ABSTRACTION: A NOTION FOR REVERSE ENGINEERING

PhD Thesis

De Montfort University

Xiaodong Liu

September 1999

Abstract

The importance and popularity of software reengineering increase as more and more successful computing systems become legacy systems. However, one prominent problem hinders software engineers from effective and efficient reengineering of legacy systems, that is, the difficulty of comprehension of the original system.

This difficulty is due to constant system evolution and incomplete or obsolete documents which legacy systems tend to have. It is proved that the most or only reliable information on a legacy system is source code itself. However, source code is difficult to understand, especially when in a large amount. Since program design or specification is at a higher abstraction level, which is more concise and easier to understand, successful extraction of semantics-oriented specification from legacy source code will facilitate the comprehension and therefore reengineering of legacy systems greatly.

The thesis first proposes a unified approach for software reengineering based on the characteristics of legacy systems. The approach is based on the construction of a wide spectrum language, known as RWSL, which enjoys a sound formal semantics. The architecture and working flow of the approach are proposed, and the structure of RWSL is defined to provide a spectrum of abstractions of the reengineered system, from source code to specification.

Based on this framework, the thesis then focuses on engaging abstraction technology to extract formal specification from legacy source code. A taxonomy of abstraction is developed to identify diverse kinds of abstractions. Monotonicity and relations between these abstractions are formally described. For practical reverse engineering, a set of abstraction rules are developed to solve how to conduct abstraction. All these rules are formally defined and proved sound. Healthiness obligations are developed as axioms to guarantee correct and sensible abstraction during reverse engineering.

A formal notation is adopted widely to provide a solid unambiguous semantic foun-

dation of the proposed approach. The extracted specification is set to be formal to give the reengineered systems a rigorous description. An automatic tool would benefit from the use of formalism. Due to its distinct advantage for both time critical and non-time systems, Interval Temporal Logic (ITL) is adopted to be the specification layer of RWSL, and to define formal semantics of other layers of RWSL. Furthermore, the abstraction taxonomy and rules, monotonicity and relations between abstractions, and healthiness obligations are all formally defined and proved sound (if applicable) within ITL.

The proposed approach aims at time critical systems with parallelism as well as sequential non-time systems. This is a particular challenging research area because within such a system the functional behaviour and non-functional timing requirement are combined, implicit and can be very difficult to recover.

A prototype tool is developed for three purposes: to test the approach, to speed and to scale up reengineering based on the proposed approach. A number of case studies are used for experiments with the approach and the prototype tool.

Conclusion is drawn based on analysis, which shows that the proposed approach is feasible and promising in its domain. Further research directions are also discussed.

Acknowledgements

I wish to express my most profound thanks to my supervisors Dr. Hongji Yang and Professor Hussein Zedan for their invaluable advice, support and encouragement during my three year study. Without any of these, the work in this thesis would only be impossible. There are so many intensive discussions that impressed me so much.

I also wish to thank Dr. Antonio Cau, for his innovative and valuable contribution to the thesis, together with his excellent technical support of a highly efficient computer environment.

Meanwhile, I would like to thank colleagues at Software Technology Research Laboratory and Department of Computer Science in De Montfort University for their support and feedback, and for providing such a stimulating and friendly working atmosphere. There are too many to list individually. The regular seminars provide us with a good opportunity to communicate, discuss and co-stimulate.

I would also like to thank the Research Office in De Montfort University for their outstanding management.

Finally, I must thank my wife and my parents for all their memorable support and encouragement, which are too precious to forget.

Declaration

I declare that the work described within this thesis was originally taken by me between the dates of registration for the degree of Doctor of Philosophy at De Montfort University, September 1996 to September 1999.

The thesis is written by me and has been produced using LaTeX.

Contents

Chapter 1

Introduction

1.1 Purpose of Research and Overview of Problem

In the early days of computing, software reengineering attracted little attention. Nowadays it has become evident that old architectures severely constrain new designs, which leads to demands for changes to existing software, for instance, fixing errors, adding enhancements and making optimisations. However, the implementation of the changes themselves often creates problems over and above those that are being rectified [10, 148, 149, 168, 164].

The large cost associated with software reengineering is the result of the software having proved difficult to reengineer. Early systems tended to be unstructured and *ad hoc*, which makes it hard to understand their behaviour. System documentation is often incomplete, or out of date. With current methods, it is often difficult to retest or verify a system after a change has been made. Successful software will inevitably evolve, but the process of evolution will lead to degraded structure, e.g., improper extension, and yet greater complexity, e.g., enhancement of functionality.

