by pointer equality thanks to the maximal subterm sharing.

Another line of development is the ToolBus Integrated Debugging Environment (TIDE) described in [105].

Today, beginning 2003, it turns out that the original software engineering goals that triggered the development of the ToolBus have been achieved and that the Meta-Environment can now be even further stretched than anticipated [36]. Therefore, it is time for some reflection. What have we learned from this major renovation project and what are the implications for the ToolBus design and implementation?

4.1.3   Plan of this Chapter

In Sect. 4.2 we discuss component coordination, representation and computation and introduce the ToolBus: our component coordination architecture. Following, in Sect. 4.3, we demonstrate some of the ToolBus-features by means of an example. In Sect. 4.4 we show how we used the ToolBus in the ASF+SDF Meta Environment to migrate from a monolithic to a distributed architecture. Then, in Sect. 4.5 we elaborate on the various issues that we would like to tackle in a next generation of the ToolBus. We conclude the chapter with an overview of the current status of our current implementation of this next generation ToolBus (Sect. 4.6) and some concluding remarks (Sect. 4.7).

4.2   The ToolBus Architecture

In [65] it was advocated that the overall architecture of a software system can be improved by separating *coordination* from *computation*. In addition to this, we also distinguish *representation* and use the following definitions:

- Coordination: the way in which program and system parts interact (using procedure calls, remote method invocation, middleware, and others).

- Representation: language and machine neutral data exchanged between components.
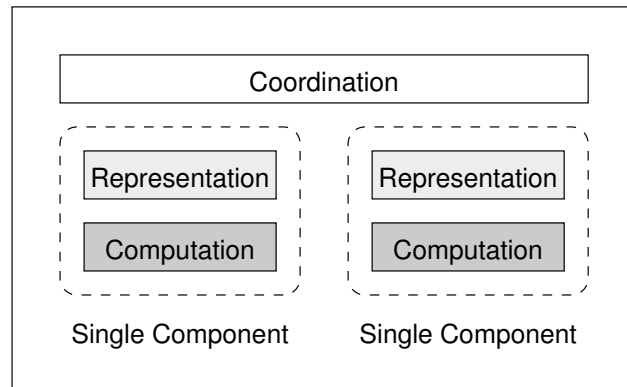
75

Figure 4.2: Separating coordination from computation

- Computation: program code that carries out a specialized task.

The assumption is now that *a rigorous separation of coordination, representation and computation leads to flexible and reusable systems.* This subdivision is sketched in Fig. 4.2. Our ToolBus approach follows this paradigm and is illustrated in Fig. 4.3.

The goal of the ToolBus is to integrate tools written in different languages running on different machines. This is achieved by means of a programmable software bus. The ToolBus coordinates the cooperation of a number of tools. This cooperation is described by a TSCRIPT that runs inside the ToolBus. The result is a set of concurrent processes inside the ToolBus that can communicate with each other and with the tools. Tools can be written in any language and can run on different machines. They exchange data by way of ATerms.

A typical cooperation scenario is illustrated in Fig. 4.4. A user-interface (UI) and a database (DB) are combined in an application. Pushing a button in the user-interface leads to a database action and the result is displayed in the user-interface. In a traditional approach, the database action is directly connected to the user-interface button by means of a call-back function. This implies that the user-interface needs some knowledge about the database tool and *vice versa*. In the ToolBus approach the two components are completely decoupled: pushing the button only leads to an event that is handled by some process in the ToolBus. This process routes the event to the database tool (likely via some intermediary process) and gets the answer back via the inverse route. This implies that the configuration knowledge is now completely localized in the TSCRIPT and that UI and DB do not even know about each others existence.

The primitives that can be used in TSCRIPTs are listed in Table 4.1.

4.3 An Example: the Address Book Service

To make the scenario from Fig. 4.4 more concrete, we describe the construction of an address book holding (name, address) pairs. Typical uses include creating a new
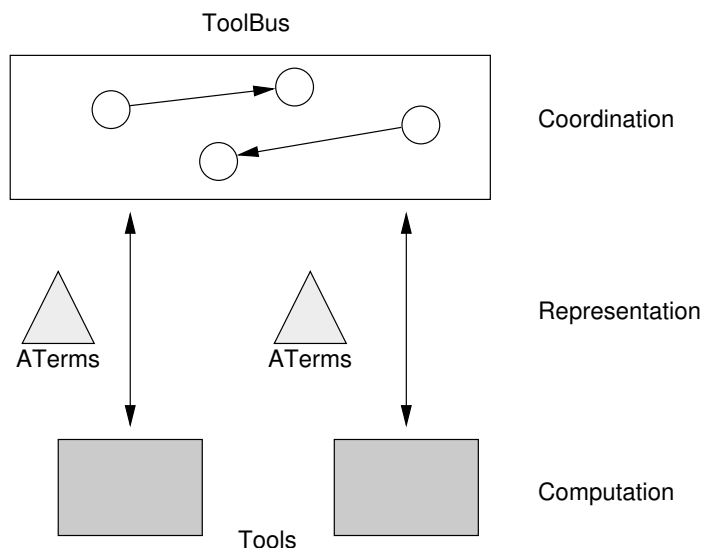
Figure 4.3: The ToolBus architecture
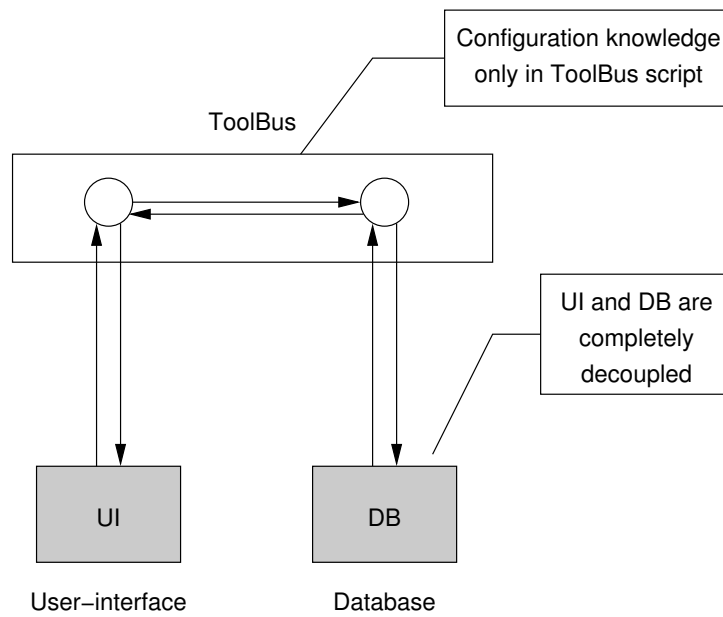
Figure 4.4: A typical cooperation scenario

Primitive	Description
`delta`	inaction ("deadlock")
`+`	choice between two alternatives (P_1 or P_2)
`.`	sequential composition (P_1 followed by P_2)
`*`	iteration (zero or more times P_1 followed by P_2)
`create`	process creation
`snd-msg`	send a message (binary, synchronous)
`rec-msg`	receive a message (binary, synchronous)
`snd-note`	send a note (broadcast, asynchronous)
`rec-note`	receive a note (asynchronous)
`no-note`	no notes available for process
`subscribe`	subscribe to notes
`unsubscribe`	unsubscribe from notes
`snd-eval`	send evaluation request to tool
`rec-value`	receive a value from a tool
`snd-do`	send request to tool (no return value)
`rec-event`	receive event from tool
`snd-ack-event`	acknowledge a previous event from a tool
`if ... then ... fi`	guarded command
`if ... then ... else ... fi`	conditional
	expressions
`||`	communication-free merge (parallel composition)
`let ... in ... endlet`	local variables
`:=`	assignment
`delay`	relative time delay
`abs-delay`	absolute time delay
`timeout`	relative timeout
`abs-timeout`	absolute timeout
`rec-connect`	receive a connection request from a tool
`rec-disconnect`	receive a disconnection request from a tool
`execute`	execute a tool
`snd-terminate`	terminate the execution of a tool
`shutdown`	terminate ToolBus
`attach-monitor`	attach a monitoring tool to a process
`detach-monitor`	detach a monitoring tool from a process

Table 4.1: Overview of ToolBus primitives

address, finding an address based on the name, etc. First we consider some aspects of the User Interface. An instance of the UI connects to the ToolBus and during the subsequent session, the user can:

create a new entry in the address book database;

delete an existing entry from the database;

search for an entry in the database;

update an existing entry in the database.

Each of these use cases can be described as a ToolBus process which, together with a process that explains how these use cases interact, form the ToolBus script describing our Address Book Service.

4.3.1 ToolBus Processes for the Address Book Service

The `ADDRESSBOOK` process tells the ToolBus that an instance of our `address-book` tool is to be executed, followed by a loop which invokes *one of* the processes CREATE, DELETE, SEARCH or UPDATE in each iteration. This construction, using the + operator ensures that at this level, the sub-processes can be regarded atomically. This means that for example no DELETE will happen during an UPDATE.

```
process ADDRESSBOOK is
let AB : address-book
in
  execute(address-book, AB?) .
  (
    CREATE(AB) + DELETE(AB) + SEARCH(AB) + UPDATE(AB)
  ) * delta
endlet
```

The operating system level details of starting the tool are defined in a separate section (one for each tool if multiple tools are involved):

```
tool address-book is {
  command = "java-adapter -class AddressBookService"
}
```

In this case, the ToolBus is told that our tool is written in Java, and that the main class to be started is called AddressBookService.

The `CREATE` process can be described as a ToolBus process as follows:

```
process CREATE(AB : address-book) is
let AID : int
in
  rec-msg(create-address) .
  snd-eval(AB, create-entry) .
  rec-value(AB, new-entry(AID?)) .
  snd-msg(address-created(AID))
endlet
```

The request to create a new address book entry is received and delegated to the tool, so it can update its state. In this case, our tool yields a unique id for reference to the new entry, which is returned as the result of the creation message. Note that communication between processes involves the matching of the arguments of `snd-msg` and `rec-msg`. The same holds for the communication between a process and a tool using `snd-eval` and `rec-value`. In all these cases, a *result variable* of the form `V?` gets a value assigned as the result of a successful match.

The DELETE process   differs only from the CREATE process in that it does not need a return value:

```
...
  rec-msg(delete-address(AID?)) .
  snd-do(AB, delete-entry(AID)) .
  snd-msg(address-deleted(AID))
...
```

The SEARCH process   in our example implements but a single query: finding an address book entry by name. It shows how different results from a tool-evaluation request can be processed in much the same way that different messages are handled. Upon receiving a `find-by-name` message from another process, this request is delegated to the tool. Depending on whether or not the entry exists in the database, the tool replies with a `found` or a `not-found` message, respectively. This result is then propagated to the process that sent the initial `find-by-name` message.

```
process SEARCH(AB : address-book) is
let
  Aid : int,
  Name : str
in
  rec-msg(find-by-name(Name?)) .
  snd-eval(AB, find-by-name(Name)) .
  (
    rec-value(AB, found(Aid?)) .
    snd-msg(found(Aid))
  +
    rec-value(AB, not-found) .
    snd-msg(not-found)
  )
endlet
```

The UPDATE process   is more interesting. It shows that each update of an address entry is *guarded*. A process wanting to update an entry first has to announce this fact by sending an `update-entry` message, before it can do one or more updates to the entry. It then finishes the update by sending an `update-entry-done` message. Because matching `snd-msg` and `rec-msg` messages are connected synchronously, it is not possible for one update transaction to interfere with another. After a sender and receiver of the `update-entry` message are connected, all other processes that want to send a `update-entry` message have to wait until the receiving process is ready to receive an `update-entry` message again. This message pair thus acts as a very

primitive locking scheme. More elaborate schemes are very well possible, but are not discussed in this chapter. Summarizing, the UPDATE process shows that *outside* the implementation of the address book service, we can enforce the order in which certain parts of the service are invoked, as well as mutual exclusion of some of its sections.

```
process UPDATE(AB : address-book) is
let
  AID : int,
  Name : str,
  Address : str
in
  rec-msg(update-entry(AID?)) .
  ( rec-msg(set-name(Name?)) .
    snd-do(AB, set-name(AID, Name))
  + rec-msg(set-address(Address?)) .
    snd-do(AB, set-address(AID, Address))
  ) *
  rec-msg(update-entry-done(AID))
endlet
```

4.3.2 ToolBus Process for the User Interface

Because users can connect at any time to the ToolBus to start a session with the Address Book Service, the ToolBus itself does not execute instances of the UI (as it did with the address book tool). Instead UITool instances can connect, make zero or more requests to the service, and disconnect at their convenience. A ui tool is declared to exist, but no operating system level details are provided. The following definition of the UI process shows how UI requests for the creation of a new entry and a name-change can be realized:

```
tool ui is { /* the ToolBus does not execute ui-instances */ }

process UI is
let
  UITool : ui,
  AID : int,
  Name : str
in
  rec-connect(UITool?) .
  (
    rec-event(UITool, create-address) .
    snd-msg(create-address) .
    rec-msg(address-created) .
    snd-ack-event(UITool, create-address)
  +
    rec-event(UITool, update-name(AID?, Name?)) .
    snd-msg(update-entry(AID)) .
    snd-msg(set-name(Name)) .
    snd-msg(update-entry-done(AID)) .
    snd-ack-event(UITool, update-name(AID, Name))
  +
    ... /* more UI requests */
  )
  * rec-disconnect(UITool)
endlet
```

81
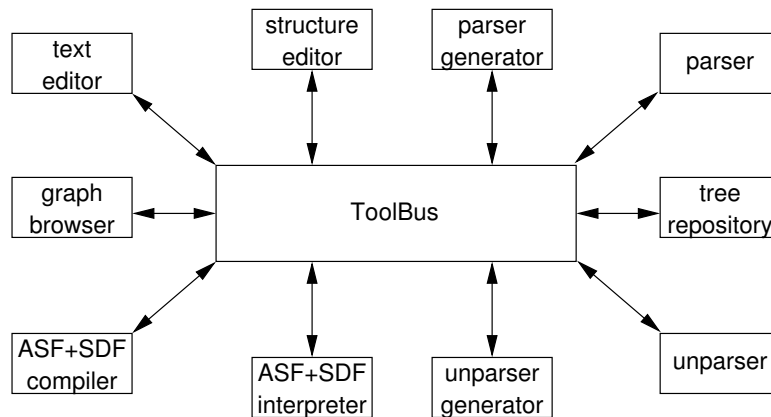
Figure 4.5: Architecture of the ASF+SDF Meta-Environment

Language	KLOC†	Generated KLOC	Total KLOC
ASF+SDF	12	170 (C)	
C	80††		
Java, Tcl/Tk	5		
Makefiles, etc	5		
TSCRIPT	5		
Total LOC:	107	170	277
TSCRIPT	4.6%		1.8%

† Kilo Lines of Code excluding third party code such as emacs, dot, and the like.
†† This includes 10 KLOC (C code) for the ToolBus implementation itself.

Table 4.2: Facts concerning implementation languages

4.4 Application to the ASF+SDF Meta-Environment

As explained in Sect. 4.1.2, the ToolBus has been used to restructure the ASF+SDF Meta-Environment. It consists of a cooperation of 27 tools ranging from a user-interface, graph browser, various editors, compiler and interpreter, to a parser generator and a repository for parse trees. A simplified view is shown in Fig. 4.5.

Our insight can be further increased by considering some statistics. Table 4.2 shows the relative sizes of the various implementation languages used in the Meta-Environment. In the column *language* the various languages are listed. In column *KLOC* the size (in Kilo Lines Of Code) is given for each language. The result is 107 KLOC for the whole system of which 4.6% are TSCRIPTs. If we consider the fact that ASF+SDF specifications are compiled to C code, another view is possible as well: 12 KLOC of ASF+SDF generates 170 KLOC of C code. Taking this generated code into account, the total size of the whole system amounts to 277 KLOC of which 1.8% are

Primitive	Number of occurrences
process definitions	104
tool definitions	27
. (sequential composition)	4343
+ (choice)	341
* (iteration)	243
|| (parallel composition)	3
snd-msg	555
rec-msg	541
snd-note	100
rec-note	24
snd-do/snd-eval	220
rec-event	56
create	58

Table 4.3: Facts concerning TSCRIPT primitives

TSCRIPTs. This is compatible with the expectation that TSCRIPTs are relatively small and form high-level "glue" to connect much larger components.

Part of the generated C code is currently done by ApiGen [79]. This is an API generator which takes an SDF grammar as input and generates a C library which gives type-safe access to the underlying ATerm representation of the parse trees over this grammar.

Another conclusion from these facts is that low-level information for building the software (makefiles and configuration scripts) are of the same size as the high level TSCRIPTs. This points into the direction that the level of these build scripts should be raised. This conclusion will, however, not be further explored in this chapter.

Another view is given in Table 4.3 where the frequency of occurrence of TSCRIPT primitives is shown. Clearly, sequential composition (.) is the dominant operator and sending/receiving (snd-msg, rec-msg) messages is the dominant communication mechanism, followed by communication with tools (snd-do, snd-eval). It may be surprising that parallel composition (\|\|) is used so infrequently. However, one should be aware that at the top level all ToolBus processes run concurrently and that \|\| is only used for explicit concurrency inside a process. The level of concurrency is therefore approximately 100 (104 process definitions and 3 explicit \|\| operators).

Empirical evidence shows that:

- The ToolBus-based version of the ASF+SDF Meta-Environment is more flexible as illustrated by the fact that clones of the Meta-Environment start to appear for other languages than ASF+SDF. Examples are Action Semantics [100] and Elan [38].

- Various components of the ASF+SDF Meta-Environment are being reused in other projects [51, 19].

4.5 Issues in a Next-Generation ToolBus

The ToolBus has been used in various applications of which the Meta-Environment is by far the largest. Some of the questions posed by our users and ourselves are:

- I find it difficult to see which messages are requests and which are replies; can you provide support for this? See Sect. 4.5.1.

- If a tool crashes, what is the best way to describe the recovery in the TSCRIPT? See Sect. 4.5.2.

- I have huge data values that are exchanged between tools and the ToolBus becomes a data bottleneck; can you improve this? See Sect. 4.5.3.

- The ToolBus and tools are running as separate tasks of the operating systems. Would it not be more efficient to run ToolBus and tools in a single task? See Sect. 4.6.

4.5.1 Undisciplined Message Patterns

The classical pattern of a remote procedure call is shown in Fig. 4.6: a caller performs a call to a callee. During the call the caller suspends execution and the callee executes until it has computed a reply. At that point in time, the caller continues its execution.
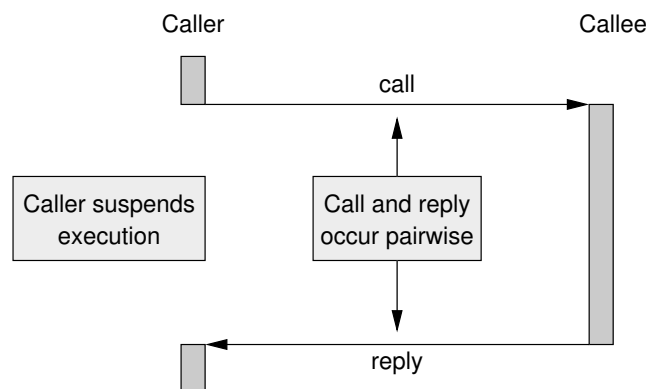
Figure 4.6: Communication pattern for remote procedure call

Compare this simple situation with general message communication as shown in Fig. 4.7: the caller continues execution after sending a message *msg1* to *Callee1* and may even send a message *msg2* to *Callee2*. At a certain point in time *Callee2* may send message *msg3* back to *Caller*. In this case, the three parties involved continue their execution while messages are being exchanged and there is no obvious pairing of calls and replies.

84
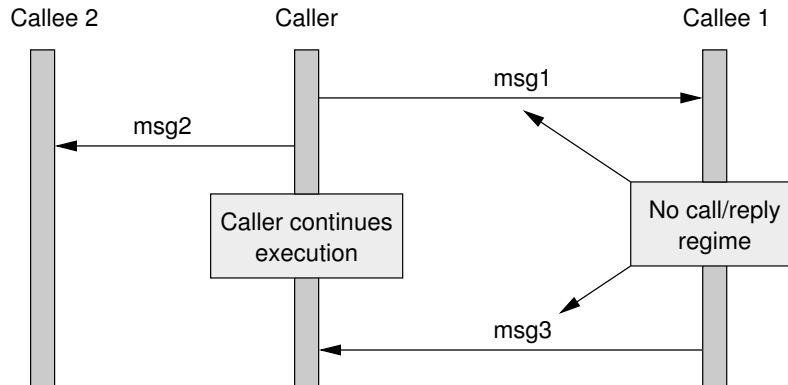
Figure 4.7: Communication pattern for general messages

In the ToolBus case, a `snd-msg` and a `rec-msg` can interact with each other if their arguments match. A typical sequence is:

Process A:
```
snd-msg(calculate(E)) .
... other actions ...
rec-msg(value(E, V?))
```

Process B:
```
rec-msg(calculate(E?)) .
... actions that compute value V ...
snd-msg(value(E, V))
```

What we see here is that a form of call/reply regime is encoded in the messages: process B returns the value V that it has computed as `snd-msg(value(E, V))`. The E is completely redundant but serves as identification for process A to which message this is an answer.

The call/reply regime is thus implicitly encoded in messages. This makes error handling harder (which reply did not come?) and makes the TSCRIPTs harder to understand. This is particularly so, since unstructured combinations of `snd-msg`/`rec-msg` and sequential composition, choice, iteration and parallel composition are allowed.

The only solution for the above problems is to limit the occurrences of `snd-msg` or `rec-msg` in such a way that a form of very general call/reply regime is enforced. Our approach is to syntactically enforce that `snd-msg`/`rec-msg` or `rec-msg`/`snd-msg` may only occur in (possibly nested) pairs and that in between arbitrary operations are allowed. In fact, the matching `snd-msg` or `rec-msg` may be an arbitrary expression provided that all its alternatives begin with a matching `snd-msg` or `rec-msg`.

We replace thus

```
snd-msg(req(E)) . arbitrary process expression .
rec-msg(ans(A?))
```

by the syntactic construct

```
snd-msg(req(E)) { arbitrary process expression }
```

```
rec-msg(ans(A?))
```

and also allow more general cases like:

```
snd-msg(req(E)) { arbitrary process expression }
( rec-msg(ans(A?)) + rec-msg(error(M?) )
```

It is an interesting property of Process Algebra that every process expression can be normalized to a so-called *action prefix form*: a list of choices where each choice starts with an atomic action. An action prefix form has the following structure: $a_1.P_1$ + $a_2.P_2$ +...+ $a_n.P_n$. Using this property we can formulate the most general constraint that we impose on occurrences of `snd-msg` and `rec-msg`. Consider P_1 { Q } P_2 and let P_1' and P_2' be the action prefix forms of P_1 and P_2, respectively. Our requirement is now that each choice in P_1' starts with a `snd-msg` and each choice in P_2' with a `rec-msg`, or *vice versa*. Note that this constraint can be checked statically.

4.5.2 Exception Handling

Exception handling is notorious for its complexity and impact on the structure of program code. The mainstream exception handling approach as used in, for instance, Java associates one or more exception handlers with a specific method call. If the call completes successfully, the handlers are ignored. If the call raises an exception, it is checked whether this exception can be handled locally by one of the given handlers. If not, the exception is propagated to the caller of the current code. This model does, however, not work well in a setting where multiple processes are active and the occurrence of an exception may require recovery in several processes.

Local Exception Handling   We start with the simpler case of local error handling and introduce the *disrupt operator* (>>) proposed in LOTOS [43]. A process algebra variant of this operator is described in [55]. It has the form `P >> E`, where `P` describes the normal processing and `E` the exceptional processing. It adds the exception `E` as alternative to each atomic action in `P`. If the action prefix form of `P` is

$$a_1.P_1 + a_2.P_2 + \ldots + a_n.P_n$$

then

$$\texttt{P >> E} \equiv (a_1 + \texttt{E}).(\texttt{P}_1 \texttt{>> E}) + \ldots + (a_n + \texttt{E}).(\texttt{P}_n \texttt{>> E})$$

Global Exception Handling   Global exception handling in distributed systems is a very well-studied subject from the perspective of crash recovery and transaction management in distributed databases. An overview of rollback-recovery protocols in message-passing systems is, for instance, given in [57].

In the context of system reliability, the notion of a *recovery block* has been introduced by Randell [109]. Its purpose was to provide several alternative algorithms for doing the same computation. Upon completion of one algorithm, an acceptance test is made. If the test succeeds, the program proceeds normally, but if it fails a rollback is made to the system state before the algorithm was started and one of the alternative algorithms is tried. In [85] this idea is applied to backtracking in string processing languages. It turns out that the preservation of the system state can be done efficiently by only saving updates to the state after the last recovery point.

Recovery blocks also form the basis for Coordinated Atomic Actions described in [130]. Recovery blocks are intended for the error recovery in a single process. They can be generalized to *conversations* between more than one process: several processes can enter a conversation at different times but they can only leave it simultaneously, when all participating processes satisfy their acceptance test. In case one participant fails to pass its test, each participant is rolled back to the state when it entered the conversation.

We are currently studying this model since it can be fit easily in the ToolBus framework and seems to solve our problem of global exception handling. It is helpful that a backtrack operator similar to the one described in [85] has also been described for Process Algebra [17]. What remains to be studied is how the recovery of *tools* has to be organized. Most likely, we will add a limited undo request to the tool interface to recover from the last few operations carried out by a tool.

4.5.3   Call-By-Value Versus Call-By-Reference

Background   The concepts of call-by-reference and call-by-value are well-known in programming languages. They describe how an actual parameter value is transmitted from a procedure call to the body of the called procedure. In the case of call-by-reference, a *pointer* to the parameter is transmitted to the body. Call-by-reference is efficient (only a pointer has to be transmitted) and the parameter value can be changed during execution of the procedure body (via the pointer). In the case of call-by-value, a *physical copy* of the parameter is transmitted to the procedure body. Call-by value is less efficient for large values and does not allow the called procedure to make changes to the parameter value in the calling procedure.

These considerations also apply to value transmissions in a distributed setting, with the added complication that values can be accessed or modified by more than one party. Call by reference (Fig. 4.8) is efficient for infrequent access or update. It is the prevalent mechanism in, for instance, CORBA [49]. However, uncontrolled modifications by different parties can lead to disaster.

Call-by-value (Fig. 4.9) is inefficient for large values and any sharing between calls is lost. To us, this is of particular interest, because we need to preserve sharing in huge parse trees. In the case of Java RMI [112], value transmission is achieved via serialization and works only for communication with other Java components. Using IIOP [103] communication with non-Java components is possible.

Current ToolBus approach   Currently, the ToolBus provides a transport mechanism based on call-by-value as shown in Fig. 4.10(a). It is transparent since the transmitted
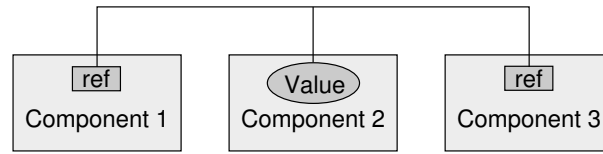
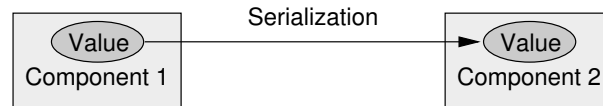Figure 4.8: Call-by-reference in a distributed application

Figure 4.9: Call-by-value in a (Java-based) distributed application

values are ATerms (see Sect. 4.1.2) that can be exchanged with components written in any language. Since pure values are exchanged, there is no need for distributed garbage collection.

Note that the call-by-reference model can easily be mimicked in the ToolBus. For instance, one tool can maintain a shared database and can communicate with other tools using record keys and field names so that only the values of record fields have to be exchanged (as opposed to complete records or even the complete database). In this way the access control to the shared database can be spelled out in detail and concurrency conflicts can be avoided. This solves one of the major disadvantages of the pure call-by-reference model in a distributed environment.

The downside is, however, that the ToolBus becomes a data bottleneck when huge values really have to be transmitted between tools. Currently, two workarounds are used. A first workaround is to get temporary relief by sending compressed values rather than the values themselves. A second workaround is to store the large value in the filesystem and to send a file name rather than the file itself. It does scale, but it also creates an additional inter-tool dependency and assumes that both tools have access to the same shared file system.

We will now first discuss how related frameworks handle call-by-reference and then we come back to implications for the ToolBus design. In particular, we will discuss channel-based transmission as already shown in Fig. 4.10(b).

4.5.4   Related Frameworks: Java RMI, RMI-IIOP and Java IDL

Given our needs and desires for a next generation ToolBus it is interesting to see what other solutions are applied in similar projects. In this section, we briefly look at three related mechanisms:

- Java Remote Method Invocation (RMI) which connects distributed objects written in Java;

ToolBus

Tools

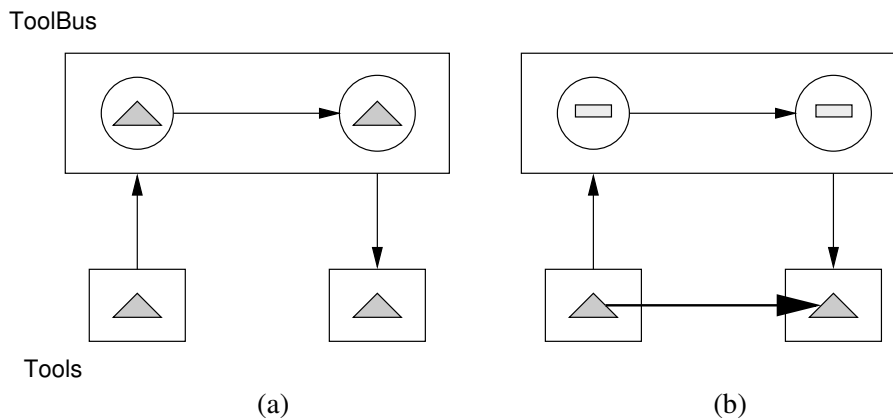<center>(a)   (b)</center>

Figure 4.10: Value-based (a) versus channel-based (b) transmission in the ToolBus

- Java RMI over Internet Inter-ORB Protocol (IIOP) which is like RMI, but uses IIOP as the underlying protocol;

- Java IDL which connects Java implementations of CORBA interfaces.

Java RMI   Java Remote Method Invocation is similar to the ToolBus architecture in the sense that it connects different tools, possibly running on different machines. It differs from the ToolBus setting because it is strictly Java based: only components written in Java can communicate via RMI.

For components to work together in RMI, first a *remote interface* is established. This is a Java interface that has a "real" implementation in the tool (or *server*) and a "stub" implementation on the client sides (Fig. 4.11). The interface is written by the programmer as opposed to the generated interfaces in a ToolBus setting where they are derived from the communication patterns found in the ToolBus script. The stubs in the RMI setting are then generated from this Java interface using `rmic`: the RMI compiler. Stubs act as a client-side proxy, delegating the method call via the RMI system to the server object. In RMI, any object that implements a remote interface is called a *remote object*.

In RMI, arguments to or return values from remote methods can be primitive data (e.g. `int`), remote objects, or *serializable* objects. In Java, an object is said to be serializable if it implements the `java.util.Serializable` interface. Both primitive data and serializable objects are passed by value using Java's object serialization. Remote objects are essentially passed by reference. This means that changes to them are actually performed on the server, and updates become available to all clients. Only the behavior that was defined in the remote interface is available to the clients.

RMI programmers should be aware of the fact that any parameters, return values and exceptions that are not remote objects are passed by value. This makes it hard to understand when looking at a system of RMI objects exactly which method calls will
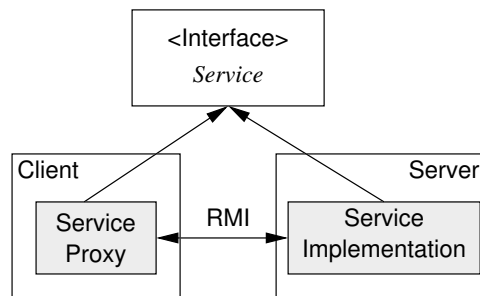
Figure 4.11: Client-server model in RMI framework.

result in a *local* (i.e. client side) state change, and which will have *global* (server side) effect.

Consider, again, our address book example. If the AddressBookService is implemented as a remote object in RMI, then client-side invocations of the `setAddress` method will cause a *global* update. If, on the other hand, the AddressBookEntries are made *serializable* and instances of this class are returned as the result of a query to the AddressBookService, then updates on these instances will have a *local* state change only.

Finally, before two RMI components can connect, the server side needs to register itself with an `rmiregistry`, after which the client needs to explicitly obtain a reference to the (remote) server object.

Java RMI over IIOP   By making RMI programs conform to some restrictions, they can be made available over the Internet Inter-ORB Protocol (IIOP). This means that functionality offered by the RMI program can be made available to CORBA clients written in any (CORBA supported) language. The restrictions are mostly namespace oriented: programmers need to take special care not to use certain names that might collide with CORBA generated names, but some reservations should also be made regarding sharing preservation of object references. References to objects that are equal according to the `==` operator in one component, need not necessarily be equal in a remote component. Instead the `equals` method should be used to discern equality.

RMI over IIOP is best used when combining several Java tools for which the programmer would like to use RMI, and some tools written in another CORBA-supported language need to use (some of) the services provided by the Java tools. The component's interface is established by writing a Java interface, just as in plain RMI.

Java IDL   Apart from Java RMI, which is optimized for connecting components that are all written in Java, there is also a connection from Java to CORBA using the Java Interface Definition Language (IDL). This alternative to Java RMI is for Java programmers who want to program in the Java programming language, based on interfaces defined in the CORBA Interface Definition Language.

90

	ToolBus	RMI	RMI-IIOP	Java IDL
Architecture	Component coordination	Client Server	Client Client	Client Server
Interface	TSCRIPT	Java Interface	Java Interface	IDL
GC	yes	yes	no	no
parameters / return values	by-value	local: by-value remote: by-ref	local: by-value remote: by-ref	depends on signature
language	any with TB adapter	only Java	CORBA objects if interface in Java	any with IDL binding
component coordination	yes	no	no	no

Table 4.4: Related architectures: a feature overview.

Using this bridge, it becomes possible to let Java components communicate with CORBA objects written in any language that has Interface Definition Language (IDL) mappings.

Instead of writing a Java interface as is done in RMI, in Java IDL the definition is written in IDL: a special purpose interface language used as the base for CORBA implementations. This IDL definition is then used to generate the necessary *stubs* (client side proxies to delegate method invocations to the server) and *skeletons*, *holder* and *helper* classes (server side classes that hide low-level CORBA details).

Feature summary Table 4.4 shows some of the similarities and differences in Tool-Bus, RMI, RMI-IIOP and Java IDL.

- RMI, RMI-IIOP and Java IDL make an explicit distinction between *client* and *server* sides of a set of cooperating components. In the ToolBus setting all components are considered equal (and none are more equal than others).

- In RMI and RMI-IIOP, the programmer writes a Java interface which describes the component's incoming and outgoing method signature, from which stubs and skeletons are generated. In Java IDL a CORBA interface is written. In the ToolBus setting, these signatures are generated from the ToolBus script which describes much more of the component's behavior in terms of method call interaction, rather than just method signatures.

- The ToolBus takes care of garbage collection of the ATerms that are used to represent data as it is sent from one component to another. RMI allows programmers access to Java's Distributed Garbage Collection API. In RMI-IIOP and Java IDL however, this is not possible, because the underlying CORBA architecture is used, which does not support (distributed) GC, but places this burden entirely on the developer.

- In the ToolBus all data is sent by-value. RMI and RMI-IIOP use both pass-by-value and pass-by-reference, depending on whether the relevant data is serializable (it is a primitive type, or it implements `Serializable`) or is a remote

object. In Java IDL the components abide by IDL prescribed interfaces. Determination of whether a parameter is to be passed by-value or by-reference is made by examination of the parameter's formal type (i.e. in the IDL signature of the method it is being passed to). If it is a CORBA *value type*, it is passed by-value. If it is an ordinary CORBA *interface type* (the "normal" case for all CORBA objects), it is passed by-reference.

- The ToolBus allows components in any language for which a ToolBus adapter exists. Programming languages such as C and Java are supported, but adapters also exist for a wide range of languages and applications, including e.g., Perl, Prolog, MySQL, Tcl and ASF+SDF. In RMI, only Java components can be connected; in RMI-IIOP the service is implemented in Java, its functionality (client-side) is available to CORBA clients. The Java IDL framework is fully CORBA compliant.

- Only the ToolBus has coordination support for component interaction. In the three other cases any undesired sequence of incoming and outgoing method calls will have to be prohibited by adding code to the component's internals. Whereas RMI, RMI-IIOP and Java IDL just perform the *wiring* that connects the components, the ToolBus also provides *workflow* support. In relation to this workflow support, it would be interesting to compare the ToolBus to related workflow description languages such as the Business Process Modeling language [44] and the Web Services Description Language [123].

Implications for the ToolBus Approach   To overcome the problems of value-based transmission, we envisage the introduction of channels as sketched in Fig. 4.10(b). This model is inspired by the second workaround mentioned at the end of Sect. 4.5.3 and is completely transparent for the user.

The idea is to stick to the strict call-by-value transmission model, but to implement the actual value transmission by data communication between sending tool and receiving tool thus offloading the ToolBus itself. Via the ToolBus, only an identification of the data value is transmitted between sender and receiver. The downside of this model is that it introduces the need for distributed garbage collection, since a value may be distributed to more than one receiving tool and the sender does not known when all receivers have retrieved their copy. Adding expiration times to values or reference counting at the ToolBus level may solve this problem.

4.6   Current Status

The current ToolBus was first specified in ASF+SDF and has then been implemented manually in C. Its primary target was the renovation of the ASF+SDF Meta-Environment.

The next generation ToolBus is being implemented in Java and aims at supporting larger applications such as, for instance, a multi-user game site like `www.gamesquare.nl` with thousands of users. High performance and recovery of

crashed game clients are then of paramount importance. The Java implementation is organized in such a way that the actual implementation of tools is as much hidden as possible. This is achieved by introducing the interface `ToolInterface` that describes the required ToolBus/tool interaction. This interface can be implemented by a variety of classes:

`ClassicToolBusTool:` this implements the ToolBus/tool communication as used in current applications. The tool is executed as a separate, operating system level, process and the ToolBus/tool communication is achieved using sockets.

`JavaTool:` this implements a new model that addresses one of the issues mentioned in Sect. 4.5: when ToolBus and tool run on the same computer *and* the tool is written in Java, then the tool can be loaded dynamically in the executing ToolBus, e.g. using Java Threads. In this way, the overhead of interprocess communication can be eliminated.

`JavaRMITool:` this is a special case where a Java tool runs on another computer.

`SOAPTool:` this implements communication with a tool that has a SOAP interface.

A prototype implementation is under development that allows experimentation with the features mentioned in this chapter.

4.7 Concluding Remarks

In this chapter we have reflected on our experiences over the past years with the use of the ToolBus as a means to refactor a previously monolithic system: the ASF+SDF Meta Environment. This real test case of the ToolBus has taught us some of its shortcomings: its data bottleneck in case very large data items are sent using pass-by-value, maintenance issues related to undisciplined message passing and questions such as how to deal with exceptions caused by e.g. crashing tools.

Some of the ideas we showed in this chapter could be implemented by changing or extending the TSCRIPT (e.g. to implement a call-reply regime as discussed in Sect. 4.5.1), others will also require extending the ToolBus and the tool-adapters (e.g. to detect crashed tools in combination with exception handling as discussed in Sect. 4.5.2).

We have also studied some related ideas and frameworks and we are now in a position where we have a new prototype of the ToolBus in Java, with a very open structure which allows for all sorts of experiments and case studies based on the experience we have with the existing ToolBus and the ideas presented in this chapter.

Acknowledgments

C H A P T E R 5

My Favorite Editor
Anywhere

5.1   Introduction

Many applications such as email clients, instant messengers, web browsers, and programming environments provide editing facilities. Full fletched, off-the-shelf editing solutions such as GNU Emacs [111] and Vim [98] are readily available, but many application developers still choose to write their own editing software. Some utility libraries (e.g. Java's JFC/Swing library) contain partial solutions in the form of reusable editing widgets. Still, developing and extending your own editor to encompass the feature richness common in mature text editors is far from a *rapid* software engineering exercise.

Offering a single built-in editor obviously also limits the user to this editor. This poses no problem as long as the editing sessions are brief, e.g. during login or password entry. However, when the editor is used for lengthy (programming) sessions, being forced to use the keybindings dictated by an editor that is not your personal favorite can easily lead to frustration.

This chapter describes how we reuse and integrate existing editors in a programming environment. Although our implementation is based on needs we have in our own environment, both the idea and most of the implementation can carry over to other projects. Basically, projects that need editing support for structured documents and where interactivity with these editing sessions is desirable, could benefit from the architecture we describe.

The structure of this chapter is as follows. This section continues with some background, motivation and discussion of related work. Section 5.2 describes how we coordinate simultaneous editing sessions, and we show the architecture used to deal with various editors. Section 5.3 describes some of the implementation details of the architecture: the MULTIPLEXER which orchestrates simultaneous editing sessions and the *glue* that is needed between the MULTIPLEXER and the various editor instances. We conclude with a summary of our contribution and a discussion of ideas for future work in Section 5.4.
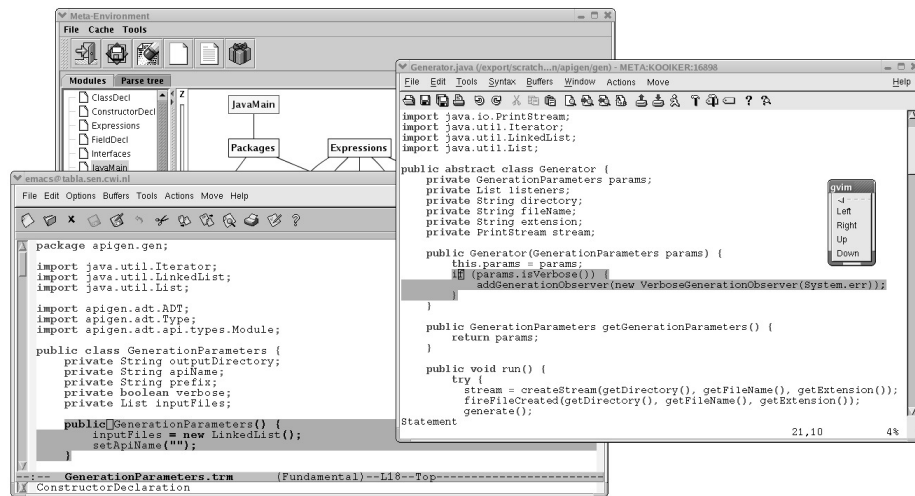
Figure 5.1: GNU Emacs and Vim used simultaneously by an IDE.

5.1.1 Background

The ASF+SDF Meta-Environment [24, 86] is a programming environment generator: given a language definition consisting of a syntax definition (grammar) and tool descriptions (using rewrite rules) a language specific environment is generated. Figure 5.1 shows a screenshot of the ASF+SDF Meta-Environment. A language definition typically includes such features as pretty printing, type checking, analysis, transformation and execution of programs in the target language. The ASF+SDF Meta-Environment is used to create tools for domain-specific languages and for the analysis and transformation of software systems.

The ASF+SDF Meta-Environment is used in several academic [28], industrial [25], and financial projects [87, 117]. Presently, the ASF+SDF Meta-Environment is intensively used in the software renovation oriented research project CaLCE: "Computer-Aided Life Cycle Enabling". This project is financed by the Dutch Ministry of Economic Affairs and aims at the development of tooling to improve the overall quality of systems deployed in the financial setting.

5.1.2 Related work

Some applications (e.g. the KDE and Gnome window managers) allow the configuration of a *foreign* editor. Whenever a body of text needs to be edited, the application executes the configured editor and waits for the user to complete the editing session. During this session, there is no interaction between the main application and the foreign editor: the editing session is *unguided*. In some applications instantiations of external editors can be *embedded*. Some examples are KDE's filemanager `konqueror` and email reader `kmail` which can embed instances of a specially crafted version of the

Vim text editor. In these cases, the host application (`kmail`) encapsulates the editor (`kvim`) and shows its window as if the editor were part of the application. This gives the user the feeling that his favorite editor is integrated in the application, even when this integration is only visual and there is no real interaction between host application and editor.

Our focus is not so much on the *visual* integration achieved by embedding the editor instances. Instead we emphasize *functional* interaction *during* the editing session.

Another way to look at application-editor interaction is to look at the editor as the main application, and to view external tools as subordinates of the editor. Especially users of the Emacs family of editors find ways to link their email reader, spell-checker, or other popular application into Emacs by writing support *glue* in Emacs LISP.

5.2 Design

In any IDE it is common to have multiple simultaneous editing sessions, as users start and finish editing, switching from one file to another. To take care of any administrative issues we have to deal with the following tasks:

Managing Using multiple editing sessions requires administration of open sessions and addressing these editing sessions.

Executing Supporting several editors almost certainly results in different startup procedures for each editor. We provide an open and generic architecture for supporting several editors.

Marshalling We need full interaction with the supported editors, which means that data has to be transferred from the application to the editor instance and vice versa.

We first have a look at the requirements (Section 5.2.1) and then split the design into editor-independent (Section 5.2.2) and editor-specific (Section 5.2.3) details, and we show how the components connect (Section 5.2.4) to form our multiplexing editor architecture.

5.2.1 Requirements and considerations

Given our experience with editing issues in the Meta-Environment (Section 5.1.1) we are interested in a solution which is:

Noninvasive We are strongly determined *not* to edit the source code of any particular editor itself.

Simple Keep the number of methods in the editor interface low: 10 rather than 100 methods. Prefer implementation of these methods in established programming languages (e.g. C or Java), rather than the editor's (sometimes arcane) domain specific scripting language.

Open Both in terms of *platform* and *language*:

- Platform independence: although designed for a Unix environment, the implementation should be independent of whether this is e.g. Linux, SunOS, or Windows/Cygwin;
- Language independence: the architecture does not dictate any particular programming language for the editor connectors.

From the Meta-Environment point of view, we are at least interested in the following interesting editor actions and events:

Menu We want to add menu items in the editor which, when selected by the user, are forwarded to the environment where they are handled.

Cursor Cursor positioning and text highlighting can be directed by the environment (model) and rendered in the editor (view).

Modification The editor notifies the environment of any changes the user makes to the file.

Save/Load The environment can request the editor to save its contents or re-read them from the file system.

We start out with this restricted set, but we keep the design open to allow for later extensions. The less demands, the more editors we can potentially support. If for example an editor offers no support to add user-defined menus, we cannot set them up from another application either. Although we *could* patch the editor sources to add menu support we deliberately refrain from doing so.

5.2.2   Editor-independent design

The editor-independent design describes a generic way of managing and communicating with editor instances. Without knowledge of the actual editor instance, one can provide an abstract level of communication by defining a common interface which provides all necessary functionality to fulfill the requirements given in Section 5.2.1. A tool that implements this design takes care of managing editing sessions, including starting and shutting down sessions, and communication with these editing sessions. The MULTIPLEXER described in Section 5.3.1 is a tool that implements this.

5.2.3   Editor-specific design

Managing editing sessions can be done in a generic way, but actual communication and execution of editor instances has to be editor specific. This communication can be done in various ways. While Vim makes use of an arcane syntax-based communication protocol via the commandline, OpenOffice for example can be controlled by using an extensive API. These differences lead to different design implementations for different editors. To prevent changes to the MULTIPLEXER for every editor that has to
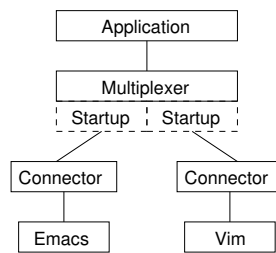
Figure 5.2: Overview showing how editors are connected to an application.

be supported we introduce a connector (see Section 5.3.2) mechanism which separates communication with the actual editor instances from managing the editing sessions. For each supported editor there has to be a corresponding connector. All editor-specific communication details are known to this connector, while the MULTIPLEXER can be implemented in a generic way. The generic interface provided by the MULTIPLEXER has to be implemented by every connector.