The above situation leads to an increasing industrial demand to carry out maintenance more efficiently, which triggered the research described in this thesis. Reengineering consists of mainly two parts, reverse engineering and forward engineering,

where reverse engineering is the first step and forward engineering is the follow-on step. It is natural to assume that the forward engineering part of reengineering can be carried out by borrowing an existing suitable software development method which has been well developed. Because it is believed that the main problems associated with the forward engineering part of reengineering are to interface an existing well developed software development method, reengineering research should be focusing on the reverse engineering part, i.e., how to understand an existing system or how to obtain design and/or specification from an existing system.

Due to evolution and obsolete or incomplete documents, the most reliable source of information on a legacy software is the code itself. This means that the extraction of the program design or specification of legacy program code is a vital step where software abstraction is apparently the key technique. Successfully extracted specification can facilitate the software engineer's understanding of the legacy system, both in efficiency and accuracy, because of its conciseness and problem-oriented nature (in contrast with code). The benefit is worth the cost, especially for critical legacy systems.

In this thesis, we are going to discuss how to tackle the abstraction problem in reverse engineering, based on the following observations [110]:

- Most existing research/commercial reverse engineering approaches/tools can basically "restructure" existing code, and these operate at the same level of abstraction. Hence, abstraction methods are desperately needed for reverse engineering.

- Formal methods can provide a solid theoretical foundation for integrating both restructuring and abstraction techniques in building a practical software reverse engineering tool.

- Object-Oriented techniques, which have been recognised as the best way currently available for structuring software systems, can help reengineering in grouping together data and operations performed on them, thereby encapsulating the whole system behind a clean interface, and organising the resulting entities in a hierarchy based on specialisation in functionalities.

- In an interrupt-driven, real-time program with time constraints, the functional behaviour of this program and the non-functional timing requirements are combined, implicit and can be very difficult to recover. Reverse engineering such a program is a particular challenging research area.

The terms used in the chapter, such as software maintenance, reverse engineering, abstraction, etc., will be defined in the following chapters.

1.2 Scope of the Thesis and Original Contribution

In this thesis, a unified approach for software reengineering is proposed. The approach is based on the construction of a wide spectrum language, known as RWSL, which enjoys a sound formal semantics. The thesis concentrates on engaging abstraction technology to extract formal specification from legacy source code. The scope of research includes:

- The architectural design of the unified software reengineering approach: the architecture and working flow of the approach are proposed, and the structure of RWSL, and its formal and informal syntax and semantics, are defined.

- The formalisation of the notion of "abstraction": a taxonomy of abstraction is developed, including definitions and relations of several kinds of abstractions. And rules to conduct abstractions in the reverse engineering part of reengineering are then developed aiming at extracting formal specification from legacy code.

- Implementation of a prototype tool and experimentation with case studies: a system is developed to demonstrate the success of the proposed approach. Another purpose of the prototype tool is to speed up and scale up the proposed approach. A number of case studies are used for experiments with the approach and the prototype system.

The original contribution of the thesis lies in three aspects:

- Abstraction.
  - Levels of abstraction. RWSL provides a spectrum of abstractions of the reengineered system, from concrete code to specification. These abstractions are integrated and cooperate in a uniform manner. All the layers in RWSL have formal syntax and semantics, which gives the target system unambiguous descriptions at various abstraction levels.
  - Abstraction taxonomy and rules. In the proposed approach, reverse engineering is carried out by extracting more concise system descriptions from a less abstract level, e.g., code. This involves crossing levels of abstraction. To achieve this, a taxonomy of abstraction is developed to answer "what abstraction is", including definitions and relations of diverse abstractions. Then, abstraction rules are developed to solve how to conduct abstractions. All the abstraction rules are defined formally, which assures precise and rigorous semantics. With these rules a satisfactory specification can be extracted from source code.
- Real-time domain. At present, reverse engineering technology is mainly limited to sequential and non-time systems no matter whether it adopts formal techniques or ad hoc techniques [110]. The proposed approach treats time-critical systems with parallelism as its specific application domain, together with normal sequential non-timed systems.
- Object orientation. The proposed approach supports the reverse engineering of object-oriented systems, i.e., with the abstraction rules, an object-oriented program can be abstracted into a formal specification.

The literature survey in Chapter 2 and 3 shows that there is not any reverse engineering approach or tool dealing with the extraction of specification from code in real-time or object-oriented domain through formally defined abstraction rules.