5.2.4 Execution models

No two editor implementations are the same, and they are often written based on different designs. Editors based on the GNU Emacs philosophy prefer to interact with external processes only if they are executed by the editor. Other editors are more easily controlled by an external process.

We accomodate for this difference by allowing two execution models. Either the MULTIPLEXER first launches the connector which launches the editor, or the MULTI-PLEXER launches the editor instructing it to immediately launch the connector.

Independent of the execution model, the final state is the same: the MULTIPLEXER communicates with the editor via a dedicated connector (Figure 5.2).

5.3 Implementation

Given the design from Section 5.2, we describe the MULTIPLEXER which contains the editor-independent implementation in Section 5.3.1. This MULTIPLEXER invokes interface methods which in turn are implemented in editor-specific connectors which are detailed in Section 5.3.2. Finally, we explain how we *glue* it all together in Section 5.3.3.

5.3.1 Editor Multiplexer

The editor MULTIPLEXER manages multiple simultaneous edit sessions by assigning each session a unique ID. Subsequent calls to the edit session carry this ID as one of the call's parameters. This allows the MULTIPLEXER to uniquely identify to which connected editor the request needs to be forwarded.

The MULTIPLEXER is currently implemented as a ToolBus tool, written in the C programming language. The ToolBus coordination architecture is a middleware layer with a process algebra based scripting language. [15] offers a comprehensive explanation of the ToolBus scripting language. Because the entire Meta-Environment architecture uses the ToolBus coordination architecture, making the MULTIPLEXER a ToolBus tool is the obvious choice. For applications that do not use the ToolBus, an implementation in the form of a C library would be equally feasible.

The choice for C as the implementation language was pragmatic. C offers direct access to operating system functionality such as process duplication through the use of the fork system call, execution of external processes using exec and has additional low level support for sockets, pipes and file descriptors. Although we also experimented with an implementation in Java during research in the context of connecting the Eclipse IDE editor [41], we opted for C's easy link to operating system functionality.

We show a simplified ToolBus interface definition of our MULTIPLEXER.

```
01  tool multiplexer is { command = "./editor-multiplexer" }
02
03  process EditorMultiplexer is
04  let
05    EM: multiplexer,
06    Editor, Filename: str,
07    SessionID, SL, SC, EL, EC: int,
08    MainMenu, SubMenu: str
09  in
10    execute(multiplexer, EM?)
11    .
12    (
13      rec-msg(edit-text(Editor?, Filename?))
14      . snd-eval(EM, Editor, Filename))
15      . rec-value(EM, SessionID?)
16      . snd-msg(edit-text(Editor, Filename, SessionID))
17    +
18      rec-msg(set-focus(SessionID?, SL?, SC?, EL?, EC?))
19      . snd-do(EM, set-focus(SessionID, SL, SC, EL, EC))
20    +
21      rec-event(EM, menu-selected(SessionID?, MainMenu?, SubMenu?))
22      . snd-msg(menu-selected(SessionID, MainMenu, SubMenu))
23    )
24    * delta
25  endlet
```

This example is limited to showing the execution (line 10) of the previously declared multiplexer tool (line 01). Following the execution is a looping construct (lines 12-24). During each iteration exactly one of the declared scenarios can occur. First, a request to start a new session is handled (lines 13-16). Second a request to set the focus to a particular region delimited by start-line, start-column, end-line and end-column (lines 18-19) to any existing editor can be handled. Finally, a menu event can come in from one of the connected editors (lines 21-22).

Applications that do not use the ToolBus, could use e.g. pipes, sockets or library calls to communicate with the MULTIPLEXER.

5.3.2 Editor Connectors

For each supported editor, we implement a small connector that translates the editor-independent interface calls into the editor specific implementation. These connectors

are necessary because each editor has its own unique scripting facilities or program-ming language (Vim uses Vim script, GNU Emacs uses Emacs Lisp), and because communication with each editor is usually handled in a slightly different way. We describe the connectors we implemented for Vim, GNU Emacs, and for a proprietary implementation of an editor in JFC/Swing.

Vim

The Vim connector is implemented partially in `C` and partially in Vim's scripting lan-guage. The `C` functions implement the text editor interface. Commands *from* the MUL-TIPLEXER *to* the editor are sent using Vim's remote scripting feature.

For example, the implementation of the `setCursor(int offset)` method looks like this:

```
01  static void gotoCursorAtOffset(int offset) {
02    char cmd[BUFSIZ];
03    sprintf(cmd, ":goto %d", offset);
04    sendToVim(cmd);
05  }
```

Events *from* the editor *to* the MULTIPLEXER, are initiated by Vim. E.g. Vim is instructed to forward buffer changes resulting from user editing by means of the Vim hook called `BufWritePost`:

```
01  func! EnableModificationDetection()
02    autocmd BufWritePost * :call BufModified()
03  endfunc
```

where `BufModified` is a function (in Vim script) that forwards this event to the MULTIPLEXER.

Currently, the editor-specific glue for Vim is expressed in 501 lines of `C` code, and 77 lines of Vim script.

GNU Emacs

Similar to the `sendToVim` function, `sendToEmacs` is used to communicate from the MULTIPLEXER to GNU Emacs. The difference is that where Vim lacks a regular communication channel and we had to resort to using its remote scripting feature, with GNU Emacs we can communicate using a a pipe.

```
01  static void sendToEmacs(int write_to_editor_fd, const char *cmd) {
02    write(write_to_editor_fd, cmd, strlen(cmd));
03    write(write_to_editor_fd, "\n", 1);
04  }
```

The communication channel may be simpler in this version, but not all comes easy when dealing with GNU Emacs. The initial scripting necessary to setup the connector is programmed in Emacs LISP:

```
01  (defun init (args)
02    (setq emacs-connector
03      (let ((process-connection-type nil))
04          (apply 'start-process "emacs-connector" "*Meta*" "emacs-connector"
```

```
05            (split-string args))))
06    (set-process-filter emacs-connector 'multiplexer-input)
07    (process-kill-without-query emacs-connector)
08    (define-key global-map [mouse-1] 'mouse-clicked)
09    (add-hook 'after-change-functions 'buffer-modified () t)
10  )
```

Lines `02-08` execute the connector and register the LISP function `multiplexer- input` as input handler for the connector. Line `10` registers a mouse-click listener, and line `11` registers the `buffer-modified` function so it gets invoked whenever user editing causes the buffer to change.

Currently, the editor-specific glue for GNU Emacs is expressed in 436 lines of C code, and 108 lines of Emacs LISP.

JFC/Swing Editor

As an experiment and possible extension to the ASF+SDF Meta-Environment, we also created an editor based on the GUI classes available in JFC/Swing. Again similar to the previous implementations, we were able to connect this Java editor to the MULTI-PLEXER. We do not show implementation details, but it is worthwhile to mention that the connection to this editor is based on sockets, rather than pipes (as we used for the GNU Emacs connector). Although we could have used the commonly accepted route where the standard input and output streams are sacrificed and used for communication via a pipe, we opted for the socket approach, just to add this route to our repertoire.

Currently, the editor-specific glue for our JFC/Swing editor is expressed in 411 lines of C code, and a 5 line shell script to invoke java with the correct classpath for the editor.

5.3.3 Glueing it all together

Now that we have the editor-independent MULTIPLEXER, and the editor specific connectors, we can finally glue them together to get a working system. We describe how the MULTIPLEXER executes and communicates with an editor.

Executing an editor

The MULTIPLEXER executes the requested editor as follows. For each editor, we write a small piece of (C) code that is loaded as a dynamic library. This mini library contains a single `startup` function with three parameters: the filename to be edited and the two file descriptors to be used for communication with the MULTIPLEXER. The startup function for the Vim editor looks like this:

```
01  void startup(const char *filename, int readFromFD, int writeToFD) {
02    char fromMultiFD[10], toMultiFD[10]; /* file descriptors as string */
03
04    sprintf(fromMultiFD, "%d", readFromFD);
05    sprintf(toMultiFD, "%d", writeToFD);
06
07    execlp("gvim-connector", "gvim-connector",
08          "--read_from_multiplexer_fd", fromMultiFD,
09          "--write_to_multiplexer_fd", toMultiFD,
10          "--filename", filename,
```

```
11          NULL);
12
13    perror("execlp:gvim/startup");
14    exit(errno);
15   }
```

The MULTIPLEXER invokes `startup` by using the `dlopen` and `dlsym` system calls (not shown here) for interacting with dynamic libraries. We thus *extend* the MULTIPLEXER with a single function per specific editor.

In the `startup` function, we choose one of the two execution models described in Section 5.2.4. For Vim we `execute` (lines `07-11`) the connector, thus following the *connector first* execution model.

For GNU Emacs, we have a similar `startup` function. Only it was more convenient to execute `emacs` first and have it fire up the connector instead. GNU Emacs is then told to load the editor-specific startup script (in this case written in Emacs LISP) and to begin by executing the function `init`:

```
01  void startup(const char *filename, int readFromFD, int writeToFD) {
02    char evalargs[BUFSIZ];
03    sprintf(evalargs,
04          "(init \"--read_from_fd %d --write_to_fd %d --filename %s\")",
05          readFromFD, writeToFD, filename);
06
07    execlp(EDITOR, EDITOR, filename, "-load", "gnu-emacs.el",
08          "-eval", evalargs, NULL);
...  /* error handling code omitted */
11    }
```

Communicating with an editor

Depending on the functionality offered by each specific editor, we use different means of setting up a communication channel with the editor. We have used different channels ranging from a `pipe` (in GNU Emacs), to a `socket` (in the JFC/Swing editor), to the more esoteric remote scripting feature offered by Vim.

Independent of the type of the available communication channel, we use the *same* technique to *marshal* data over this channel. Instead of writing ad-hoc marshalling and de-marshalling code in the MULTIPLEXER and the connectors, we use ApiGen [79]. ApiGen takes as input an abstract data type description (ADT) and generates a `C` library or `Java` jar-file containing a.o. `set`, `get` and serialization methods.

Each command to and event from the editor is formalized in the text editor ADT. From this specification ApiGen generates the API implementation which we use to (de-)marshal communication between the MULTIPLEXER and editor.

5.4   Discussion and Future work

We have implemented a framework that allows reuse of off-the-shelf editors such as GNU Emacs and Vim in the ASF+SDF Meta-Environment. By implementing as much as possible of this framework in a generic, editor-independent way (our MULTIPLEXER), we can easily and rapidly add other editors to our environment. Deploying code generation techniques (ApiGen), and an available (programmable) middleware layer (ToolBus) ensures the solution is cheap in maintenance.

103

Our editing solution is *noninvase*: we never change any editor internals, *simple*: only a handful of lines of code in the editor's own scripting language are needed, and *open*: our editing support has been tested on various Linux platforms, and we have both `C` and `Java` connectors.

Our editing framework was primarily designed for use in the Meta-Environment, which relies heavily on the ToolBus as its middleware layer. However, our contribution is not limited to using the ToolBus, and we plan to offer our results for use in a non-ToolBus setting, as a downloadable package.

Another direction of interest is figuring out in which ways we can expand the text editor interaction. We have already experimented with syntax highlighting (i.e. one tool describes which part of the text gets which font attributes and colour and the rendering is done by the text editor), and structured editing, but conceivably several more applications can benefit from our support.

Obviously, the more complicated the things we demand, the fewer editors we will be able to fully support. Vim, for example lacks atomic functionality to colour a specific region of characters, although it does offer complex syntax highlighting. This leads to the following question: *what is the set of text editing primitives small enough to be covered by almost any editor, but large enough to be useful in most applications that require editing?*

Finally, as its name states, one of the MULTIPLEXER's task is *multiplexing* simultaneous editing sessions. In a coordination architecture such as the ToolBus, the multiplexing *concern* could be applicable to other tools as well. If this notion were lifted to a ToolBus primitive, any setting that launches multiple instances of a tool with the same interface could possibly benefit.

Chapter 6

Software System Extensibility

6.1 Introduction

In Section 1.1.1 we described six criteria by which the quality of a software architecture can be measured [9]: availability, modifiability, performance, security, testability, and usability. In this Chapter we focus on the *modifiability* of software systems.

From a software engineering perspective, answers to the following basic questions can be useful in assessing the impact of the change to the system:

Who Who will make the change? Depending on whether the source is the developer, a system administrator, or an end user, different forms of modifiability are needed.

What What needs to be changed? The changes to be made can be functional changes (e.g., adding/removing a function, or changing an existing one). They can also be changes to the qualities of a system, e.g., improving responsiveness or availability.

Where Where does the system need to be changed? Which part of the system has to be adapted depends on the type of change. Changes can affect any part of the system. They could, e.g., be local to the user interface (changing the menu font), they could be part of the system's persistence engine (switching from one database system to another), or to the middleware used (switch from UNIX pipes to CORBA).

When When can the change be made? Changes can be made at different moments in the development and deployment of a system, e.g., during design time, compile time, build time, initiation time, and runtime.

The obvious next question: *Why should we make this change?* is less interesting from a modifiability point of view. Although debating the motivation for a particular change may be (very) useful in discussions about a software system, we will assume a change request *as is*.

In [9] these who/what/where/when questions, along with a response to the change request (whoever makes the change must understand how to make, test, and deploy it), and a response measure (all possible responses take time and cost money) define a *modifiability scenario*. These scenarios can then used to evaluate the modifiability quality attribute of a software architecture. An example of such a scenario is "A developer wishes to change the user interface. This change will be made to the code at design time, it will take less than three hours to make and test the change, and no side-effect changes will occur in the behaviour."

In this Chapter, we are primarily interested in modifications and extensions to a software system which can be made by *end users*. We are interested in any kind of changes they can make, without any a priori assumptions about *what* the change is, or *where* it should be implemented, although we do realize that end users are unlikely to change, e.g., the middleware used in an application. As for *when* the changes can be implemented, we would like this to be as late as possible, to get as much flexibility as possible.

6.1.1   Research Context

The ASF+SDF Meta-Environment [24] consists of several components, one of which is the user interface. Unfortunately, in a way, the GUI has become a monolith of its own. Taking care of the user interaction for *all* the other components, it has become a central component with detailed knowledge about many parts of the system. This makes it hard to add new components that require user interaction to the system. When a new component is added, or when new functionality is added to the system using existing components, the user interface has to be adapted to account for the new user interaction.

In more general terms, we are concerned about the phenomenon that in a multi component application, a single component ends up knowing about several (if not all) otherwise independent subcomponents. How can we keep the system open and extensible? This phenomenon is not unique to the Meta-Environment, various popular software systems also allow their product to be modified or extended by their users. Some examples are:

Firefox, Thunderbird The Firefox web browser [60] and Thunderbird e-mail client [116], can be extended to allow new functionality and their user interface can be changed to allow for different ordering of menus or buttons on the tool bar. The appearance of the user interface can be changed through the installation of themes.

Eclipse The Eclipse IDE [62] can be extended through a plug-in mechanism. Moreover, these plug-ins can themselves be extended by other plug-ins.

Winamp, Windows Media Player Media players such as Winamp [114] and Microsoft Windows Media Player [108] can be extended to allow new media types to be played which are not supported by default. Users can also add new animations which can be shown when the player is playing an audio only file.

106

The extension mechanisms offered by these software systems are widely used. Firefox, for example, has well over six hundred extensions written by members of the Firefox user community. While the core of the software system is developed and maintained by the system's own team of programmers, many extensions are developed by independent (teams of) programmers.

One way to adapt or extend a software system, is to do so by actually changing the existing source code. For example, aspect oriented software development [84] focuses on isolating specific concerns of a software system, developing source code specific for each of these aspect independent of the rest of the system, and finally *weaving* these aspects into an existing body of source code. In a similar way, invasive software composition [5] regards software components as being distinct components initially, which can be merged in a specific implementation. Both aspect oriented software development and invasive software composition are centered around a standard programming language (Java), and operate by changing (merging) a body of source code.

In the approach described in this Chapter, we target mainstream programming and scripting languages, and try to achieve software system extensibility without changing the source code of individual components in an invasive way.

6.1.2 Research Questions

From a technical point of view it is interesting to study the systems mentioned above, and to compare their ingredients. How can we categorize plug-in systems? Which implementation techniques are better suited for which category of plug-in system? In particular, we are interested in answers to the following questions:

- How do current software systems achieve various levels of extensibility?

- How do they deal with consistency and interaction between the core of the system and the plug-ins, and between different plug-ins?

- How do these plug-in mechanisms relate to a system using a component coordination architecture such as the ToolBus?

6.1.3 Overview

We study several extension mechanisms in popular software systems, and characterize each of them. In Section 6.2 we look at three different extension mechanisms offered by the Mozilla suite. Section 6.3 elaborates on the plug-in system implemented in the Eclipse platform. In Section 6.4 we study the Java Plug-in Framework (JPF). Finally, in Section 6.5 we have a look at the Winamp Media Player. Each section explains how the extension mechanism works, and states some specific characteristics of that particular system. In Section 6.6 we compare these characteristics and define some basic "Software Extension System Categories". In Section 6.7 we show how the use of a plug-in framework we implemented made the Meta-Environment more open and extensible, solving the monolithic GUI problems it suffered from. Section 6.9 summarizes this chapter and discusses some conclusions.

6.2   The Mozilla Software Suite

The Mozilla all-in-one Internet Application Suite [101] and its derivatives, the Firefox web browser [60] and Thunderbird e-mail client [116], offer three ways to change the way the components look and behave, each with its own specific goal:

themes   A *theme* changes the look and feel of the user interface;

plug-ins   A *plug-in* is the way to add support to Mozilla for proprietary document formats, such as the Acrobat Reader for PDF documents by Adobe Systems;

extensions   An *extension* is a small add-on that adds new functionality to Mozilla, or that changes existing functionality. A popular example is Adblock which allows advertisements to be blocked from view.

The Firefox download area hosts 114 themes, 7 plug-ins, and 673 extensions[1]. The fact that there are almost one hundred times as many extensions as there are plug-ins can be explained by their difference in complexity and the purpose they serve.

6.2.1   Mozilla Themes

Themes are skins for the Mozilla components. They allow users to change the look and feel of the user interface. Themes do not add new functionality to Mozilla, and are therefore not very interesting from an extensibility point of view. In fact, a Mozilla theme is not really an extension, but merely a specific configuration of an otherwise unchanged component.

6.2.2   Mozilla Plug-ins

The Mozilla system can be augmented by plug-ins which allow third parties to embed new functionality into the core of the Firefox browser. This allows proprietary content such as a specific document format to be rendered inside the browser. Examples include the Acrobat Reader from Adobe Systems for displaying PDF documents inside the Firefox browser, and the Java plug-in by SUN Microsystems which enables Firefox to run applications that use Java technology.

Mozilla plug-ins are heavyweight components, requiring a solid understanding of the internals of Mozilla. The existing plug-ins are written in C++. They are connected using a cross platform framework similar to COM, called XPCOM, with a corresponding interface definition language, called XPIDL. Without elaborating on the implementation details of plug-ins, it is clear that they are meant to add a significant chunk of specific functionality to Mozilla: interaction with a specific type of web content.

Mozilla plug-ins are *complete* and *closed*, meaning a plug-in does not rely on the existence of other plug-ins, nor is it intended to be extended itself by other plug-ins. It plugs into its own niche in the system, being used only to interact with web content it is meant for. Each plug-in registers which particular content type it supports, and Mozilla invokes the plug-in whenever it encounters content of that type. Plug-ins do not interfere or interact with other plug-ins.

---

[1]Data taken in September 2005.

	Mozilla Theme	*Mozilla Plug-in*	*Mozilla Extension*
Task	Fixed	Fixed	Arbitrary
Implementation	Graphics, CSS	C++/ XPCOM	Javascript
Initiative	Framework	Framework	Both
Interaction	None	Independent	Ad hoc
Extensibility	None	None	By overriding

Table 6.1: Characteristics of the *Theme*, *Plug-in* , and *Extension* mechanisms of Mozilla.

6.2.3 Mozilla Extensions

The Mozilla system can be augmented by *extensions*, which are small add-ons that add new functionality to, or change existing functionality of the application. Extensions differ widely in application domain as well as in which part of the browser the extension modifies. Examples include Bookmark Synchronization to a central server, automatically entering fields in web forms, blocking advertisements on a web page, and enhancing the interaction with the Google Search Engine.

Obviously, Mozilla extensions are much more lightweight components than the plug-in relatives. Where writing a plug-in requires a significant programming background, extensions can be written by anyone capable of toying around with Javascript.

6.2.4 Summary

Table 6.1 summarizes the characteristics of the *theme*, *plug-in* , and *extension* mechanisms of Mozilla.

Task   The themes and plug-ins are used to implement fixed tasks. Themes change the look and feel of the user interface, plug-ins allow proprietary web content to be displayed in the browser. Mozilla extensions, however have no fixed task. They can be used to add arbitrary new functionality, or change existing functionality.

Implementation   Themes contain images and cascading style sheets (CSS) and some metadata package information. Plug-ins are implemented in C++ and the cross platform XPCOM suite. Mozilla extensions are implemented in Javascript.

Initiative   Mozilla takes the initiative to activate a certain theme. Themes do not trigger any activity in Mozilla themselves. Similarly, the plug-ins are instantiated by the framework when Mozilla delegates rendering of web content specific to the plug-in. Plug-ins lie dormant in the framework, until activated. Mozilla extensions *can* take the initiative and invoke parts of the framework.

Interaction   Only one theme is active in Mozilla at any given time. Therefore, there is no interaction between themes. Although there can be multiple plug-ins active in

the system, each plug-in has a dedicated task, and for each instance of those tasks, only one plug-in can be used. Plug-ins therefore neither interact nor interfere with each other. Mozilla extensions are much less restricted. They can interact with the framework itself, but by the very nature of their implementation (Javascript), can also override functionality in other extensions. The framework does not enforce one extension manipulating the functionality of another.

Extensibility Themes cannot be extended. Likewise, the plug-ins currently available are also closed components. Although they could be extended via the cross platform COM architecture, this requires significant knowledge of the workings of the third party plug-in. Mozilla extensions, however, are open to modification by other extensions. Their implementation in a script language makes it easy to see how other extensions are implemented and promote ad hoc patching against a specific version of another extension.

6.3 The Eclipse platform

The Eclipse IDE [62] provides an extensible development platform and application frameworks for building software. In [22], the Eclipse platform is described as being an extensible platform for building IDEs. It provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders can extend the core by wrapping their tool in a pluggable component, called an Eclipse plug-in. New plug-ins can add new processing elements to existing plug-ins. To this end, plug-ins support the notion of *extension point*. Each plug-in can define one or more extension points, and is itself responsible for dealing with other plug-ins that extend these extension points. At the same time, a plug-in can declare itself to extend one or more extension points in another plug-in.

Via the update manager functionality in the Eclipse IDE, a wide variety of plug-ins can be installed. In fact, there are so many plug-ins and plug-ins can be so small, that a number of plug-ins that depend on each other are grouped together into what is called a *feature*. Users select which features they want to install, and all necessary plug-ins that make up this feature are then downloaded and installed. There are about 75 features available for installation from `eclipse.org`.

6.3.1 Extension participants and roles

Because plug-ins are pivotal in the extension mechanism of Eclipse, we take a close look at its participants and the roles they play.

Extension point At the heart of the extension mechanism is the notion of *extension point*. An extension point models the concept of "smallest undividable unit of extension". For each plug-in the exact definition of what constitutes an extension point will be different. The developer decides which level of granularity in "pluggability" the plug-in will offer. For example, in a plug-in dealing with GUI menus, one definition of a "menu item extension point" would be to allow the insertion of individual

110

menu items. Another definition could be to only allow the insertion of entire menus. Depending on the application that the developer has in mind, either definition can be considered equally well.

Extension   Once defined, these extension points can be extended by what is called an *extension*. A single extension points can be extended multiple times. For example, multiple plug-ins can each add their own menu via the menu extension point we just described. In fact, even a single plug-in can extend the same extension point multiple times. In our example this can happen if a plug-in wants to add more than one menu.