1.3 Criteria for Success

The following criteria are given to judge the success of the research described in this thesis:

- For a heavily modified legacy system which has never been developed in a well structured or object-oriented method, how viable is it to extract a specification from its source code with abstraction technology?

- Is the extracted specification consistent to the original design? Is it reliable to perform redesign or re-specification on the base of the extracted specification?

- Is the extracted specification unambiguous and easy to understand?

- What kind of legacy systems can the approach deal with? Besides sequential non-time systems, can it tackle more complex and emergent-in-need but rarely addressed systems, such as parallel and time critical systems?

- Crossing levels of abstraction involves both semantics change/selection and transformation in representation. How does the proposed approach solve this problem? Is the taxonomy of abstraction comprehensive enough and are the abstraction rules reliable?

- Is the approach feasible for realisation? For example, is it possible to build a practical tool based on the approach?

- Is the approach capable for industrial-scaled systems?

1.4 Thesis Structure

The thesis is organised as follows:

- Chapter 1 gives the background, motivation, scope and original contribution of the thesis.

- Chapter 2 provides an overview of the current state of software reengineering, formal methods, and in particular, their intersection, that is, software reengineering adopting formal technology.

- Chapter 3 investigates the existing related work, especially those involving specification recovery, design recovery and usage of formal techniques. Conclusion is drawn based on the investigation.

- Chapter 4 first discusses the characteristics of legacy systems, and then proposes an integral framework together with a relevant approach for reengineering based on a wide spectrum language which enjoys a formal semantics.

- Chapter 5 explores the proposed wide spectrum language in detail, including syntax, formal and informal semantics of each layer.

- Chapter 6 is the gist of the thesis. A taxonomy of abstraction is developed, including definitions of several abstractions, their relations and monotonicity. Abstraction rules to conduct specification extraction from legacy source code are then developed. Demonstrative examples are given.

- Chapter 7 is about realisation of the proposed approach by building a tool, namely, Reengineering Assistant. The chapter covers the tool's general system architecture, internal database, inference procedures and user interface.

- Chapter 8 deals with case studies, which include various legacy systems, from sequential non-time system to real-time systems with parallelism and communication.

- Chapter 9 discusses the proposed approach and the supporting tool according to a set of criteria. Conclusion is drawn based on this discussion, and prospective further work is also discussed.

- Appendix A gives the proof of soundness of abstraction rules, monotonicity of abstractions and relations between different kinds of abstractions.

- Appendix B gives the results of case studies, including legacy code, RWSL code and extracted specifications.

Chapter 2

Software Reengineering with Formal Technology

2.1 Introduction

Any computing system, both hardware and software systems, will inevitable grow in scale and functionality. Because of this complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause catastrophic loss of money, time, or even human life. Large systems are so complex that it is impossible for a single individual to build and maintain all aspects of its design. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity [45, 11, 10, 98]. One way of achieving this goal is using *formal methods*, which are mathematically-based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not guarantee correctness, however, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected [154].

The maintenance of large-scale computing systems is a crucial aspect of software lifecycle. This is due to the fact that systems are continually evolving. Their evolution is mainly due to three factors: **a)** change of original requirement, i.e. either increas-

ing or decreasing functionality; **b)** adapting on different hardware platforms; and/or **c)** improving its efficiency.

As a combination of reverse engineering and forward engineering, software reengineering technology is a practical solution for the above problem of evolution of existing computing systems. Dynamic change management of software systems has been largely performed by using *ad hoc* techniques which are normally rather expensive and in some cases, impossible (if the designer has not documented or left the company). There are at least two advantages of using formal methods as the foundation of software reengineering. Firstly, formal methods can help software engineers acquiring a rigorous and precise description of the system being reengineered, therefore greatly increase the quality of the new system. Secondly, automation is one of the key goals of reengineering. By applying formal methods, it may be possible to make the process of reengineering more automated.

This chapter investigates the current situation of software reengineering and formal methods. It proposes the basic criteria for formal methods applied in software reengineering domain. Among the range of the application areas of software reengineering, the thesis concentrates on real-time systems with parallelism. Based on the criteria, investigation and assessment are made about the existing popular formal methods, especially those potentially suitable for software reengineering. In the last section, conclusion is drawn up based on analysis results and discussions.

2.2 Software Evolution and Maintenance

Software Engineering As one of the most important areas of computer science, software engineering had its origin as a solution to the first "software crisis". According to the IEEE Standards, software engineering is defined as:

> **Software Engineering** *is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software* [92].