Plug-in   A plug-in is then a bundle consisting of a plug-in class, a number of "conventional" `Java` classes, and a manifest file. The plug-in class acts as intermediary between the Eclipse platform and the core functionality in the plug-in. The conventional `Java` classes implement this core, abstracting from the fact that they are used in a plug-in context. The manifest file describes information such as which extension points are offered by this particular plug-in, which extension points from other plug-ins are extended by this plug-in, and which classes in this plug-in are responsible for which particular extension points.

The plug-ins and conventional classes now play one of three roles in the extension mechanism: *host plug-in* , *extender plug-in* , and *extension callback*.

Host Plug-in   The *host* plug-in of the extension *provides* extension point(s), and *is extended*. Apart from any functionality the plug-in may already be able to perform without any extension, the host plug-in also acts as the coordinator and controller of its extensions.

Extender Plug-in   The *extender* plug-in defines and implements the *extension*. Usually it makes its functionality available to the host plug-in by registering an extension callback in the host plug-in for each extension.

Extension Callback   The extension callback is a conventional `Java` object (i.e., it is not a plug-in itself) that is registered in the host plug-in by the extender plug-in. It is called by the host plug-in when an event specified in the corresponding extension point contract occurs.

6.3.2   Example

An example of how *host* and *extender* plug-in work together is given in Figure 6.1 (taken from [22]). It shows how the Eclipse workbench UI menus are extended by the Eclipse help system.

The *host* plug-in in this case is the Eclipse Workbench UI, which is identified as `org.eclipse.ui` in the example. The host plug-in offers three extension points called `editors`, `views`, and `actionSets`, respectively. The `actionSets` extension point is used in this example to add menus to the Workbench menu.
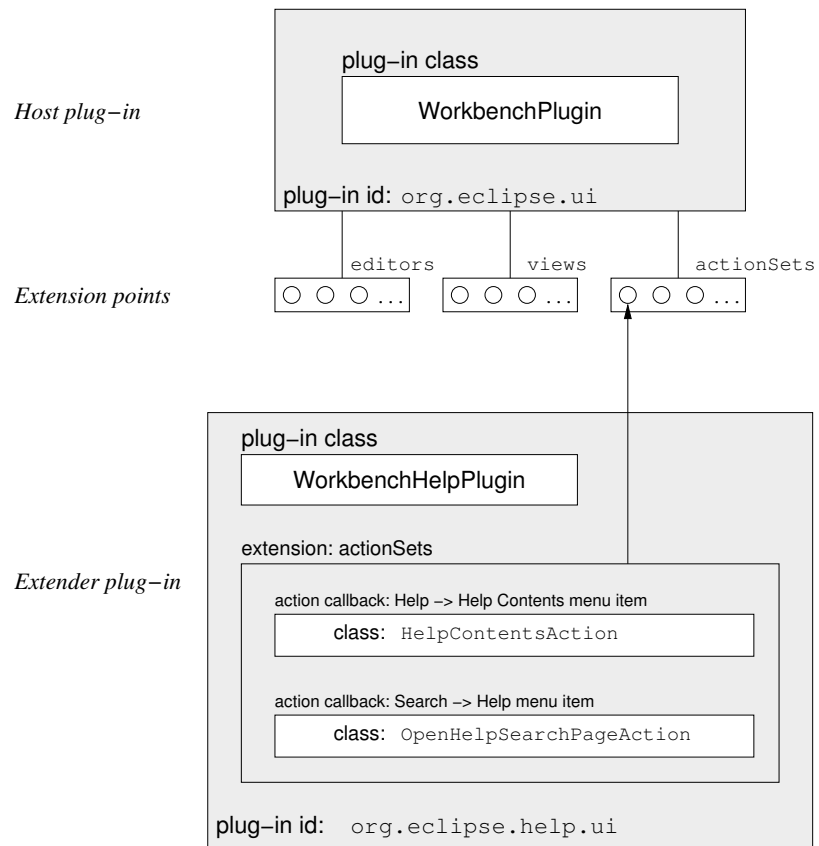
Figure 6.1: The Eclipse plug-in extension mechanism in action.

	Eclipse Plug-in
Task	Arbitrary
Implementation	Java
Initiative	Framework Controlled
Interaction	Plug-in controlled
Extensibility	Plug-in controlled

Table 6.2: Characteristics of the Eclipse Plug-in extension architecture.

The *extender* plug-in in this case is the Eclipse Help System's UI, identified as `org.eclipse.help.ui`. It contains two extensions of the `actionSets` extension point. Both install a callback in the Workbench, one for the "Help → Help Contents" menu item, and one for the "Search → Help" menu item. These callbacks are subsequently invoked when the corresponding menu item is selected by the user.

6.3.3   Summary

The Eclipse plug-in model offers deployment time pluggable components. A plug-in consists of a collection of Java classes implementing the plug-in functionality, and a manifest file with details about the plug-in. This manifest file includes the extension points the plug-in offers, and the extension points from other plug-ins, that this plug-in uses, if any. The manifest file is interpreted at runtime by the Eclipse system to instantiate the plug-in and relate it to other plug-ins.

Table 6.2 summarizes the characteristics of the Eclipse Plug-in extension architecture.

Task   Eclipse plug-ins can be used for arbitrary tasks.

Implementation   Eclipse plug-ins are implemented in `Java`, with manifest files formalized in an XML document.

Initiative   Each plug-in is fully responsible for delegating functionality of its extension points to its extenders. The framework itself is no exception to this rule, defining several extension points and delegating authority to the extending plug-ins. Each plug-in thus controls when a sub-plug-in comes into action.

Interaction   Interaction is coordinated via the plug-in hierarchy, resulting in a chain of command. The framework delegates to its immediate plug-ins, which can delegate to sub-plug-ins. At each level, the plug-in at that level coordinates whether sub-plug-ins of an extension point can work together, or whether they are independent.

Extensibility   Each plug-in can define its own extension points, thus delegating part of its functionality to other plug-ins.

6.4 The Java Plug-in Framework

The Java Plug-in Framework (JPF) [106] intends to provide a standard plug-in infrastructure for existing or new Java projects. One way to look at the JPF plug-in architecture is to view it as Eclipse, but dressed down in functionality to only the plug-in framework.

JPF thus elevates the plug-in architecture from the extension mechanism in Eclipse to a loose plug-in framework intending to achieve modular and extensible Java applications in general.

JPF uses the same terminology as Eclipse for extension points. JPF also defines a plug-in as a collection of Java classes bundled with a manifest file. Although conceptually the JPF framework is very close to the Eclipse framework, JPF was implemented from scratch. Therefore, JPF differs from Eclipse mostly in terms of implementation details.

The main conceptual difference with Eclipse is that JPF *only* offers the plug-in architecture itself. It has none of the programming tools that come with Eclipse, nor does it have a built-in graphical user interface.

One result from the fact that JPF is a re-implementation of the plug-in architecture in Eclipse is that their plug-in manifest files are incompatible. Although both use an XML file containing *almost* the same information about the plug-in, they don't use the same schema (or DTD) for the manifest file. As a result, Eclipse plug-ins cannot trivially be used by JPF, or the other way around.

Because JPF is intended to be only the *framework* that offers basic functionality to create your own application using a plug-in architecture, it offers *no* plug-ins on its website. The idea is that each application using JPF will eventually host plug-ins relevant to that application on its own website. The JPF project website currently describes one successful application that uses JPF.

6.4.1 Summary

Conceptually, the plug-in architecture in JPF is equal to that in Eclipse. In fact, JPF implements *only* the plug-in framework as is used in Eclipse. Consequently, the characteristics of JPF shown in Table 6.3, are very similar to those found in Eclipse.

An interesting feature offered by JPF over Eclipse is that it has built-in integrity checking of all registered plug-ins. Because in JPF plug-ins can be registered and unregistered at run-time, it is useful that the system monitors inter-plug-in integrity, issuing warnings when a plug-in that is needed by another plug-in is about to be removed from the system.

6.5 Winamp

Another group of software systems that rely on extensions to complete their functionality are the Media Players. A Media Player is a software system that allows audio and video streams to be played on a computer. Winamp [114] and Windows Media

	JPF Plug-in
Task	Arbitrary
Implementation	Java
Initiative	Framework Controlled
Interaction	Verified by Framework, Plug-in controlled
Extensibility	Plug-in controlled

Table 6.3: Characteristics of the Java Plug-in Framework (JPF) architecture.

Player [108] are two popular Media Player. Media streams come in various formats, e.g. MP3 or WAV for audio streams, and Divx or WMF for video streams.

One of the reasons to have extensions in a media player is to allow for easy "plugging" of different encoding/decoding components, called Codecs for short. Each Codec implements the encoding and decoding of a particular audio or video stream and thus extends the generic functionality of the Media Player with support for that particular type of media stream.

Another popular goal of plug-ins in Media Players is to have pluggable visualization components for audio streams. When an audio stream is played, most of the user interface is static and boring, and to liven up the looks of the application, a plug-in can show an animation based on the audio stream currently played by the Media Player.

The Winamp website offers well over a thousand different plug-ins. In Section 6.5.2 we give a rough break-down of the number of plug-ins per category.

6.5.1   Winamp Themes

Similar to Mozilla Themes (Section 6.2.1), Winamp also offers users a way to alter the way the application looks and feels. In fact, Winamp adds the possibility to have animations in the interface. But again, a Winamp Theme does not add new functionality that is unavailable.

6.5.2   Winamp Plug-ins

According to the documentation, Winamp plug-ins are implemented as 32-bit Windows Dynamically Linked Libraries (DLLs), with primary support for the Microsoft Visual C++ platform. Each Winamp plug-in is always an instance of exactly one of the following plug-in types:

Input Input plug-ins give Winamp the ability to play additional file types that are not natively supported by the application. There are about 75 input plug-ins.

Output Output plug-ins allow Winamp to manifest audio data in different ways. There are about 30 output plug-ins.

DSP/effect Digital signal processing (DSP) Plug-ins manipulate audio data before actually being sent to the speakers (or whatever the Output plug-in decides to do with it). There are over a 100 effect plug-ins.

Visualization Visualization Plug-ins display some sort of visual effect based on audio as it is being decoded by Winamp. There are about 250 visualization plug-ins.

Language pack Language packs are used to internationalize Winamp to the language of your choice. There are about 50 language packs available.

Media Library Media Library plug-ins extend the media library for instance, for portable devices such as iPods, accessing Media Library databases, etc. There are about 5 such plug-ins available.

General purpose Anything that needs to run continuously in the background or does not require audio processing qualifies as a General purpose plug-in. There are well over 500 general purpose plug-ins.

With the exception of the *Language pack* which only contains internationalization strings for the textual representation of Winamp menu items, all other plug-in types allow actual functionality changes and thus extend the behavior of Winamp in exactly one of the defined types.

6.5.3   Summary

Table 6.4 summarizes the characteristics of the Winamp theme and plug-in extensions.

Task Both themes and plug-ins have specific tasks in Winamp. Each plug-in must be of one of the defined plug-in types.

Implementation Winamp themes consist of a bundle of graphics files for the buttons and menus, and XML files describing the layout of the user interface. The plug-ins are implemented as Windows 32bit Dynamically Linked Libraries (DLLs) and support is primarily offered for those plug-ins written in C++.

Initiative Themes are fully controlled by the framework. Plug-ins are invoked by the framework when and where they are needed.

Interaction Only one theme is active at any given time, so there is never any interaction between themes. As plug-ins serve independent roles, they do not interfere with one another.

Extensibility Winamp themes and plug-ins are not extensible themselves.

6.6   Extension Mechanism Comparison

In this chapter, we studied several software systems that offer various extension mechanisms. One way to look at the characteristics of these systems, is to group them into categories. The remainder of this Section describes four extension mechanism categories we observed in existing software systems: *decoration* (6.6.1), *delegation* (6.6.2), *mediation* (6.6.3), and *adaptation* (6.6.4). Similar to the way software design patterns are identified in [64], these names describe the primary use of that particular plug-in pattern.

	Winamp theme	*Winamp Plug-in*
Task	Fixed	Fixed
Implementation	Graphics + XML	Win32 DLL, C++
Initiative	Framework	Framework
Interaction	None	Independent
Extensibility	None	None

Table 6.4: Characteristics of the Winamp Media Player architecture.

6.6.1   Decoration

Definition 1 *A Decoration extension changes the application in a cosmetic way, offering no new functionality relevant to the main goal of the system.*

This category is the home of *themes*. Serving a fixed purpose, namely changing the way the user interface looks and feels, they add no *new* functionality to the system. Only a single theme is active at any given time, so themes do not interact with one another. The framework is fully responsible for switching from one theme to another, themes do not initiate functional activity of the framework. Themes are implemented mostly using graphics files and some metadata describing how graphics, buttons, and textual menus are laid out in the user interface. All added functionality, if any, is restricted to user interface animation and signaling events to the framework.

Some examples of the use of *decoration* extensions are the themes used in Mozilla and Winamp.

6.6.2   Delegation

Definition 2 *The Delegation extension pattern is used to select a specific implementation for a pre-defined task in the system.*

This category contains embedded plug-ins that are responsible for the implementation of a single task, such as decoding a specific kind of audio or video stream. Instances of different plug-ins possibly coexist in the same application, but because each plug-in implements only a specific function, the instantiated plug-ins act independently of each other. The framework invokes a well known entry point in the plug-in, supplying it the relevant parameters. The framework thus *delegates* the specific implementation of a more generic concept to the plug-in.

Examples of the use of extensions by *delegation* are the Adobe Acrobat plug-in and SUN Java plug-in in the Mozilla Suite.

6.6.3   Mediation

Definition 3 *In the Mediation extension pattern, (almost) all core activity of the application is achieved through the use of multiple cooperating extensions. The extension framework plays a crucial, mediating role in the functioning of the entire system.*

117

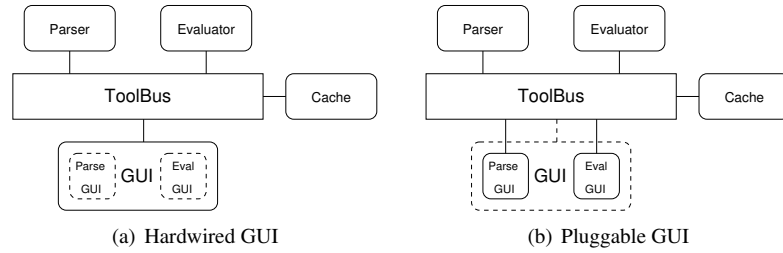(a) Hardwired GUI   (b) Pluggable GUI

Figure 6.2: Component knowledge in the GUI: hardwiring versus plugging.

The frameworks in this category offer some basic functionality but, as a primary goal, are highly extensible. In contrast to systems that use plug-ins as delegates for specific tasks, the framework acts as a *mediator* between a number of cooperating extensions.

Examples of the use of extensions through *mediation* are the Eclipse plug-in system, and the Java Plug-in Framework.

6.6.4   Adaptation

Definition 4 *In the Adaptation extension pattern, the system offers a way to adapt the system without exercising strict control over the extension.*

Some frameworks allow users to *adapt* its functionality, usually by means of (small) scripts. The goal of these adaptations is not to add new chunks of functionality to the system, but to make minor modifications given the already rich set of functionality offered by the system.

An example of extension through *adaptation* is the Firefox Adblock extension.

6.7   A Plug-in Architecture for the Meta-Environment

Having studied various plug-in architectures, how does the Meta-Environment fit into all of this? In Section 6.1 we noted that the GUI component in the Meta-Environment has evolved into a monolith with knowledge about many other components.

Figure 6.2(a) shows a largely simplified view of the Meta-Environment consisting of four components: a parser, an evaluator, a cache, and the GUI. These components are interconnected using the ToolBus [15]: a component coordination architecture. The ToolBus coordinates the connected components by means of a ToolBus script, a scripting language based on process algebra. This script dictates all possible interaction between any of the connected components.

In the example, the parser and the evaluator both require user interaction, the cache remembers computationally expensive results from the parser and evaluator, but requires no user interaction. Although the GUI is a separate component, the fact that two other components need user interaction is reflected in the GUI. In this example, the GUI

118

knows how to visualize parse trees (a result from the parser) and how to interactively deal with the evaluator. If a new tool is added which also requires user interaction, the source code of the GUI has to be adapted, to allow for the extra information needed to deal with the new component.

The GUI contains identifiable chunks that deal with user interaction for specific subcomponents of the Meta-Environment. It would be nice if smaller pieces of user interaction related tasks could be encapsulated in their own GUI element. In our example, we would like the parsing component to also be responsible for the part of the GUI that deals with parsing, and the evaluator for the part of the GUI that deals with its interaction. Figure 6.2(b) proposes a change to the architecture by removing the parser and evaluator specific parts from the GUI, and replacing the GUI by a generic plug-in capable one. The parser and evaluator specific portions of the GUI are then plugged into the GUI, but are each themselves connected to the ToolBus, subjecting them to the coordinating regime.

6.7.1   Extension mechanisms in the Meta-Environment

What are the architectural options to organize the GUI in the way we just described, given the constraint that individual GUI elements should operate independently? We consider the four extension mechanisms we identified and reflect on their use in the Meta-Environment. What are they used for currently, and could they be used to reorganize our GUI?

Decoration   The Meta-Environment already uses some instance of the *decoration* extension mechanism in that font types and sizes, as well as various color schemes are supported through the use of a `configuration manager`. A property file binds an abstract name such as `menu.font.family` used in the GUI implementation, to a concrete value (e.g., "Helvetica, bold, 12pt").

Delegation   Each tool in the Meta-Environment serves a single purpose, and for each purpose there usually is only a single implementation for each goal. Still, in a way the Meta-Environment uses the *delegation* extension mechanism for all these tools, even though each one only has a single implementation. The actual tool executed to perform the task can be replaced by another implementation (even in another programming language) as long as it abides by the same ToolBus interface of that tool. For example, we have successfully reused a different term rewriting engine [37] in the Meta-Environment by simply executing the new rewriter instead of the default one. So, although in reality we often only have a single implementation, the way these external tools are connected to the system can be seen as an instance of the delegation extension mechanism.

Mediation   The *mediation* pattern appears to be the most useful extension mechanism to use for the GUI. It allows us to have a central GUI tool with certain core functionality, and it allows the GUI parts of other components to be installed in the core GUI, without loosing control over the application.

One way to implement this in the current Meta-Environment, would be to extract GUI functionality specific to certain tools into plug-ins, and to replace the monolithic GUI by a core with plug-in support.

To this end, we experiment with the Java primitives available for dynamically loading external classes. We then connect these plug-ins to the ToolBus in the same way other `Java` tools are connected to the ToolBus. In effect each plug-in thus becomes a ToolBus tool, but since it is started from within the same `Java` virtual machine, it has access to the GUI functionality offered by the GUI core. The framework needs to be able to locate and instantiate a plug-in, and connect it to the ToolBus. It then offers sufficient functionality to allow plug-ins to add a window in the core GUI.

Adaptation   The ToolBus scripting language is used extensively in the Meta-Environment to connect all independent components. In fact, all functionality offered by the application as a whole is expressed at the ToolBus script level in terms of cooperation between otherwise functionally independent tools. Extending the Meta-Environment can be done by editing the ToolBus scripts. This has already been done to extend the Meta-Environment into an Elan [37] environment, and to incorporate Action Semantics [100] to create an Action Semantics Environment [29]. So the interaction between existing and new tools can be extended by *adaptation* of the ToolBus scripts, but this mechanism is not suited to change the functionality *inside* the GUI.

6.7.2   The Basic GUI Framework

We will explain the GUI framework in two stages. First we show how the `Java` contract between GUI and plug-in works, and then we show how the GUI and plug-ins are connected to the ToolBus.

We start out with a GUI that is basically nothing more than an *empty* shell. All it does is display an initial frame with a menu bar containing a single menu item: `File->Exit`. The only possible user interaction is activating this menu item, or clicking the close button on the frame.

The contract between GUI and plug-in is deliberately kept very small. Figure 6.3 shows the `Gui` and `Plug-in` interfaces as well as two simple implementations. A plug-in is initialized using the `initPlugin` method, passing a reference to the GUI instance as a parameter. The plug-in then adds its user interface elements, such as windows, labels, buttons, etc., to the GUI using the `add` method from the `Gui` interface. As our GUI implementation uses JFC/Swing we opted for the use of `JComponent` as the base class of all user interface elements that can be added to the GUI by a plug-in.

Now that we have our basic GUI, and know what a plug-in looks like, how can we actually instantiate and initialize a plug-in? Obviously, the GUI does not know in advance which plug-ins exist nor where to find the compiled `Java` code that implements them. Our answer is to make the GUI a ToolBus tool. First, we give a ToolBus interface definition for a `TrivialGUI`, which we will expand to allow the use of plug-ins.

```
public interface Gui {          public interface Plugin {
  void add(JComponent c);         void initPlugin(Gui g);
}                               }
```
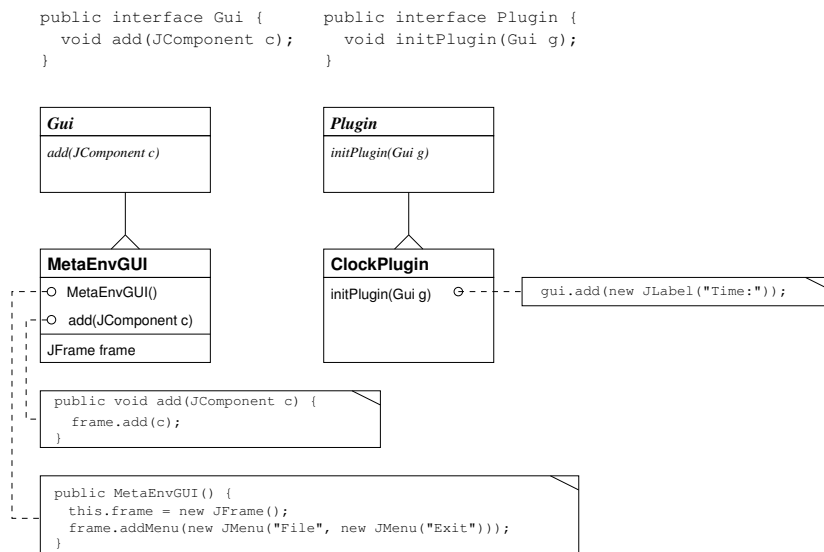

Figure 6.3: The interface between GUI and plug-in at the Java implementation level.

```
process TrivialGUI is
let
  GUI: gui
in
  execute(gui, GUI?) .
  (
   rec-event(GUI, gui-close-event)
   . snd-note(gui-close-request)
   . snd-ack-event(GUI, gui-close-event)
  ) * delta
endlet
```

This `TrivialGUI` only starts the GUI and propagates any `gui-close-events` it receives from the GUI.

Now, we need some uniform way to package all files belonging to a plug-in and define a way to deliver this package to the GUI. A collection of compiled `Java` classes are usually packaged in a Java ARchive (`jar`) file. Such a `jar` file also contains a manifest file with meta information about the plug-in, including the name of the main class of the archive. Therefore, such archives are very suitable to package plug-ins, because it contains all necessary `Java` classes and extra information that we need, and it is a mainstream way of packaging `Java` applications. We pass the location of the `jar` file by means of a Uniform Resource Locator (URL) which allows both local (on the user's machine) and remote (anywhere on the web) plug-ins to be used.