Software Evolution It is safe to say that from the day that a large software system goes into service, functional, performance, operator and environmental requirements will undergo changes. Moreover, the delivered software system will contain some latent defects that were not detected during testing. These factors cause software systems inevitably to evolve in scale, environment and functionality, especially those successful enough to survive a long period [18, 12]. Software evolution is regarded as being divided into corrective actions to fix latent defects, adaptive actions to deal with changing environments, and perfective actions to accommodate new requirements. Software evolution is the main cause of software maintenance activities.

Software Maintenance Software maintenance is attracting more and more attention. As a terminology, it is defined as:

> **Software Maintenance** *is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment* [6].

According to the change types that software maintenance is required to meet, maintenance activities can be classified into three categories [153, 23, 24].

The first category is called **corrective maintenance**. There may be a *fault* in the software, so that its behaviour does not conform to its specification. This fault may contradict the specification, or it may demonstrate that the specification is incomplete or inconsistent, so that the user's assumed specification is not sustained. Corrective maintenance involves removing these faults.

Even if a software system is fault-free, the environment in which it operates will often be subject to change, e.g., the upgrade of the computer hardware or moving a system from a mainframe to a PC.

Modifications performed as a result of changes to the external environment are categorised as **adaptive maintenance**, e.g., the manufacturer may introduce new versions of the operating systems, or remove support for existing facilities, and the software may

be ported to a new environment, or to different hardware.

The third category of maintenance is called **perfective maintenance**. This is undertaken as a consequence of a change in user requirements of the software. For example, a payroll suite may need to be altered to reflect new taxation laws; a real-time power station control system may need upgrading to meet safety standards.

The last category, **preventive maintenance** may be undertaken on a system in order to anticipate future problems and make subsequent maintenance easier [19]. For example, a particular part of a large suite may have been found to require sustained corrective maintenance over a period of time. It could be sensible to re-implement this part, using modern software engineering technology, in the expectation that subsequent errors will be reduced.

Software Reengineering The process of *reengineering* computing systems involves three main steps: *restructuring, reverse engineering* and *forward engineering* [21, 24]. In the present survey, we take the following view:

- *Restructuring*. It is the process of creating a logically equivalent system from the given one. This process is performed at the same level of abstraction and does not involve semantic understanding of the original system.

- *Reverse Engineering*. It is the process of analysing a system in order to obtain and identify major system components and their inter-relationships and behaviours. It involves the extraction of higher level specifications from the original system.

- *Forward engineering*. The process of developing a system starting from the requirement specification and moving down towards implementation and deployment.

In essence the reengineering model takes the following form:

$$\textit{Re-engineering} = \textit{Restructuring} + \textit{Reverse engineering} + \textit{Forward engineering}.$$

Abstraction Model of Software Life Cycle Bachman introduced a Reengineering Cycle chart (Figure 2.1) [14], which features both forward and reverse engineering. Reverse engineering, our focus here, begins at the bottom left with the definition of existing applications and raises the applications to successively higher levels of abstraction. At the top, the design objects created by the reverse engineering steps are enhanced and validated to become the revised design objects which may be used in the forward engineering process. At the bottom, a new application system becomes an existing application system at the moment that it goes into production.

To generalise this model, many software systems typically undergo the following stages:

$$\textit{Specification} \rightarrow \textit{Design} \rightarrow \textit{Implementation} \rightarrow \textit{Design} \rightarrow \textit{Specification}$$

This represents a process whereby: before implementing a program, a specification was written first; then a design was derived from the given specification; the program was implemented and then operated for a period of time; when the program needed to be maintained a design or specification (which may be different from the original one) was obtained through reverse engineering (the design or specification can be used for the purposes of maintenance, reengineering, etc.).

A specification specifies "what" a program does; a design states both "what a program does and how it does it", and the program itself implements "how to do the job". Therefore the above process can be represented as follows:

$$\textit{what?} \rightarrow \textit{what/how?} \rightarrow \textit{how?} \rightarrow \textit{what/how?} \rightarrow \textit{what?}$$

A specification, a design and an implementation of a program are usually at different levels of abstraction. To move from one stage (e.g., specification stage) to another stage (e.g., design stage) involves a process of crossing levels of abstraction. Usually a specification is more abstract than its implementation, therefore the above process can be again represented as:

$$\textit{abstract} \rightarrow \textit{less abstract} \rightarrow \textit{concrete} \rightarrow \textit{more abstract} \rightarrow \textit{abstract}$$

This suggests that abstractness of software is an important feature when both forward and reverse engineering are carried out and therefore conducting abstraction is significant for both reverse and forward engineering.