We extend `TrivialGUI` with an `add-plugin` message that allows a plug-in to be added by passing the location (URL) of the `jar` file of the plug-in. We call this extended GUI `PluggableGUI`. Extensions to the previous script are given in **boldface**.

```
process PluggableGUI is
let
  GUI: gui,
  PluginURL: str
in
  execute(gui, GUI?) .
  (
    rec-event(GUI, gui-close-event)
    . snd-msg(gui-close-request)
    . snd-ack-event(GUI, gui-close-event)
    +
    rec-msg(add-plugin(PluginURL?))
    . snd-do(GUI, add-plugin(PluginURL))
  ) * delta
endlet
```

After extending the ToolBus interface, we need to implement the `Java` counterpart of `add-plugin`. There we need to load the `jar` file and instantiate the plug-in. Fortunately, `Java` comes with ample support for locating `jar` files via an URL, and provides easy access to the `jar` file's meta information, such as the fully qualified name of the main class of the `jar` file. We show a simplified version, without any exception handling details, of how this is done in the `MetaEnvGUI`. This version focuses on how a plug-in can be added to the GUI by loading a `jar` file and instantiating its main class.

```
public class MetaEnvGUI implements Gui {
  private JFrame frame;
  public MyGui() { this.frame = new JFrame(); }

  public void add(JComponent c) { frame.add(c); }

  public void addPlugin(String pluginURL) {
    new PluginLoader(pluginURL).newInstance().initPlugin(this);
  }
}

class PluginLoader extends URLClassLoader {
  private URL url;
  public PluginLoader(String pluginURL) { url = new URL(pluginURL); }

  private String getPluginMain() {
    JarURLConnection juc = url.openConnection();
    Attributes attrs = juc.getMainAttributes();
    return attrs.getValue(Attributes.Name.MAIN_CLASS);
  }

  public Plugin instantiatePlugin() {
    String pluginMain = getPluginMain();
    Class pluginClass = loadClass(pluginMain);
    return (Plugin) pluginClass.newInstance();
  }
}
```
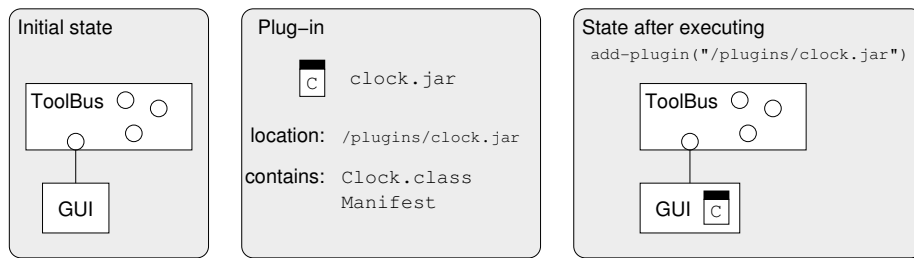
Figure 6.4: The ToolBus instructs the `Java` GUI to instantiate the clock plug-in by loading the corresponding `jar` file.

6.7.3   Example: simple clock

We now have all the ingredients for a simple example. Suppose we want to enrich our bare GUI with a clock plug-in. This clock adds a label to the GUI showing the current time. Figure 6.4 shows the loading of `clock.jar` in the GUI.

Without going into details on how to implement the actual clock functionality, our clock class could look something like this:

```
public Class Clock implements Plugin, Runnable {
  private JLabel label;

  public void initPlugin(Gui gui) {
    this.label = new JLabel();
    gui.add(label);
    new Thread(this).start();
  }

  public void run() {
    while (true) {
      label.setText("Time: " + getTime());
      Thread.sleep(100);
    }
  }
}
```

Assuming that the clock's `jar` is stored in the file `/plugins/clock.jar`, the clock plug-in can now be activated from a ToolBus script as follows:

```
snd-msg(add-plugin("/plugins/clock.jar"))
```

6.7.4   Extension: Allowing communication to a plug-in

So far, we have loaded the plug-in in the GUI, but we have not yet facilitated a way for later communication to (or from) the plug-in. As a next step, suppose that we want to allow for external synchronization of the clock, e.g., to make it display time using a different time zone offset.

In our `MetaEnvGUI` implementation we also abstracted from all ToolBus details. But as `MyGui` is a ToolBus tool, the real implementation has a connection to the Tool-Bus. We now extend the `Gui` interface by adding a `connectToToolBus` method

which allows plug-ins to ask the GUI to create a separate ToolBus connection specifically for that plug-in. At this point, we also have to define the ToolBus interface of the clock plug-in. It is no longer an independent plug-in, it now becomes part of the application because it offers functionality (setting the time zone). The plug-in will have its own connection to the ToolBus just as all other ToolBus tools have, and communication to the plug-in also uses the same mechanism.

```
process Clock is
let
  Clock: clock,
  Timezone: str
in
  rec-connect(clock, Clock?) .
  (
    rec-msg(set-timezone(Timezone?))
    . snd-do(Clock, set-timezone(Timezone))
  ) * delta
endlet
```

We extend the `Gui` interface with a `connectToToolBus` method which will allow plug-ins to request the GUI to connect them to the same ToolBus instance that the GUI is connected to:

```
public interface Gui
  public void add(JComponent c);
  public void connectToToolBus(String toolName);
```

Next, we add the ToolBus interface file, containing the `Clock` process which describes all possible interaction between the clock tool and the ToolBus to the application. And finally we change the `initPlugin` method of `Clock.java` so it actually connects to the ToolBus:

```
public void initPlugin(Gui gui)
  this.label = new JLabel();
  gui.add(label);
  gui.connectToToolBus("clock");
  new Thread(this).start();
```

Note that although we extended the GUI with a mechanism to allow plug-ins to connect to the ToolBus, none of this is specific to the clock application. No clock specific knowledge enters the core GUI. The connection architecture is now as shown in Figure 6.5.

6.7.5 Extension: Allowing communication from a plug-in

Having just added the ability to communicate to an instantiated plug-in, the next obvious step is to define communication from the plug-in back to the ToolBus level. In our clock example, we could use this to allow the clock to send an alarm signal to the ToolBus.

We extend the clock's ToolBus interface to allow for this new functionality:
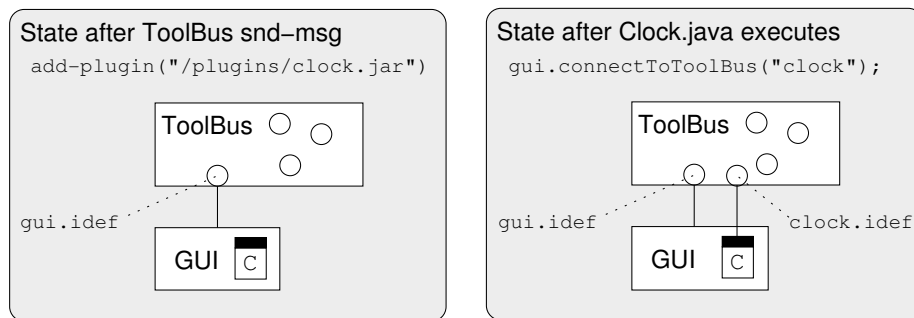
Figure 6.5: After the Clock connects to the ToolBus, both GUI and Clock have their own ToolBus connection, bound by GUI and Clock specific interfaces, respectively.

```
process Clock is
let
  Clock: clock,
  Timezone: str
in
  rec-connect(clock, Clock?) .
  (
    rec-msg(set-timezone(Timezone?))
    . snd-do(Clock, set-timezone(Timezone))
  +
    rec-event(Clock, alarm)
    . snd-note(clock-alarm)
    . snd-ack-event(Clock, alarm)
  ) * delta
endlet
```

Because the clock already has a ToolBus connection from our previous extension, nothing has to be changed except the clock's ToolBus interface and the implementation inside the clock itself. This is exactly the same as when any other ToolBus tool changes its functional interface. This keeps the distinction between plug-ins and non plug-ins minimal: the only difference is that the plug-ins have access to the core GUI.

6.7.6   Extension: Allowing inter-plug-in communication

As a final extension to our clock example, we consider what happens when we want to have two clocks, one clock with a digital display, and one with an analog display. Suppose that both clocks offer user interaction to set the current time of the clock, how can we keep both clocks synchronized? That is, if the user advances one of the clocks by an hour, how do we get the other clock to update its notion of time and have it show the same new time?

We extend the clock's ToolBus interface by adding three things. First, the clock subscribes to notes that will inform it of any time changes. Second, whenever it receives such a note, it propagates this to the tool which then updates its display. Finally, whenever the clock tool fires an event signaling a time change, the `Clock` process

sends out a note informing others of the event. The ToolBus code of the extended `Clock` process is as follows:

```
process Clock is
let
  Clock: clock,
  Time: term
in
  rec-connect(clock, Clock?)
  . subscribe(time-changed(<term>))
  .
  (
    rec-msg(set-timezone(Timezone?))
    . snd-do(Clock, set-timezone(Timezone))
  +
    rec-event(Clock, alarm)
    . snd-note(clock-alarm)
    . snd-ack-event(Clock, alarm)
  +
    rec-note(time-changed(Time?))
    . snd-do(Clock, set-time(Time))
  +
    rec-event(time-changed-event(Time?))
    . snd-note(time-changed(Time))
    . snd-ack-event(time-changed-event(Time))
  ) * delta
endlet
```

Assuming that the digital clock is in `/plugins/digital-clock.jar` and the analog one in `/plugins/analog-clock.jar`, instantiating them in the GUI is done by sending two similar ToolBus messages from anywhere in the ToolBus script. For example, we can define a process `StartClocks` which starts both clocks:

```
process StartClocks is
  snd-msg(add-plugin("/plugins/digital-clock.jar"))
  . snd-msg(add-plugin("/plugins/analog-clock.jar"))
```

In this example, where we use `subscribe` and `rec-note`, all connected clocks are kept up-to-date of any time changes by the asynchronous note mechanism offered by the ToolBus. Whenever one clock receives an event from the GUI that the time has changed, it broadcasts this to all connected clocks that use `clock.idef` causing them to update their current time as well.

6.8   Current plug-ins in the Meta-Environment

To illustrate how the plug-in system we implemented works in the ASF+SDF Meta-Environment, we now describe the plug-ins currently in use. We show some statistics of these plug-ins in Section 6.8.1. As an example, we show the ToolBus interface definition of one particular plug-in in Section 6.8.2. Finally, Section 6.8.3 elaborates on one specific instance of plug-in interaction relevant to the Meta-Environment.

Figure 6.6 shows a screenshot of the ASF+SDF Meta-Environment, with running instances of all GUI plug-ins currently available. We give a brief description and close-up of each of these plug-ins.
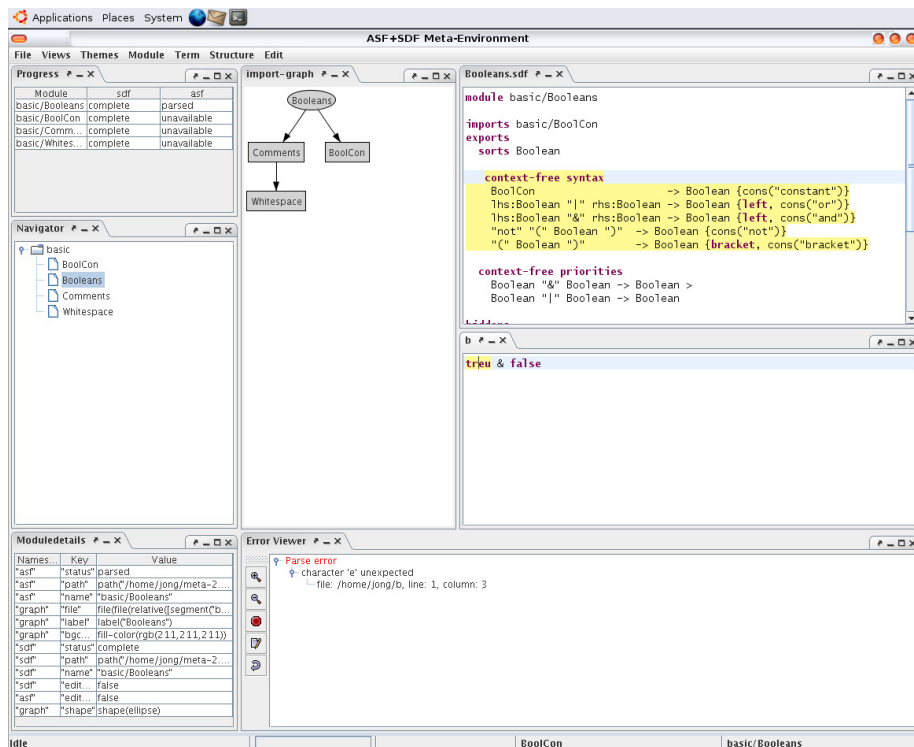
Figure 6.6: Screenshot of the plug-ins in the ASF+SDF Meta-Environment.

The GRAPH plug-in (Figure 6.7) is capable of displaying multiple graph-like structures on a canvas in the GUI. In the Meta-Environment it is used to display the import relations between active modules, and to show (parts of) parse trees selected by the user. It is an interactive component: users can click on nodes in the graph and select context sensitive menu items on those nodes.
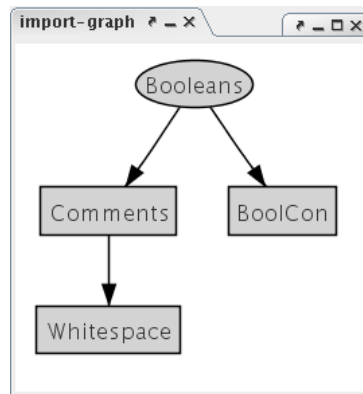

Figure 6.7: The GRAPH plug-in interacts with arbitrary graph-like structures.

The NAVIGATOR plug-in (Figure 6.8) shows all active modules and their import relations in a tree-like structure. Similar to the generic GRAPH plug-in which provides a more global overview of the import relations, the NAVIGATOR plug-in allows the user to address modules in a more structured way. It allows collapsing and expanding of the tree nodes, and can have separate context sensitive menu items for each module in the tree.
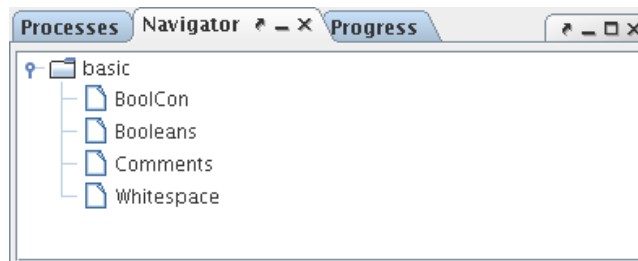

Figure 6.8: The NAVIGATOR plug-in allows interaction with modules opened in the system.

The EDITOR plug-in (Figure 6.9) is used statically for the display of text files, and dynamically for all editing sessions in the Meta-Environment. The Editor supports

externally (i.e. by other components in the Meta-Environment) guided syntax high-lighting, cursor placement, and context sensitive menus.
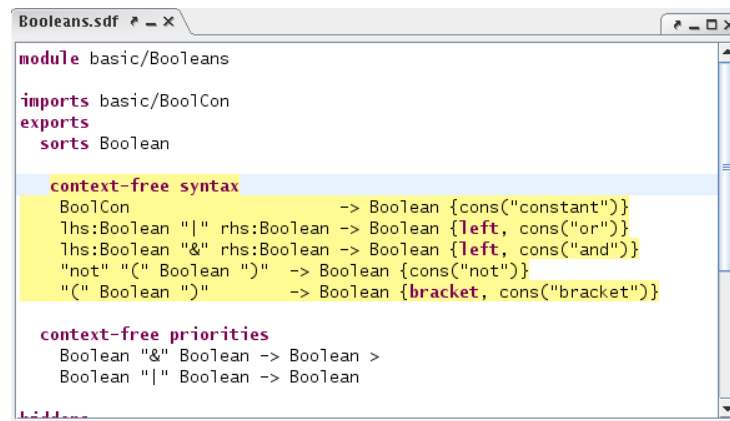

Figure 6.9: The EDITOR plug-in is used to allow user interaction with text files.

The DIALOG plug-in   contains generic support for all kinds of dialogues between the application and the user. Typical examples include a confirmation dialogue ("Are you sure you want to delete this file?"), and a file/directory chooser (Figure 6.10).

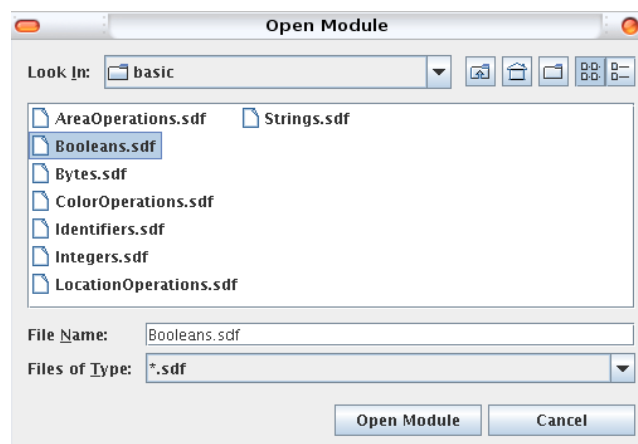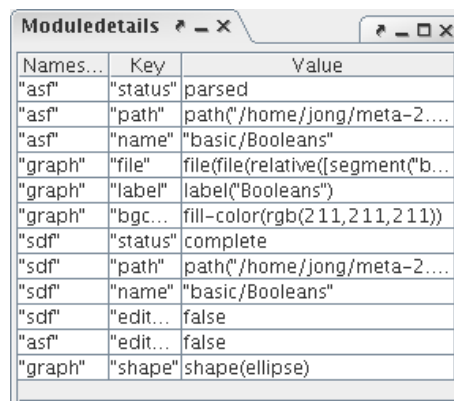Figure 6.10: The DIALOG plug-in is used for dialogues such as file selection.

The DETAILS plug-in   (Figure 6.11) is used to render module specific properties of the currently selected module in a table overview. Some examples of such properties are the exact path on the filesystem to the module, whether or not the module is editable, and the current status of the module (e.g. "parsed" or "parse error").

Figure 6.11: The DETAILS plug-in details module specific properties.

The ERROR plug-in   (Figure 6.12) deals with all kinds of errors that may occur during a session. It displays error messages as they are broadcast by various components, and allows the user to interact with these errors. Each error contains structural information about where it occurred. This means that errors can be displayed in a tree-like way, allowing the user to get more information on a particular error by expanding the node in the display. And it allows the system to "jump to" any location originally attached to the error message by the component that generated the error.
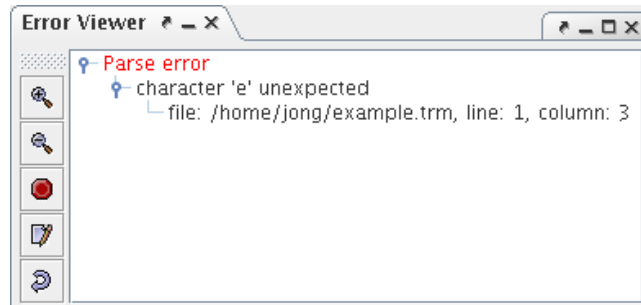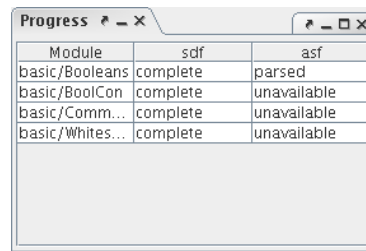
Figure 6.12:   The ERROR plug-in deals with all kinds of errors in the Meta-Environment.

The PROGRESS plug-in   (Figure 6.13) can display a progress meter for a specific (complex) task the system is currently busy working on. It can display the current state of activity (e.g. "opening file", "shutting down", or "idle"), and for tasks that have identifiable progress, it can show how much progress has been made so far (e.g. "56% of the file has been parsed").

Figure 6.13: The PROGRESS plug-in keeps track of the progress of various actions.

6.8.1 Some plug-in statistics

Table 6.5 shows some statistics on the plug-ins just described. We noted the size of the `Java` implementation of the plug-in itself, i.e., the GUI elements and the actual functionality offered by the plug-in. The next statistic is the size of the ToolBus interface definition file needed for the plug-in. This file contains the process responsible for the interaction between the plug-in and the rest of the Meta-Environment, as well as operational details, such as the location of the plug-in, and the classpath needed to run it. The final two numbers in Table 6.5 are a count of the incoming and outgoing interactions, respectively. Incoming messages request the plug-in to do something, e.g., to display some data. Outgoing messages are notifications from the plug-in to the system, such as the request for a context specific popup menu.

The source code of the Meta-Environment GUI as it was before the migration to the plug-in architecture was about 4,800 lines of `Java` code. In this version, several components which later migrated to plug-ins were already isolated in separate classes. During the migration of these components, we ran into several instances of undesirable tangling. The components were supposed to be separate, but sometimes one component would receive a reference to another component in its constructor. This link to the other component was then used in an ad-hoc manner to keep the component up-to-date with specific state changes of the other. The result was an asymmetric situation where some components were aware of (implementation) details of other components. In Section 6.8.3 we show how we were able to break these links and restore a symmetric situation where each component is again oblivious of any other component.

6.8.2 The ERROR plug-in interface

As an example, we now show the interface definition of the ERROR plug-in, responsible for the interaction with all kinds of errors in the Meta-Environment.

```
process ErrorViewer is
let
  T : error-viewer,
  Error : term,
  Location : term,
  Path: str,
  Producer : str,
  Summary : term,
```

131

Plug-in	Impl. (**Java**, LOC)	Interface (ToolBus, LOC)	# incoming messages	# outgoing messages
GRAPH	1821	69	5	3
NAVIGATOR	1122	61	3	3
EDITOR	3031	133	14	8
DIALOG	575	122	6	3
DETAILS	304	26	1	0
ERROR	945	40	3	2
PROGRESS	363	35	3	0

Table 6.5: Size of plug-in implementations and their ToolBus interfaces, as well as number of incoming (`snd-do`, `snd-eval`) and outgoing (`rec-event`) messages to the plug-in.

```
  SummaryId : str
in
  StartErrorViewer()
  . rec-connect(T?)
  .
  (
    rec-msg(ui-show-error-summary(Summary?))
    . snd-do(T, show-error-summary(Summary))
  +
    rec-msg(ui-remove-error-summary(Producer?, SummaryId?))
    . snd-do(T, remove-error-summary(Producer, SummaryId))
  +
    rec-msg(ui-remove-error-summary(Path?))
    . snd-do(T, remove-error-summary(Path))
  +
    rec-event(T, error-selected(Error?))
    . snd-msg(ui-error-selected(Error))
    . snd-ack-event(T, error-selected(Error))
  +
    rec-event(T, location-selected(Location?))
    . snd-msg(ui-location-selected(Location))
    . snd-ack-event(T, location-selected(Location))
  )
  *
  rec-disconnect(T)
endlet

% Start the ErrorViewer by loading the jar-file containing its
% implementation. The second parameter is the classpath environment in
% which the plugin is loaded. In this case, the ErrorViewer has access
% to a (generated) data-type library for Errors.
process StartErrorViewer is
  snd-msg(load-jar("file:///path/to/error-gui/error-viewer-1.2.jar",
                   "/path/to/error-support/error-api.jar"))

toolbus(ErrorViewer)
```
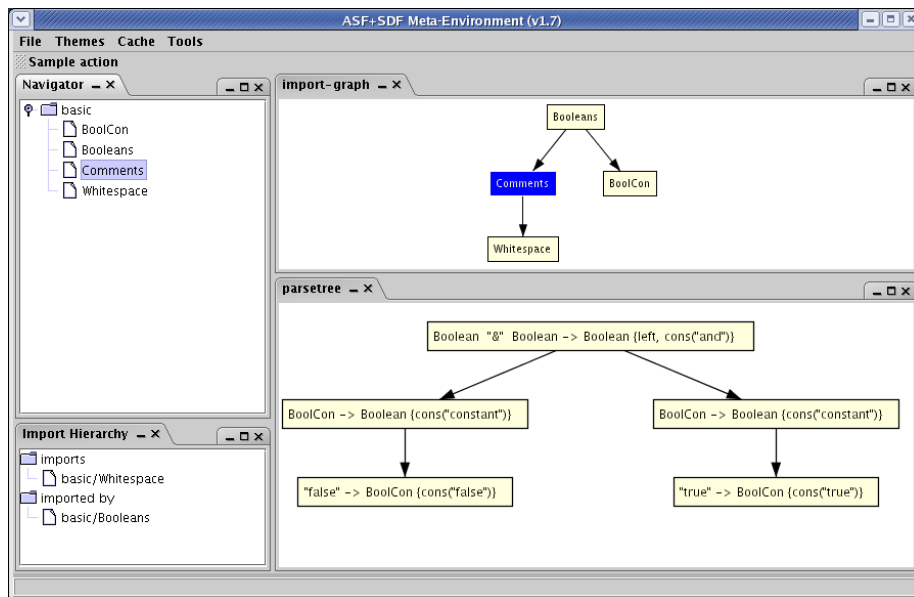
Figure 6.14:   Screenshot of the global and local NAVIGATOR views (left), and two GRAPH views (right).

Errors are displayed in a tree-like GUI element (as shown in Figure 6.12), and when the user double clicks on the location that is part of an error, this event is propagated by the ERROR plug-in to the Meta-Environment, where it can be used, e.g., to bring up the text editor (in the EDITOR plug-in) related to the error.

6.8.3   Plug-in interaction in the Meta-Environment

Figure 6.14 shows a picture of the ASF+SDF Meta-Environment system in a state where some modules have been loaded, and the GUI presents the user with various views on the import structure of these modules. In this picture, the GUI is divided into four regions. The left-hand side shows a global and a local view rendered by the NAVIGATOR, The right-hand side shows two views of graphs rendered by the GRAPH component.

In this example GUI there are both specialized views (driven by the NAVIGATOR component) and generic views (driven by the GRAPH component). The specialized views deal with the concept of a currently selected *module*, whereas the generic component deals with graphs and nodes, but is oblivious of the fact that its *nodes* are in any way related to *modules*. This is demonstrated in the `parse-tree` view where the graph nodes represent positions in a parse tree.

The conceptual difference in the notions (*specific* modules vs. *generic* nodes) used by these two GUI components makes it interesting to reason about their interaction. Focusing on the notion of *currently selected module* in our example, we describe how to

133

Conceptual layers

Example in the Meta Environment
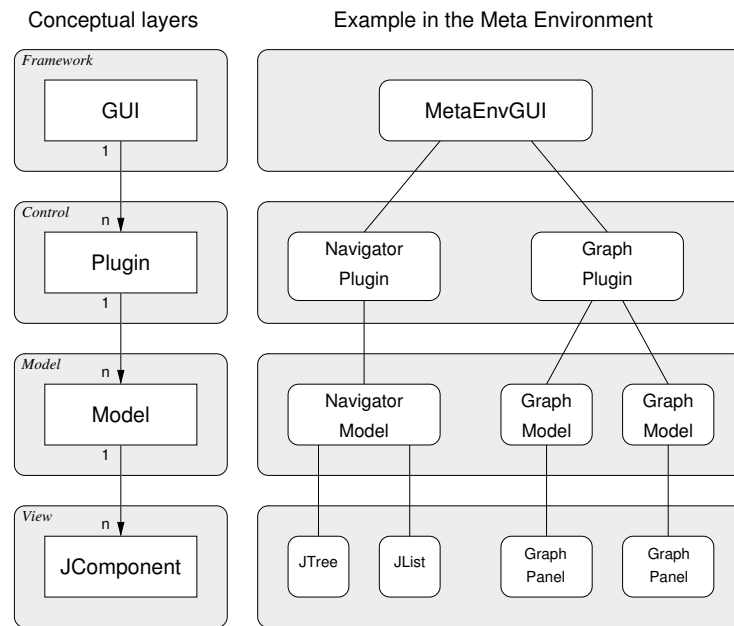
Figure 6.15: Architectural layers of the Plug-in-based GUI Framework.

keep the NAVIGATOR and GRAPH views synchronized, without either component knowing about the other.

Figure 6.15 shows how the architecture is divided into a framework layer and the three layers known from the model/view/controller design pattern [64, 89]. Also shown in Figure 6.15 is the relationship between these conceptual layers and their instances in our NAVIGATOR and GRAPH example. Figure 6.16 shows that only the framework itself and those parts of a plug-in that are in the control layer have a connection to the ToolBus. The NAVIGATOR and GRAPH plug-ins have no direct connection to each other, but they do have access to the GUI. Both framework and plug-ins have their own connection to the ToolBus, each bound by their own ToolBus interface definition.

Framework The *framework* layer contains the basic architecture which allows an arbitrary number of GUI components to be plugged into the system. Subsequent interaction with those plug-ins is handled by the control layer.

Control The *control* layer bridges control of the individual instances between the model of an instance and the ToolBus. Each plug-in controls an arbitrary number of actual views. For example, the GRAPH plug-in is responsible for two different graphs.

Model The *model* layer holds the data model for one or more views and translates events from any of its views (e.g., a mouse event) into data specific events (e.g., a module event). In the example, the NAVIGATOR plug-in renders both a global and a local view based on the same model.
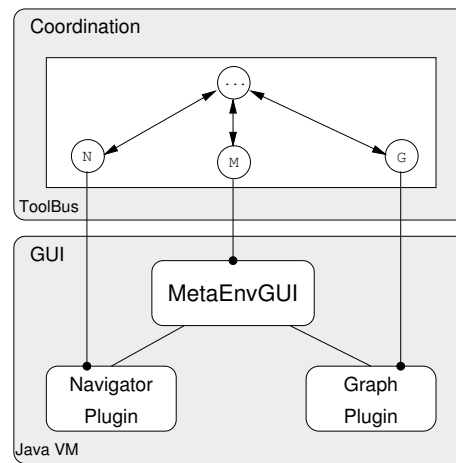
Figure 6.16: Connecting Framework and Plug-ins to the ToolBus.

View The *view* layer actually renders the data from the model in a GUI specific way, and it allows interaction with the data (e.g., by passing mouse events on a tree view).

Selecting a module, e.g., by clicking the mouse on a module name in the NAVIGATOR, is now achieved in the following steps:

- The mouse-clicked event is received by one of the *view* components (in this case the NAVIGATOR) and is forwarded to the *model* layer;

- The *model* layer translates the mouse-clicked event on a specific coordinate into a module-clicked event on a specific module, which is passed to the *control* layer;

- In the *control* layer, which bridges the GUI framework to the ToolBus, the actual event handling takes places via regular ToolBus processes and ToolBus communication;

- The control layer in the ToolBus sends out two notifications: one to update the selected *module* in the NAVIGATOR, and one to update the selected *node* in the Import Graph view of the GRAPH plug-in;

- The NAVIGATOR and GRAPH plug-in in the control layer forward the events to the NAVIGATOR model and the Import Graph model, respectively;

- The NAVIGATOR model updates both of its views and the GRAPH view showing the Import Graph updates its selected node.

The NAVIGATOR and GRAPH plug-ins deal with events and selections at their own conceptual level. That is, the NAVIGATOR plug-in deals with *modules*, and the GRAPH plug-in deals with *nodes*. The real interaction between the plug-ins is orchestrated, at

```
00  process SetCurrentModule(Module: str) is
01    snd-msg(nav-select-module(Module))
02    . snd-msg(gp-select-node("import-graph", Module))
03
04
05  process NavigatorSelectionHandler is
06  let
07    Module: str
08  in
09    subscribe(nav-module-clicked(<str>)) .
10    (
11      rec-note(nav-module-clicked(Module?))
12      . SetCurrentModule(Module)
13    ) * delta
14  endlet
15
16  toolbus(NavigatorSelectionHandler)
17
18
19  process ImportGraphSelectionHandler is
20  let
21    Node: str
22  in
23    subscribe(gp-node-clicked("import-graph", <str>))
24    .
25    (
26      rec-note(gp-node-clicked("import-graph", Node?))
27      . SetCurrentModule(Node)
28    ) * delta
29  endlet
30
31  toolbus(ImportGraphSelectionHandler)
```

Figure 6.17: Plug-in interaction coordinated by ToolBus processes.

the ToolBus level. Figure 6.17 shows the ToolBus script that notifies the Navigator and Graph plug-ins when the *currently selected module* changes.

A noteworthy issue in Figure 6.17 is in lines 02 and 27 where *nodes* and *modules* are treated as if they were the same thing. In the Meta-Environment a node in the Import Graph and the module it refers to are identical, which allows this trivial "translation" between nodes and modules. The example still holds if a more complex mapping is needed, however. This translation would be applied just before line 02 to map a module to a node, and its inverse mapping would be applied just before line 27 to map a node to a module.

6.9   Summary and Conclusions

We have studied a number of popular, contemporary applications which all employ some sort of extension mechanism to allow user modification. This study allowed us to identify four main categories of "plug-in patterns": *decoration*, *delegation*, *mediation*, and *adaptation*. We described our lightweight plug-in architecture which we implemented in the Meta-Environment, an instance of a *mediating* extension architecture.

We now return to the questions from Section 6.1.2 and try to answer them in the remainder of this Section.

6.9.1   Plug-in Techniques

How do current software systems achieve various levels of extensibility?

For *decoration* purposes, a combination of graphics files and metadata files describing where in the application to deploy those graphics, are bundled. The use of the resulting *theme* is then fully controlled by the application.

Applications that need to *delegate* specific functionality (e.g., the decoding of a media stream) do so by invoking a method in plug-ins, which are embedded in the application. Plug-ins are written in the same programming language as the application itself (usually C++). The application does not allow arbitrary extension, but dictates exactly where plug-ins can be hooked in, and what purpose they serve.

Application frameworks which have extensibility as one of their central goals (e.g., Eclipse) *mediate* these extensions by allowing them to register centrally controlled extension points. The framework can verify whether prerequisites of plug-ins are fulfilled and releases control to the specific plug-in whenever the application uses one of the extension points. These plug-ins are written in the same programming language as the application itself (usually `Java`). Extension points can be defined arbitrarily, the platform does not dictate when or where plug-ins can allow themselves to be extended.

Finally, some applications (e.g., Mozilla) can be *adapted* to the user's needs by allowing extensions written in a scripting language (`Javascript`) to be interpreted by the application. This class of extensions is not intended to be used to add a significant portion of functionality to the application, but allows for minor modifications. The scripting nature of *adaptive* extensions makes them feel more lightweight than the compiled nature of their *delegated* and *mediated* cousins.

6.9.2　Plug-in interaction

How do current software systems deal with consistency and interaction between the core of the system and the plug-ins, and between different plug-ins?

The *mediating* frameworks we studied (Eclipse, JPF) allow for a hierarchy of plug-ins. Once the framework releases control of the application to the plug-in, that plug-in itself is responsible for its own (sub-)extension points. The plug-in uses the framework to locate and instantiate these sub-plug-ins, but when and how to invoke them is left to the plug-in itself.

Interaction between *adaptive* extensions is not explicitly dealt with by the framework at all. Because of the freedom inherent in the scripting language, extensions can dynamically override functionality in other extensions. This can easily lead to potential collisions when different extensions want to affect the same functionality of the application in a conflicting way.

6.9.3　The ToolBus in a Plug-in Framework

How do current plug-in mechanisms relate to a system using a component coordination architecture such as the ToolBus?

The ToolBus with its process algebra based scripting language to coordinate the communication between independent component is very flexible. The plug-in patterns that are most interesting from a coordination perspective, i.e., all but the decorative pattern which does not affect the application's functionality, can all be implemented using the ToolBus.

In the adaptation and mediation systems we studied, plug-ins are always implemented in the language of the application itself. In the adaptation systems we studied, a special scripting language was used to write the plug-ins in. A system built on the ToolBus architecture allows the plug-in to be implemented in the programming language that is most suitable for the job. In the case of the Meta-Environment the GUI parts are implemented in Java, but because each GUI part of the plug-in has its own ToolBus connection, other tools could also be used in the implementation of the plug-in.

Control between framework and plug-ins, as well as between different plug-ins is all coordinated at the ToolBus level. The ToolBus scripting language is highly suited for this job. Because coordination is centralized, the application maintains a firm grip on how plug-ins interact, but it is flexible enough to allow arbitrary changes. The rigorous separation of the *coordination* and *computation* concerns in a ToolBus application allows plug-ins to be maximally oblivious of the rest of the application, and at the same time allows the application fine grained control over its core behavior and any extensions to it.

6.9.4　Impact on the ASF+SDF Meta-Environment

In order to understand the impact of applying the plug-in concepts we studied and our implementation thereof in a ToolBus setting, we zoom out from the technical details and show a higher level *before-after* scenario.

(a) Before Plug–in architecture   (b) Using Plug–in architecture
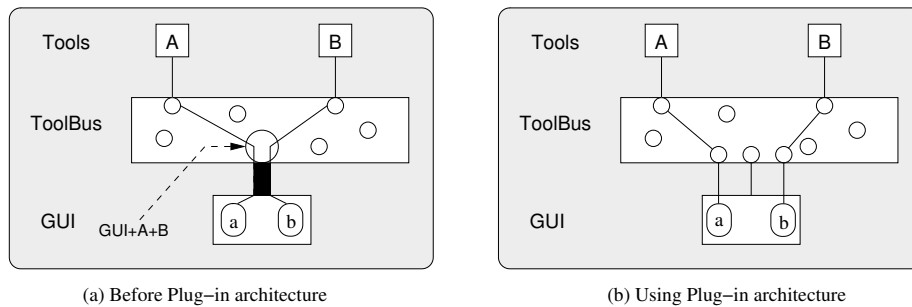
Figure 6.18: ToolBus interface of the GUI before and after application of the plug-in architecture.

Before using the plug-in architecture, the Meta-Environment was a collection of tools, coordinated by a ToolBus script. The GUI was a single tool, and all user inter-action with the system was in one way or another described in the ToolBus interface definition of this central GUI. Whenever new functionality was added to the system, the GUI had to be adapted, and its ToolBus interface extended to cope with the new functionality. In a way, the complexity of the ToolBus connection to the GUI grew bigger and bigger as the system evolved. Figure 6.18(a) shows a system before the plug-in architecture using the ToolBus, two tools called A and B which each require user interaction, and the monolithic GUI. The figure emphasizes how the ToolBus in-terface of the GUI grows as more tools need access to a portion of the GUI. Whereas the GUI *should be* just like any other tool, it in fact has become a bottleneck.

Now that we have implemented the plug-in architecture we presented here, the GUI has a single, fixed ToolBus interface. All it does besides offering basic functionality to shutdown the application, is facilitate arbitrary GUI units, called plug-ins to be instan-tiated in the central visible GUI component. When user-interaction with the system needs to be added, a new plug-in with its own ToolBus interface can be created and added to the system, without intervention in any of the existing tools. Figure 6.18(b) schematically shows how the GUI parts of our tools A and B, although plugged into the GUI, now have their own ToolBus interface. The GUI interface no longer needs to grow for each tool that is added to the system.

After the introduction of the plug-in architecture in the ASF+SDF Meta-Environ-ment we were able to refactor and migrate the tool-specific portions of the GUI to corresponding tools. By eliminating the dependency of the GUI on all other tools that required user interaction, both the source code of the GUI as well as parts of the build infrastructure (mainly Makefiles, and `ant` build scripts) were simplified. During this refactoring we encountered several hidden and unwanted dependencies in portions of the user interface where implementation details of one tool had emerged in another. The strict separation into plug-ins reinforces the separation into tools that is already present on the ToolBus level, but which had evaporated in the GUI.

Currently the ASF+SDF Meta-Environment has about ten plug-ins. So far, none of these are optional. Each plug-in is important to the core functionality of the system. Therefore, at present, the main advantage we gain from using a plug-in architecture

is the much improved *looseness* in binding and coupling of the various parts of the system. However, now that we have this open system, first of all we ourselves can build extensions to the Meta-Environment more easily, and second it will finally be possible to deal with third party extensions (e.g., developed by end-users of the system) in a systematic way.

6.9.5   Contributions

The contributions of the plug-in architecture we presented in this chapter can be summarized as follows:

- Application of the plug-in architecture solves the typical modularization bottleneck found in an *omniscient GUI* approach;

- In contrast to the Eclipse platform where plug-ins are entirely `Java` based, we can implement plug-ins in various programming languages: only the part of the plug-in that is rendered in the GUI is written in `Java`;

- Contrary to other architectures where plug-ins are free to do whatever they want with the application, we keep a firm, global grip on all interaction between plug-ins by having the ToolBus coordinate all the interaction.

Part IV

Conclusion

C H A P T E R 7

Conclusions

In the introduction (Chapter 1) we explained that the work in this thesis is structured around two central research questions.

Then, in Chapters 2 and 3 we presented work related to the answer of the first research question about space efficient data exchange and the type-safe access thereof.

Subsequently, in Chapters 4, 5, and 6 we showed the results of some studies done in the context of the second research question about the implications of using component coordination techniques to develop an interactive development environment.

In this chapter we reflect on the presented work, and summarize our findings and try to answer the research questions.

7.1   Space efficient, type-safe data exchange

> **Research Question 1:** *How can structured data be exchanged between heterogeneous components in a space efficient, type-safe way?*

In Chapter 2 we presented a design and implementation of the Annotated Term (ATerm) data type, which utilizes maximal subterm sharing. This technique ensures a space efficient encoding for data that are structured in a tree like form. Not only are ATerms represented in an efficient way in memory, they can also be exchanged between components in a very compact way. By keeping the maximal subterm sharing in the serialized representation, the actual amount of data that needs to be exchanged between components is kept low, and the effort needed in the destination component to rebuild the internal representation of the data is cheaper than if this component had to rediscover the sharing present in the data itself.

In Chapter 3 we presented the design and implementation of a code generator which generates a type-safe access layer on top of the generic, untyped ATerm data type. By generating the method names based on descriptive names from the formal definition (grammar) of the data type, instead of numbering arguments or using other heuristics, developers can abstract from the underlying representation. By generating the access

layer code for different programming languages, using the same data definition as contract, the error prone marshalling and unmarshalling of data no longer has to be done by the developer. The result is a software system that first of all contains less code that needs to be maintained manually, and second is closer to the abstraction level of the data description, than of its underlying representation.

In more general terms we developed a highly optimized low-level implementation for a specific set of data types, yet we maintain an abstract view on this data type by generating a high-level type safe access layer on top of the data representation.

7.2 Building an IDE using a coordination architecture

> **Research Question 2:** *What are the implications of using component coordination techniques on the architecture of interactive software development environments?*

In an attempt to answer this research question, we studied three particular areas.

First, there is the impact on the coordination architecture itself. We looked at the issues relevant to interactive software development environments, from a component coordination architecture point of view. Section 7.2.1 details our findings.

The second area we studied is that of connecting off-the-shelf available components to an IDE built with a component coordination architecture. In Section 7.2.2 we summarize our ideas on what kind of infrastructure is needed in order to be able to use off-the-shelf components in a robust way, and to stay clear of changes in those components as they develop over time.

The third area we studied is closely related to the *decoupling paradox* explained in Section 1.2: after decoupling components, how can we have a single, centralized graphical user interface that allows users to interact with the system, without this GUI knowing about all the components. We reflect on how the plug-in architecture we developed is a solution to this issue in Section 7.2.3.

7.2.1 Component coordination techniques for an IDE

In Chapter 4 we showed our experiences in using a component coordination architecture as the basic architecture to build an interactive software development environment. We showed how the use of the ToolBus coordination architecture resulted in a more open implementation of the environment. Some of the individual components are reused outside of the context of the Meta-Environment, but also other environments are built using the same ToolBus based architecture.

Also in Chapter 4 we studied possible changes to both the ToolBus architecture and the scripting language, based on the Meta-Environment case study. From these results we can conclude that a scripting language based on process algebra, such as the one used to model the application in the ToolBus environment, is powerful and expressive enough to describe the coordination patterns found in an interactive application. One area where the language currently lacks proper support is the handling of errors

and exceptions. Although primitives are available to deal with errors, these are very low-level. A higher abstraction level to deal with exceptions, such as is found in the `Java` programming language, is desirable. This can be added without giving up the fundamental process algebra nature of the ToolBus script language.

We realize that the pass-by-value way of passing data between components in the ToolBus is a bottleneck, and studied how related architectures bypass this bottleneck. However, having a central architecture that takes care of garbage collection helps avoid the difficult problem area of distributed garbage collection. In the context of building user interaction-bound applications (such as a software development environment), the use of pass-by-value techniques does not pose too much of a performance issue. This changes drastically if the ToolBus were to be used in a high-throughput system, e.g., to coordinate webservices with high volumes of requests.

7.2.2   Using off-the-shelf components in a robust way

In Chapter 5 we studied the problem domain of connecting off-the-shelf components to the ToolBus. Each *foreign* component needs some sort of glue to get it to work in a particular architecture setting. Even if components have the same functional interface, their behavioral interface need not be equal. In the context of reusing third party editing facilities, we showed how a proxy component with a certain functional interface can be used to adapt other components with the same functional interface, but with a different behavioral interface. The proxy component hides operational details such as starting up and communicating with the third party component. We showed how we can avoid having to adapt the foreign component's source code by using the proxy component approach.

7.2.3   A central GUI for decoupled components

In Chapter 6 we have studied a phenomenon common in software applications with user interaction using a (graphical) user interface. Where individual pieces of work can be done in isolated components, the user interaction with all these components has to be centralized. In a component coordination architecture setting where we strive to decentralize computational components, we are faced with the paradox of having a central component, the GUI, which needs to offer the user a view on the state of all other components. We designed and implemented a central, open, GUI architecture that allows separate pieces of the GUI to be plugged in during execution of the application. In doing so, we separate user interaction concerned with the application at a higher level (e.g., startup, shutdown, loading a plug-in) from user interaction targeted only at a specific part of the application. We argue that this architecture is open for all sorts of extensions to the application, but that our approach does not break the idea that the central coordinator (the ToolBus in our case) is ultimately responsible for coordinating component interaction.

7.2.4 Applicability Considerations

Although the primary research context for the architecture described in this thesis has been an academic one (the ASF+SDF Meta-Environment), the applicability of the architecture is by no means restricted to just that environment. However, we do note that there are various *implementation* issues related to the ToolBus, that would have to be addressed before it can be realistically used in a non academic setting. For example, *security* would have to be improved drastically for any conceivable deployment in, e.g., a banking or online commercial applications. Similarly, *availability* would have to be improved for any uses in, e.g., health critical systems. For a web service, various optimizations will be needed to ensure a timely response, even when thousands of requests per second need to be handled.

However, with a solid base in fundamental Process Algebra [7], the ToolBus can be a very powerful instrument in dealing with scenarios of components which would otherwise be more difficult to solve. Interaction cycles, e.g., mutually recursive calls between components, can be dealt with in ToolBus scripts in virtually the same way as these concerns can be expressed in Process Algebra.

7.3 Future Work

In this section we describe some thoughts on future work. In particular we propose some ideas on how the work on the generation of APIs described in this thesis can be applied in a ToolBus setting to achieve more application wide type-safety in Section 7.3.1. The work on our plug-in framework leads to some questions on the granularity and dynamics of plug-ins, which we describe in Section 7.3.2.

7.3.1 Application wide type-safety

Data is exchanged between components using ATerms, a generic term representation. Accessing this data inside components is done in a type-safe manner using the generated APIs. At the coordination level, in the ToolBus scripts, however, much of this type-safety is lost. The focus is on coordinating message patterns between components, and the data exchanged between components is usually referred to using generic term patterns. Figure 7.1 illustrates how data-access in a ToolBus setting is roughly divided into two layers. In the ToolBus scripts, variables of the *generic* type `term` are used to refer to data that is in fact of a more application *specific* type. The tools use generated access libraries (as described in Chapter 3) to access the same data in a much more type-safe fashion.

Current practise in ToolBus scripts Consider the following example ToolBus script, taken from the ASF+SDF Meta-Environment:

```
process ParseTreeHandler(ModuleId: term, Path: str) is
let
  ErrorMessage: term,
  ParseError: term,
```
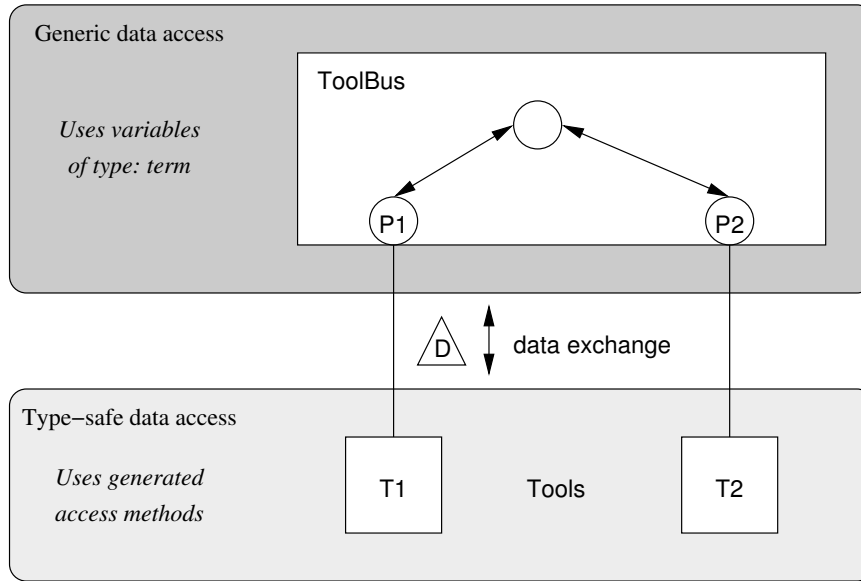
Figure 7.1: Generic data transfer in ToolBus and type-safe access in the tools.

```
  ParseResult: term,
  Pid: int,
  Summary: term
in
  Pid := process-id
  .
  (
    (
      rec-msg(parse-tree(Pid, ParseResult?))
      . RemoveSummary("sglr", Path)
    +
      rec-msg(parse-forest(Pid, ParseResult?, ErrorMessage?))
      . AddFilenameToSummary(ErrorMessage, Path, ParseError?)
      . MakeErrorSummary("sglr", Path, [ParseError], Summary?)
      . RemoveSummary("sglr", Path)
      . DisplaySummary(Summary)
    )
    . snd-msg(parse-handler-done(Pid, ModuleId, ParseResult))
  +
    rec-msg(parse-error(Pid, ErrorMessage?))
    . AddFilenameToSummary(ErrorMessage, Path, ParseError?)
    . MakeErrorSummary("sglr", Path, [ParseError], Summary?)
    . RemoveSummary("sglr", Path)
    . DisplaySummary(Summary)
    . snd-msg(parse-handler-done(Pid))
  )
endlet
```

147

All variable occurrences of type `term` have been highlighted in **boldface**. These occurrences mark potential problem areas, because in principal *any* kind of data can be passed through a `term` variable.

Current possibilities   The `term` data type (supported by its implementation in several programming languages) is a very useful part of the ToolBus architecture. It allows components to exchange data in a generic way, without having to bother with issues such as marshalling input and output by parsing (input) and pretty printing (output). The open character of the `term` type allows access to the data exchanged exchanged between components, not only at the component level, but also at the ToolBus script level. This means the ToolBus script can inspect data (e.g., for debugging purposes), annotate data (e.g., for statistical measurements), and even modify data (e.g., to connect two components with similar, but not quite identical interfaces). In other words, we strongly support the generic and open nature of the `term` data type.

However, the use of `term` typed variables in a ToolBus script is, in a way, equivalent to using variables of type `Object` in `Java`, or using `void *` pointer variables in a `C` program. In some programming languages (e.g. in `C`) the use of the generic type is the only way to deal with generic implementations. Implementing a hashtable in `C` for example, will rely on the use of `void *` pointers. However, in programming languages that support strict typing, it is usually a good idea to use the strict types whenever possible, as it allows static detection of programming errors related to type errors.

Just as the implementation of a parser in `Java` would probably use types (classes) like `ParseTable`, `ParseTree`, and `ParseError`, we would like to be able use the same types in ToolBus scripts.

In particular, we would be interested in using the abstract data types discussed in Chapter 3 in some form. This would allow us to use the exact same data type names in ToolBus scripts, as we do in our tools.

Apart from the generic `term` type, the ToolBus already allows more strict typing, but using such types relies on structural knowledge of the actual term *representation*, rather than the mere *name* of the type. Instead of declaring a variable of type `term`, a term pattern can be used. A match between a `snd-msg` and `rec-msg` with a result variable declared to be of a specific term pattern will only succeed, if the sent value matches the receiving term pattern.

For example, if a variable is declared as follows:

```
let B: book
```

and it is subsequently used to receive requests to store a book:

```
rec-msg(store-item(B?))
```

then only terms that match the signature

```
snd-msg(store-item(book(<term>)))
```

will match. Note how the type of the variable (`book`) is used to restrict the matching pattern.

In a similar way, more complex patterns can be used as the pattern for the variable.

```
let B: book(title(<str>), author(name(<str>)))
```

Recognizing the pitfall   Unfortunately, by using this type mechanism, we have to manually enter *structural* information about our data, and scatter this knowledge throughout the entire script. As we discussed in Chapter 3, we consider this bad practise. Instead we would like to abstract from the data representation and use type names instead. At the same time, we might be interested in having more of the type-safe access functionality derived from the data type (again, similar to the practise described in Chapter 3).

Suggested approach   To this end, we propose to add to the existing sections in a ToolBus script that deal with *process* definitions and *tool* definitions, a section that reflects the *data* definitions used in the script. As an example, consider the following script, where the data type related changes are given in **boldface**:

```
01 tool store is { command = "/path/to/store" }
02
03 data is { url = "http://www.cwi.nl/path/to/book.adt"}
04
05 process Store is
06 let
07   B: @Book,
08   S: store
09 in
10   execute(store, S?)
11   .
12   (
13     rec-msg(store-item(B?))
14     . snd-do(S, store-book-by-author(@GetBookAuthor(B), B))
15   +
16   ...
17   ) * delta
18 endlet
```

In line 03 we tell the ToolBus that it needs to include the data definition (`book.adt`) specified at a given url. This ADT would be the same ADT that we used throughout Chapter 3, so among others, it could be generated from a syntax definition of the data type. Suppose for this example that `book.adt` contains the entry `[Book, default, book(<title(str)>,<author(str)>)]` then the generated C library contains a definition for the type `Book` as well as access functions similar to the following:

```
typedef struct _Book *Book;
ATbool hasBookTitle(Book book);
char *getBookTitle(Book book);
```

149

```
char *getBookAuthor(Book book);
Book setBookAuthor(Book book, char *author);
```

Similar to using the `Book` type and its access functions in C, we now declare a variable of that type in our ToolBus script, as is done in line 07 above. In this example, we prefixed the type with a syntactic marker (the @ sign) so the ToolBus knows it has to lookup the data type `Book` in the ADT previously declared in the data section.

In line 14 we use an access method similar to the way we used it in Chapter 3. In this case, it would extract the author from the `Book` typed parameter that was passed.

The fundamental change in the ToolBus with respect to the way it deals with data is that, where we currently only deal with data in a very generic way, we will allow an *explicit data definition*. Static type checking of the scripts should now take the data types in the supplied data file into account. In the current situation (almost) every parameter is of type `term`, meaning that errors due to type mismatches are either not found at all, or at best detected at runtime because expected messages do not match. In the new situation, parameterized processes can have strictly typed parameters, meaning that more programming errors in the scripts can be found statically.

Implementation considerations   Two approaches for the implementation can be considered: a *generational* approach as we did in ApiGen in this thesis, or an *interpretative* approach.

As the ToolBus has primitives to traverse the term structure, it would be possible to generate ToolBus code that implements processes in the same way, the C and `Java` implementations are generated by ApiGen.

Given that we have a ToolBus implementation in C and at least have a prototype implementation in `Java`, an approach where we re-use the ApiGen-generated libraries is perhaps more appealing. As `Java` has well defined *reflection* semantics and a rich reflection API, it should be possible and fairly straightforward even, to map calls in the ToolBus script (e.g. the **@GetBookAuthor** from the example) to the corresponding method in the generated `Java` API. If it is desirable to have this functionality in the current C implementation, the same link can be made by having a (generated) symbol table which maps the ToolBus invocation to a specific function in the generated C library.

Expected results   By promoting type-safety in the ToolBus scripts, we effectively raise the level of type-safety in the entire ToolBus driven application. Figure 7.2 illustrates how we arrive at an application which is mostly type-safe in both the individual tools *and* the coordinating ToolBus script, but which still has the freedom to use generic types where they are needed (e.g., to implement a generic term caching facility).

7.3.2   About ToolBus and plug-ins

The desire to leave more and more things in a software application open to changes by the user, leads to the need for a system that is highly extensible and which offers a means to change things on a varying scale of granularity. Sometimes only small things need to be changed (e.g., the order of two menu items in a menu bar), and
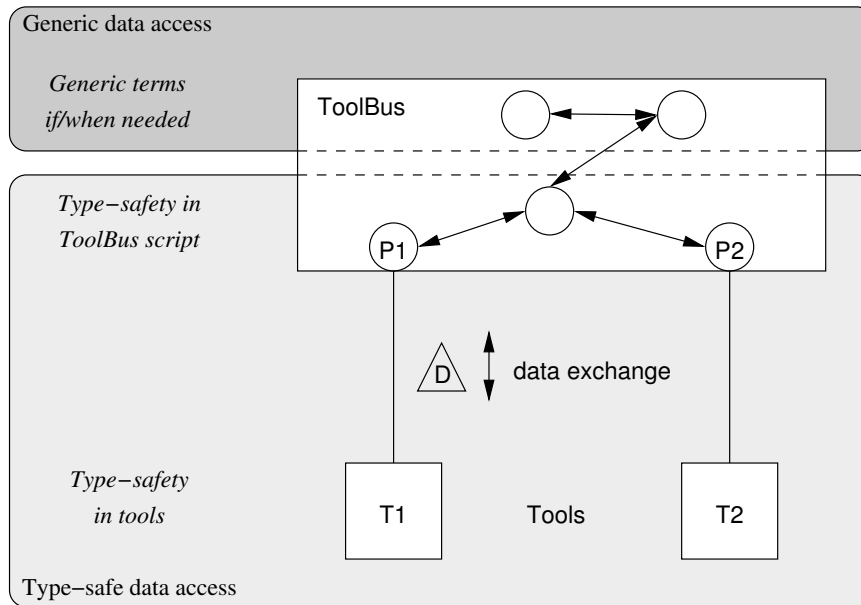
Figure 7.2: Improved type-safety in the ToolBus coordinated application.

sometimes entire components need to be changed (e.g., switching the rendering engine of a graphical application from one that only supports 2D graphics to one that supports 3D graphics as well). At the same time, it is of vital importance that consistency of the software system as a whole is guaranteed.

Currently, the extensibility offered by our ToolBus driven plug-in architecture, is of a fairly static nature. The ToolBus assumes a *closed world* of processes, tools (and after the previous section, of *data* as well). Although it is fully dynamic in the sense that new process instances can be created and that running processes can end at certain points in the execution of a ToolBus script, their *definition* is fixed from the moment the ToolBus is started. Our plug-in architecture is thus extensible in the sense that new plug-ins (`Java` components possibly accompanied by ToolBus scripts) can be added to the system, but the system has to be *restarted* whenever an extension is added.

Given the current nature (static, and large chunks of changes through plug-ins), we are left with some interesting questions:

Plug-in dynamics   What are the fundamental requirements of an extensibility architecture that can be dynamically extended and updated? It may be easy to understand how we can build a software system that can add new components without needing a restart, but what happens if currently running components need to be updated? What happens when we try to remove such components, can we still guarantee consistent behavior of the system? What role could existing transaction systems play in such a framework?

Plug-in granularity   In our system, we are able to replace chunks of software at the component level, as long as both components abide by the same functional (ToolBus) and data (ADT) interface. For example, we could easily replace our C implementation of the parser by one that is implemented in `Java`. We can replace the interaction between existing components by adding, replacing, or removing parts of the ToolBus script that guides the application. These are fairly course grained changes to the system. But what happens if a user wants to change something small, such as the order in the GUI menubar of two menu items which originate from two different plug-ins? Currently the GUI part of the framework dictates that menu items are added in the same order the plug-ins were loaded. In order to add this flexibility to our system, we would currently have to change the architecture. Would it be possible to have these kinds of system *adaptations*, without giving up our our central control model? How small can the unit of extension become, without loosing control over the consistency of the application?

Plug-ins as ToolBus primitive?   The ToolBus Next Generation studies have resulted in a new implementation of the ToolBus in `Java`. This implementation is nearing completion just at the time where we introduced the GUI plug-in framework to the Meta-Environment.

It would be very interesting to study if the plug-in framework can somehow be merged into the ToolBus itself. And, if so, what the consequences would be for the way we deal with with GUI interaction between plug-ins and the rest of the system.

Currently, the plug-in framework is implemented as a separate component. That is, it is just an ordinary tool, like any other tool that is connected to the ToolBus. However, especially in the context of using the ToolBus for applications that use a GUI, it may be interesting to promote this functionality to a primitive similar to the `execute` primitive which starts up an external tool.

Will integrating the GUI plug-in framework in the ToolBus be only a shift from external tool to ToolBus primitive? Or will it lead to the insight that it is indeed desirable to have GUI concepts and primitives at the level of the coordination architecture itself?

Syntax and Interface of ATerms

This appendix lists the concrete syntax (in SDF) of ATerms, and the interface of the ATerm library.

A.1 Concrete Syntax of ATerms

A formal definition of the concrete syntax of ATerms using the syntax definition formalism SDF [72] is presented here. Note that there is no concrete syntax defined for blobs, because a humanly readable representation of blobs depends on the type of data stored in the blob.

A.1.1 ATerms.sdf

```
module languages/aterm/syntax/ATerms

imports
  languages/aterm/syntax/IntCon
  languages/aterm/syntax/RealCon
  basic/StrCon
  basic/IdentifierCon

exports
  sorts AFun ATerm Annotation

  context-free syntax
    StrCon                        -> AFun {cons("quoted")}
    IdCon                         -> AFun {cons("unquoted")}

  context-free syntax
    IntCon                        -> ATerm {cons("int")}
    RealCon                       -> ATerm {cons("real")}
    fun:AFun                      -> ATerm {cons("fun")}
    fun:AFun "(" args:{ATerm ","}+ ")" -> ATerm {cons("appl")}
    "<" type:ATerm ">"            -> ATerm {cons("placeholder")}
    "[" elems:{ATerm ","}* "]"    -> ATerm {cons("list")}
    trm:ATerm Annotation          -> ATerm {cons("annotated")}

  context-free syntax
    "{" annos:{ATerm ","}+ "}"    -> Annotation {cons("default")}
```

153

A.1.2   IntCon.sdf

module languages/aterm/syntax/IntCon

imports
  basic/Whitespace
  basic/NatCon

exports
  sorts IntCon
  context-free syntax
    NatCon   -> IntCon {cons("natural")}
    pos:"+" NatCon -> IntCon {cons("positive")}
    neg:"-" NatCon -> IntCon {cons("negative")}

A.1.3   RealCon.sdf

module languages/aterm/syntax/RealCon

imports
  languages/aterm/syntax/IntCon

exports
  sorts OptExp RealCon

  context-free syntax
    "e" IntCon   -> OptExp {cons("present")}
                              -> OptExp {cons("absent")}

    base:IntCon "."
         decimal:NatCon
         exp:OptExp   -> RealCon {cons("real-con")}

A.1.4   StrCon.sdf

module basic/StrCon

exports
  sorts StrCon StrChar

  lexical syntax
    "\\n"   -> StrChar {cons("newline")}
    "\\t"   -> StrChar {cons("tab")}
    "\\\""   -> StrChar {cons("quote")}
    "\\\\"   -> StrChar {cons("backslash")}
    "\\" a:[0-9]b:[0-9]c:[0-9] -> StrChar {cons("decimal")}
    ~[\0-\31\n\t\"\\]   -> StrChar {cons("normal")}

    [\"] chars:StrChar* [\"] -> StrCon {cons("default")}

A.1.5   IdentifierCon.sdf

module basic/IdentifierCon

exports
  sorts IdCon

154

```
lexical syntax
  head:[A-Za-z] tail:[A-Za-z\-0-9]* -> IdCon {cons("default")}

lexical restrictions
  IdCon -/- [A-Za-z\-0-9]
```

A.1.6   NatCon.sdf

module basic/NatCon

exports
  sorts NatCon

```
  lexical syntax
    [0-9]+ -> NatCon {cons("digits")}

  lexical restrictions
    NatCon -/- [0-9]
```

A.1.7   Whitespace.sdf

module basic/Whitespace

exports
```
  lexical syntax
    [\ \t\n\r] -> LAYOUT {cons("whitespace")}

  context-free restrictions
    LAYOUT? -/- [\ \t\n\r]
```

A.2   Level 2 interface for ATerms

The operations described in Section 2.2 are not sufficient for all applications. Some applications need more control over the underlying implementation, or need operations that can be implemented using level one constructs but can be expressed more concisely and implemented more efficiently using more specialized constructs.

We have therefore designed a level 2 interface that is a strict superset of the level 1 interface described in Section 2.2. Some new datatypes are introduced, as well as some new operations on ATerms.

The level 2 interface introduces 7 new datatypes. Except for the auxiliary datatype AFun for representing function symbols, they are subtypes of the ATerm datatype, and implement the different term types. These subtypes allow us to introduce operations that are only valid for one specific term type, instead of the general ATerm operations described earlier.

ATermInt: This datatype represents integer terms. The operations on ATermInt are:

- ATermInt ATmakeInt(Integer v): Construct a new integer term corresponding to the integer value v.

155

- `Integer ATgetInt(ATermInt i)`: Retrieve the value of an integer term.

ATermReal: This datatype represents real-number terms. The operations on ATermReal are:

- `ATermReal ATmakeReal(Real v)`: Construct a new real term.

- `Real ATgetReal(ATermReal r)`: Retrieve the value of a real term.

AFun: An AFun consists of a string defining the function name, an arity, and an indication whether the symbol name is quoted or not. The operations on symbols are:

- `AFun ATmakeAFun(String nm, Integer ar, Boolean q)`: Construct a new symbol. If a symbol with the given name *nm*, arity *ar*, and quotation *q* already exists, the existing symbol is returned. Otherwise a new symbol is created and returned. AFuns are also subject to garbage collection in order to avoid long running (interactive) programs from slowly running out of symbols.

- `String ATgetName(AFun s)`: Retrieve the name of symbol *s*.

- `Integer ATgetArity(AFun s)`: Retrieve the arity of a symbol.

- `Boolean ATisQuoted(AFun s)`: Check if a symbol is quoted.

ATermAppl: This datatype represents function applications consisting of a function symbol and a number of arguments. The operations on this datatype are:

- `ATermAppl ATmakeAppln(AFun f, ATerm a_0, ..., ATerm a_{n-1})`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to construct a new function application with the given function symbol f and arguments.

- `ATermAppl ATmakeAppl(AFun f, ATermList as)`: Construct a new function application with the given function symbol f and a list of arguments *args*

- `AFun ATgetFun(ATermAppl ap)`: Retrieve the function symbol of a function application.

- `ATerm ATgetArgument(ATermAppl ap, Integer n)`: Retrieve a specific argument.

ATermList: This datatype represents the binary list constructor. Element indices start at 0. Thus a list of length n has elements $0, \ldots, n-1$. The operations on ATermList are:

- `ATermList ATmakeListn(ATerm e_0, ..., ATerm e_{n-1})`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to quickly construct small lists of terms.

156

- `Integer ATgetLength(ATermList `*`l`*`)`: Retrieve the length of *l*.

- `ATerm ATgetFirst(ATermList `*`l`*`)`: Retrieve the first element of list *l*.

- `ATermList ATgetNext(ATermList `*`l`*`)`: Retrieve all but the first element of list *l*.

- `ATermList ATgetPrefix(ATermList `*`l`*`)`: Retrieve all but the last element of list *l*.

- `ATerm ATgetLast(ATermList `*`l`*`)`: Retrieve the last element from list *l*.

- `ATermList ATgetSlice(ATermList `*`l`*`, Integer `*`frm`*`, Integer `*`to`*`)`: Retrieve the portion of list *l* from position *frm* through *to* − 1.

- `Boolean ATisEmpty(ATermList `*`l`*`)`: Check if list *l* contains zero elements.

- `ATermList ATinsert(ATermList `*`l`*`, ATerm `*`e`*`)`: Insert a single element *e* at the start of list *l*.

- `ATermList ATinsertAt(ATermList `*`l`*`, ATerm `*`e`*`, Integer `*`i`*`)`: Insert a single element *e* at position *i* in list *l*.

- `ATermList ATappend(ATermList `*`l`*`, ATerm `*`e`*`)`: Append a single element *e* to the end of list *l*.

- `ATermList ATconcat(ATermList `l_1`, ATermList `l_2`)`: Concatenate lists l_1 and l_2.

- `Integer ATindexOf(ATermList `*`l`*`, ATerm `*`e`*`, Integer `*`i`*`)`: Search for an element *e* in list *l* and return the index of the first location where *e* is present. Start searching at index *i*. If the element is not present, return −1.

- `Integer ATlastIndexOf(ATermList `*`l`*`, ATerm `*`e`*`, Integer `*`i`*`)`: Search backwards for element *e* in list *l*, and return the index of the last location where the element is present. Start searching at index *i*. If the element is not present, return −1.

- `ATerm ATelementAt(ATermList `*`l`*`, Integer `*`i`*`)`: Retrieve element at position *i* from list *l*.

- `ATermList ATremoveElement(ATermList `*`l`*`, ATerm `*`e`*`)`: Remove once occurrence of element *e* from list *l*.

- `ATermList ATremoveElementAt(ATermList `*`l`*`, Integer `*`i`*`)`: Remove the element at position *i* from list *l*.

157

ATermPlaceholder: This datatype represents placeholder terms. The operations on ATermPlaceholder are:

- `ATermPlaceholder ATmakePlaceholder(ATerm `*tp*`)`: Construct a new placeholder term.

- `ATerm ATgetPlaceholder(ATermPlaceholder `*ph*`)`: Retrieve the type of this placeholder.

ATermBlob: This datatype represents Binary Large OBject terms. The operations on ATermBlob are:

- `ATermBlob ATmakeBlob(Integer `*n*`, Data `*d*`)`: Construct a new blob term of size *n* and containing data *d*.

- `Integer ATgetBlobSize(ATermBlob `*b*`)`: Retrieve the size of blob *b*.

- `Data ATgetBlobData(ATermBlob `*blob*`)`: Retrieve the data pointer stored in blob *b*.

The memory management of blobs must be done explicitly by the application programmer.

Auxiliary: The level two interface provides functionality to create and manipulate user-defined hash tables.

APPENDIX B

B.1 Concrete Syntax of AsFix

A formal definition in SDF of the concrete syntax for parse trees in AsFix is presented here. Imported modules not listed here are presented in Appendix A.1, as they are also imported as part of the ATerms syntax.

B.1.1 Parsetree.sdf

```
module languages/asfix/syntax/Parsetree

imports
  languages/asfix/syntax/Tree
  languages/aterm/syntax/IntCon

exports
  sorts ParseTree

  context-free syntax
    parsetree(top:Tree, amb-cnt:NatCon) -> ParseTree {cons("top")}
```

B.1.2 Tree.sdf

```
module languages/asfix/syntax/Tree

imports
  languages/asfix/syntax/Annotations
  languages/asfix/syntax/Symbol
  languages/asfix/syntax/Attributes

exports
  sorts Tree Args Production

  context-free syntax
    appl(prod:Production,args:Args) -> Tree {cons("appl")}
    cycle(prod:Production)          -> Tree {cons("cycle")}
    amb(args:Args)                  -> Tree {cons("amb")}
    character:NatCon                -> Tree {cons("char")}

    "[" {Tree ","}* "]"             -> Args {cons("list")}
```

```
prod(lhs:Symbols,
     rhs:Symbol,
     attributes:Attributes)    -> Production {cons("default")}
list(rhs:Symbol)               -> Production {cons("list")}
```

B.1.3   Annotations.sdf

module languages/asfix/syntax/Annotations

imports
  languages/asfix/syntax/Tree
  languages/aterm/syntax/ATerms

exports
  context-free syntax
    Tree Annotation -> Tree {cons("annotated")}

B.1.4   Symbol.sdf

module languages/asfix/syntax/Symbol

imports
  basic/StrCon
  basic/NatCon

exports
  sorts Symbol Symbols CharRange CharRanges

  context-free syntax
```
    "empty"                      -> Symbol {cons("empty")}
    lit(string:StrCon)           -> Symbol {cons("lit")}
    cf(symbol:Symbol)            -> Symbol {cons("cf")}
    lex(symbol:Symbol)           -> Symbol {cons("lex")}
    opt(symbol:Symbol)           -> Symbol {cons("opt")}
    alt(lhs:Symbol, rhs:Symbol)  -> Symbol {cons("alt")}
    seq(symbols:Symbols)         -> Symbol {cons("seq")}
    tuple(head:Symbol, rest:Symbols) -> Symbol {cons("tuple")}
    sort(string:StrCon)          -> Symbol {cons("sort")}
    iter(symbol:Symbol)          -> Symbol {cons("iter")}
    iter-star(symbol:Symbol)     -> Symbol {cons("iter-star")}

    iter-sep(symbol:Symbol,
             separator:Symbol)   -> Symbol {cons("iter-sep")}

    iter-star-sep(symbol:Symbol,
                  separator:Symbol)  -> Symbol {cons("iter-star-sep")}

    iter-n(symbol:Symbol,
           number:NatCon)        -> Symbol {cons("iter-n")}

    iter-sep-n(symbol:Symbol,
               separator:Symbol,
               number:NatCon)    -> Symbol {cons("iter-sep-n")}

    func(symbols:Symbols,
```

160

```
          symbol:Symbol)                -> Symbol {cons("func")}

  varsym(symbol:Symbol)                 -> Symbol {cons("varsym")}
  "layout"                              -> Symbol {cons("layout")}
  char-class(CharRanges)                -> Symbol {cons("char-class")}
  strategy(lhs:Symbol, rhs:Symbol)      -> Symbol {cons("strategy")}

  parameterized-sort(
        sort:StrCon,
        parameters:Symbols) -> Symbol {cons("parameterized-sort")}

context-free syntax
  "[" {Symbol ","}* "]"             -> Symbols {cons("list")}

context-free syntax
  "[" { CharRange ","}* "]"         -> CharRanges {cons("list")}

context-free syntax
  integer:NatCon                    -> CharRange {cons("character")}
  range(start:NatCon, end:NatCon)   -> CharRange {cons("range")}
```

B.1.5   Attributes.sdf

```
module languages/asfix/syntax/Attributes

imports
  languages/aterm/syntax/ATerms

exports
  sorts Attributes Attrs Attr Associativity

  context-free syntax
    "no-attrs"                       -> Attributes {cons("no-attrs")}
    "attrs" "(" attributes:Attrs ")" -> Attributes {cons("attrs")}

  context-free syntax
    "[" {Attr ","}+ "]"              -> Attrs {cons("many")}

  context-free syntax
    "assoc" "(" associativity:Associativity ")"
                                     -> Attr {cons("assoc")}

    "term" "(" aterm:ATerm ")"       -> Attr {cons("term")}
    "id" "(" module-name:StrCon ")"  -> Attr {cons("id")}
    "bracket"                        -> Attr {cons("bracket")}
    "reject"                         -> Attr {cons("reject")}
    "prefer"                         -> Attr {cons("prefer")}
    "avoid"                          -> Attr {cons("avoid")}

  context-free syntax
    "left"      -> Associativity {cons("left")}
    "right"     -> Associativity {cons("right")}
    "assoc"     -> Associativity {cons("assoc")}
    "non-assoc" -> Associativity {cons("non-assoc")}
```

161

B.2 Example generated dictionary file

An abbreviated version of the generated dictionary file for the parse tree syntax (AsFix) of the boolean `and`. Multiple similar lines have been condensed (...).

```c
#include "test_dict.h"

AFun afun0;
AFun afun1;
...

ATerm patternBoolAnd = NULL;

/*
 * afun0 = appl(x,x)
 * afun1 = prod(x,x,x)
 * afun2 = cf(x)
 * afun3 = sort(x)
 * afun4 = "Bool"
 * afun5 = opt(x)
 * afun6 = layout
 * afun7 = lit(x)
 * afun8 = "and"
 * afun9 = attrs(x)
 * afun10 = assoc(x)
 * afun11 = left
 *
 * patternBoolAnd = appl(prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),
 *                      cf(opt(layout)),cf(sort("Bool"))],cf(sort("Bool")),
 *                      attrs([assoc(left)])),
 *                      [<term>,<term>,lit("and"),<term>,<term>])
 *
 */

static ATermList _test_dict = NULL;

#define _test_dict_LEN 239

static char _test_dict_baf[_test_dict_LEN] = {
0x00,0x8B,0xAF,0x83,0x00,0x11,0x33,0x03,0x3C,0x5F,0x3E,0x01,0x00,0x01,0x01,0x03,
0x05,0x5B,0x5F,0x2C,0x5F,0x5D,0x02,0x00,0x1A,0x0E,0x01,0x00,0x05,0x06,0x07,0x08,
0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x02,0x01,0x02,0x02,0x5B,0x5D,0x00,0x00,
...
};


void init_test_dict()
{
  ATermList afuns, terms;

  _test_dict = (ATermList)ATreadFromBinaryString(_test_dict_baf, _test_dict_LEN);

  ATprotect((ATerm *)&_test_dict);

  afuns = (ATermList)ATelementAt(_test_dict, 0);

  afun0 = ATgetAFun((ATermAppl)ATgetFirst(afuns));
  afuns = ATgetNext(afuns);
  afun1 = ATgetAFun((ATermAppl)ATgetFirst(afuns));
...

  terms = (ATermList)ATelementAt(_test_dict, 1);

  patternBoolAnd = ATgetFirst(terms);
  terms = ATgetNext(terms);
}
```

Bibliography

[1] J.R. Allen. *Anatomy of LISP*. McGraw-Hill, 1978.

[2] A.W. Appel and M.J.R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[3] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[4] Information Technology – Abstract Syntax Notation One (ASN.1): Encoding Rules – Packed Encoding Rules (PER). Technical report, International Telecommunication Union, 1995. ITU-T Recommendation X.691.

[5] U. Aßmann. *Invasive Software Composition*. Springer Verlag, 2003.

[6] C. Atkinson and D. Muthig. Enhancing component reusability through product line technology. In C. Gacek, editor, *ICSR*, volume 2319 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002.

[7] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge University Press, 1990.

[8] H.C.N. Bakker and J.W.C. Koorn. Building an editor from existing components: an exercise in software re-use. Technical Report P9312, Programming Research Group, University of Amsterdam, 1993.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd ed.)*. Addison-Wesley, 2005.

[10] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[11] J.A. Bergstra and P. Klint. The ToolBus: a component interconnection architecture. Technical Report P9408, University of Amsterdam, Programming Research Group, 1994.

[12] J.A. Bergstra and P. Klint. The discrete time ToolBus. Technical Report P9502, University of Amsterdam, Programming Research Group, 1995.

[13] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88, 1996.

[14] J.A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1996.

[15] J.A. Bergstra and P. Klint. The discrete time ToolBus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[16] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel, J.K. Lenstra, and L.G.L.T. Meertens, editors, *Mathematics & Computer Science II*, volume 4 of *CWI Monograph*. North-Holland, 1986.

[17] J.A. Bergstra, A. Ponse, and J. van Wamel. Process algebra with backtracking. In *REX School/Symposium*, pages 46–91, 1993.

[18] P.A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[19] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer-Verlag, July 2001.

[20] H. Boehm. Space efficient conservative garbage collection. *PLDI*, pages 197–206, 1993.

[21] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience (SPE)*, 18(9):807–820, 1988.

[22] A. Bolour. Notes on the eclipse plug-in architecture. July 2003.

[23] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

[24] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[25] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and A.E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[26] M.G.J. van den Brand and C. Groza. The algebraic specification of annotated abstract syntax trees. Technical Report P9414, University of Amsterdam, Programming Research Group, 1994.

[27] M.G.J. van den Brand, J. Heering, and P. Klint. Renovation of the ASF+SDF meta-environment - current state of affairs. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer-Verlag, 1997.

[28] M.G.J. van den Brand, J. Iversen, and P.D. Mosses. An Action Environment. In G. Hedin and E. van Wyk, editors, *Language Descriptions, Tools and Applications (LDTA 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2004.

[29] M.G.J. van den Brand, J. Iversen, and P.D. Mosses. An Action Environment. *Science of Computer Programming*, 61:245–264, 2006.

[30] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.

[31] M.G.J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: its all about sharing! *Information and Software Technology*, 2006. To appear.

[32] M.G.J. van den Brand, P. Klint, and P.A. Olivier. ATerms: Exchanging data between heterogeneous tools for CASL. Note T-3, in [47], 1998.

[33] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 198–213, 1999.

[34] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 139–161. Elsevier Science, 1998.

[35] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P.A. Olivier. Design and Implementation of a New Asf+Sdf Meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, November 1997. Springer-Verlag.

[36] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free. In *Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[37] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

[38] M.G.J. van den Brand and C. Ringeissen. Asf+sdf parsing tools applied to elan. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[39] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR parsers. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[40] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[41] M.G.J. van den Brand Brand, H.A. de Jong, P. Klint, and A.T. Kooiker. A language development environment for eclipse. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 55–59. ACM Press, 2003.

[42] XML/Java Data Binding and Breeze XML Binder. Technical report, The Breeze Factor, 2002. available via `http`[1].

[43] E. Brinksma. *On the Design of Extended LOTOS–A Specification Language for Open Distributed Systems*. PhD thesis, University Twente, 1988.

[44] Business Process Management Initiative. *Business Process Modeling Language*, November 2002. `http://www.bpmi.org/bpmi-downloads/BPML1.0.zip`.

[45] D. Chappell. *Understanding ActiveX(TM) and OLE*. MicroSoft Press, 1996.

[46] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[47] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents available by `http`[2], 1998.

[48] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [47], 1998.

[49] *The Common Object Request Broker: Architecture and Specification*. Object Managament Group (OMG), 1999.

---

[1] `http://www.breezefactor.com/whitepapers.html`
[2] `http://www.brics.dk/Projects/CoFI/`

[50] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[51] D. Dams and J.F. Groote. Specification and Implementation of Components of a muCRL toolbox. Logic Group Preprint Series 152, Utrecht University, Dept. of Philosoph, 1995.

[52] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[53] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[54] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[55] B. Diertens. New features in PSF I – Interrupts, Disrupts, and Priorities. Technical Report P9417, Programming Research Group, University of Amsterdam, 1994.

[56] B. Diertens. Simulation and animation of process algebra specifications. Technical Report P9713, Programming Research Group, University of Amsterdam, 1997.

[57] E.N.M. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[58] The Enhydra Project. The Zeus Java-to-XML Data Binding tool. Available via `http`[3], 2002.

[59] The ExoLab Group. Castor - a mapping framework between Java objects, XML documents, SQL & OQL databases and LDAP directories. Available via `http`[4], 2002.

[60] The firefox web browser. `http://www.mozilla.org/products/firefox`.

[61] W. Fokkink, J.F.Th. Kamperman, and H.R. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, 1998.

[62] Eclipse Foundation. the eclipse tool platform. The Eclipse Foundation website `http://www.eclipse.org`, 2004.

---

[3]`http://zeus.enhydra.org`
[4]`http://castor.exolab.org`

[63] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[65] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[66] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[67] 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):121–130, 1992.

[68] J.F. Groote and B. Lisser. Tutorial and reference guide for the μCRL toolset version 1.0. Technical report, CWI, Amsterdam, 1999.

[69] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.

[70] J. Grosch. Ast – a generator for abstract syntax trees. Technical Report 15, GMD Karlsruhe, 1992.

[71] D.R. Hanson. Early Experience with ASDL in lcc. *Software—Practice and Experience*, 29(3):417–435, 1999.

[72] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[73] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, March 2000.

[74] Java architecture for XML binding. Technical report, SUN Microsystems, 2002. available at: `http://java.sun.com/xml/jaxb`.

[75] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[76] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[77] H.A. de Jong and P. Klint. Toolbus: the next generation. In F. S. de Boer, M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 220–241. Springer-Verlag, 2003.

[78] H.A. de Jong and A.T. Kooiker. My favorite editor anywhere. In Nicolas Guelfi, editor, *Rapid Integration of Software Engineering Techniques (RISE2004)*, volume 3475 of *Lecture Notes in Computer Science*, pages 122–131. Springer-Verlag, May 2005.

[79] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming (JLAP)*, 59:35–61, April-May 2004. Issues 1–2.

[80] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M.G.J. van den Brand and D.Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[81] M. de Jonge and J. Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, Erfurt, Germany, 2001. Springer-Verlag.

[82] J.F.Th. Kamperman. GEL, a graph exchange language. Technical Report CS-R9440, CWI, Amsterdam, 1994.

[83] M. Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of the ACM SIGSOFT sixth International Symposium on Foundations of Software Engineering*, pages 131–142, 1998.

[84] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[85] P. Klint. *A Study in String Processing Languages*, volume 205 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[86] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[87] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proceedings of the International Conference on Software Maintenance*, pages 179–189. IEEE Computer Society, 2003.

[88] D. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[89] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[90] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[91] D.A. Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, 1987.

[92] R. Lämmel and J. Visser. A strafunski application letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[93] B. Lisser and J.J. van Wamel. Specification of components in a proposition solver. Technical Report SEN-R9720, Centrum voor Wiskunde en Informatica (CWI), 1997.

[94] S.P. Luttik. Description and formal specification of the link layer protocol (SEN-R9706). Technical report, CWI, Amsterdam, 1999.

[95] The manifold project. http://www.cwi.nl/projects/manifold/manifold.html.

[96] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[97] M.D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, pages 79–87, 1968.

[98] B. Moolenaar. Vim is a highly configurable text editor built to enable efficient text editing. Vim 6.3 is available for download from `http://www.vim.org`, 2004.

[99] P.E. Moreau and O. Zendra. Gc^2: a generational conservative garbage collector for the ATerm library. *Journal of Logic and Algebraic Programming (JLAP)*, 59(1-2):5–34, 2004.

[100] P.D. Mosses. System demonstration: Action semantics tools. In M.G.J. van den Brand and R. Lämmel, editors, *Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65.3 of *Electronic Notes in Theoretical Computer Science*, 2002.

[101] The mozilla all-in-one internet application suite. `http://www.mozilla.org/products/firefox`.

[102] G.J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold Company, 1978.

[103] Object Management Group. *CORBA IIOP Specification*, 2003. `www.omg.org/technology/documents/formal/corba_iiop.htm`.

[104] P.A. Olivier. Embedded system simulation: testdriving the ToolBus. Technical Report P9601, University of Amsterdam, Programming Research Group, 1996.

[105] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, University of Amsterdam, 2000.

[106] D. Olshansky. The java plug-in framework. `http://jpf.sourceforge.net`, 2004.

[107] OMG. The common object request broker: Architecture and specification, revision 2.0. Technical Report 97-02-25, Object Management Group, 1997. Available at: `http://www.omg.org`.

[108] Windows Media Player. Windows media player. `http://www.microsoft.com/windows/windowsmedia`.

[109] B. Randell. System structures for software fault tolerance. *IEEE Transactions on Software Engineering*, SE–1(3):21–232, June 1975.

[110] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

[111] R.M. Stallman. Emacs the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 147–156, 1981.

[112] Sun MicroSystems Inc. *Java Remote Method Specification*, 2003. `http://java.sun.com/j2se/1.4/docs/guide/rmi`.

[113] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[114] The Nullsoft Team. Winamp media player. `http://www.winamp.com`.

[115] M. Terashima and Y. Kanada. HLisp—its concept, implementation and applications. *Journal of Information Processing*, 13(3):265–275, 1990.

[116] The thunderbird email client. `http://www.mozilla.org/products/thunderbird`.

[117] N.P. Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4–5):219–254, 2004.

[118] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[119] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[120] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

[121] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

[122] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. Control and data transfer in the distributed editor of the ASF+SDF meta-environment. Technical Report P9415, University of Amsterdam, Programming Research Group, 1994.

[123] W3C: World Wide Web Consortium. *Web Services Description Language*, March 2001. `http://www.w3.org/TR/wsdl`.

[124] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[125] IBM Websphere MQ. http://www-306.ibm.com/software/integration/wmq/.

[126] J.E. White. A high-level framework for network-based resource sharing. `ftp://ftp.rfc-editor.org/in-notes/rfc707.txt`, 1975.

[127] Wikipedia. Pipeline (unix). `http://en.wikipedia.org/wiki/Unix_pipe`.

[128] R.P. Wilson, R.S. French, Ch.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, M.W. Hall, M.S. Lamm, and J.L. Hennessy. SUIF: An infrastructure for research on parallilizeing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[129] Extensible markup language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. Available at: `http:www.w3.org/TR/REC-xml`.

[130] A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud, and I. Welch. Using coordinated atomic actions to design dependable distributed object systems. In *OOPSLA'97 Workshop on Dependable Distributed Object Systems*, 1997.

Samenvatting

Hoe bouw je een flexibel software systeem uit reeds bestaande en nieuw te ontwerpen componenten? Hoe wisselen deze componenten onderling relevante gegevens uit? Hoe zorg je ervoor dat deze bundeling van aan elkaar "gelijmde" componenten, ondanks grote onderlinge verschillen (*heterogeniteit*) in bijv. de gebruikte programmeertaal, leidt tot de gewenste applicatie? Hoe zorg je ervoor dat die applicatie dan ook nog zo flexibel is dat er later nog uitbreidingen en veranderingen aan kunnen worden aangebracht? Dit zijn enkele vragen waar dit proefschrift een antwoord op probeert te geven. Hieronder volgt een samenvatting van de hoofdstukken uit dit proefschrift. Elk van deze hoofdstukken speelt een rol in het bouwen van een flexibel software systeem dat bestaat uit heterogene componenten.

Informatie uitwisseling   Zodra een software systeem uit twee of meer componenten bestaat, moet ook worden nagedacht hoe deze componenten onderling informatie uitwisselen. Om niet voor elke mogelijke uitwisseling opnieuw te hoeven verzinnen hoe die tot stand moet komen, is het wenselijk hiervoor een algemene oplossing te ontwerpen. In dit proefschrift wordt het ATerm data type beschreven, waarmee op een generieke manier informatie tussen componenten kan worden uitgewisseld. Een belangrijke eigenschap van het ATerm data type is dat ze op een zeer compacte manier kunnen worden gerepresenteerd, zodat zowel bewerkingen op de data, als overdracht van data tussen componenten met minimale inspanning gerealiseerd kan worden.

Automatische vertalingen   Wanneer applicatie-specifieke informatie in de ene component overgestuurd moet worden naar een andere, gebruikmakend van een generiek uitwisselingsformaat zoals een ATerm, zijn enkele vertaalslagen noodzakelijk. Specifieke informatie in de ene component moet worden vertaald naar het algemeen toegankelijke formaat om het op een generieke manier te kunnen oversturen. De ontvangende component wil vervolgens weer op zijn eigen specifieke mannier met de ontvangen informatie omgaan. De programmacode die deze noodzakelijke vertaalslagen bewerkstelligt, kan door middel van codegeneratie-technieken automatisch worden gebouwd. Dit proefschrift laat zien hoe, door op een formele manier te beschrijven welke soort informatie de componenten moeten kunnen uitwisselen, een programmabibliotheek kan worden gegenereerd in verschillende programmeertalen. Deze bibliotheken worden vervolgens gebruikt in de diverse componenten om in de brontaal van die component de uit te wisselen gegevens op te bouwen, of te inspecteren.

Zenuwstelsel Net zoals een zenuwprikkel in het menselijk lichaam via een stelsel van verbonden zenuwen uiteindelijk de juiste plaats bereikt, moet er ook in een software systeem dat bestaat uit verschillende componenten een soort zenuwstelsel zijn. Dit stelsel, ook wel *middleware*, of coördinatie-architectuur genoemd, is ervoor verantwoordelijk dat de diverse componenten in een applicatie met elkaar kunnen communiceren. In dit proefschrift wordt intensief gebruik gemaakt van de ToolBus, een programmeerbaar "zenuwstelsel". Deze coördinatie-architectuur zorgt er niet alleen voor dat informatie (in de vorm van een ATerm) wordt overgedragen van de ene naar de andere component, maar doordat de ToolBus programmeerbaar is, kan het gedrag van de communicatie gestuurd worden. Zo kan bepaalde communicatie voorrang krijgen boven andere, ondergeschikt geachte communicatie en bepaalde communicatiepatronen kunnen worden verboden. Verder is het in de ToolBus mogelijk om exact aan te geven of er één-op-één communicatie moet plaatsvinden tussen componenten, of dat een bepaalde boodschap juist naar iedereen die erin is geïnteresseerd, moet worden gestuurd. Naast intensief gebruik van de ToolBus in veel van de voorbeelden in dit proefschrift, is ook een discussie opgenomen over mogelijke verbeteringen en uitbreidingen aan dit programmeerbare zenuwstelsel voor software systemen.

Bestaande componenten Bij het bouwen van een software systeem, zou telkens de vraag gesteld kunnen worden: "Bestaat dit niet al?". Vaak is het antwoord op deze vraag "Ja, maar …", gevolgd door een uitleg waarom een bepaalde, reeds bestaande, component *bijna*, maar net niet helemaal voldoet. Om toch een bestaande component te kunnen (her-)gebruiken in een software systeem, is het interessant om te kijken hoe, met minimale inspanning, zo'n component toch aan de coördinatie-architectuur gehangen kan worden en hoe de nodige functionaliteit vervolgens aangeboden kan worden aan de rest van de applicatie. In dit proefschrift is specifiek onderzocht hoe twee bestaande en –in de informaticawereld– bekende tekstverwerkingscomponenten "Vim" en "GNU Emacs" kunnen worden ingezet als tekstverwerker voor een programmeeromgeving. Door de voor deze programmeeromgeving noodzakelijke operaties op tekst te definiëren, zoals "lees dit bestand in", "ga naar regel X" en "kleur kolom X tot en met Y blauw", ontstaat een *interface* waaraan alle tekstverwerkers vervolgens moeten voldoen. Voor elk van deze componenten wordt dan de minimale "componentlijm" geprogrammeerd, die nodig is om de component aan deze interface te laten voldoen. Het resultaat is dat een programmeeromgeving ontstaat, waarin mensen hun eigen welbekende tekstverwerker kunnen (her-)gebruiken. Het doel om niet zelf weer zo'n pakket te ontwikkelen en door hergebruik kostenbesparend een software systeem te bouwen, is dan bereikt.

Uitbreidbaarheid Een boormachine kan tegenwoordig niet alleen boren, er zijn ook opzetstukken voor te krijgen om allerhande schroeven in te draaien. Evenzo is de PC niet af als hij thuis staat, we kunnen er uitbreidingskaarten (via de PCI bus) insteken, apparaten aanhangen (USB of Firewire), of de ventilatoren ervan vervangen. In software zien we ook steeds meer de wens doorschemeren, om na installatie nog van alles te kunnen veranderen aan het systeem. De *plug-ins* zijn dan ook al niet meer weg te denken uit internet browsers en muziek- of videoafspeelprogrammatuur. Waar bij de

boormachine en PC de uitbreidingsmogelijkheden beperkt zijn, door bijv. voorgeschreven maten, of electronische specificaties, is men in de software nog meer zoekende naar de grenzen van uitbreidingen en veranderingen. Dit proefschrift bestudeert een aantal van deze uitbreidingsarchitecturen en beschrijft hoe zo'n architectuur met behulp van de ToolBus op een structurele wijze kan worden opgezet. Waar het in bestaande systemen soms onduidelijk is wat er moet gebeuren wanneer twee plug-ins op hetzelfde punt in het systeem willen ingrijpen, kan in een programmeerbare coördinatie-architectuur precies worden gespecificeerd wat het gewenste gedrag is.

Drempelverlagend   Wil hergebruik van componenten bijdragen aan een goedkopere manier van software ontwikkeling, dan zal er alles aan moeten worden gedaan om de integratie en interactie van componenten zo eenvoudig en flexibel mogelijk te laten zijn. Door drempels, die hergebruik van componenten in de weg staan, te verlagen, draagt dit proefschrift bij aan het succesvol bouwen van flexibele software systemen, opgebouwd uit heterogene componenten.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computing Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01