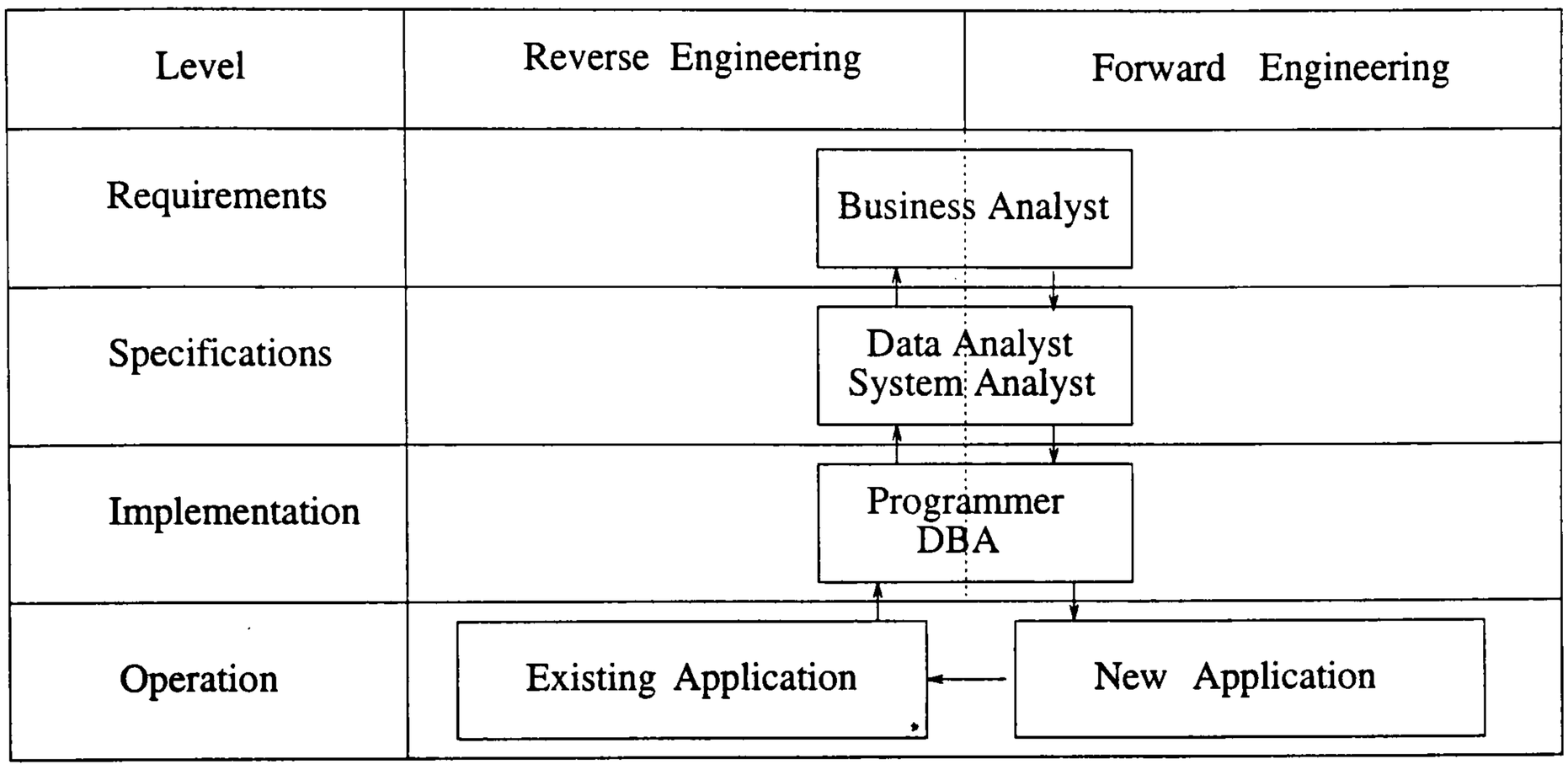

Figure 2.1: Bachman's Reengineering Cycle

2.3 Taxonomy of Software Reengineering

In this section, the following key terms and comparison provide a clear scope and taxonomy of the domain of software reengineering [45, 11, 10, 71]:

Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

Reverse Engineering is the process of analysing a subject system to **(i)** identify the system's components and their interrelationships and **(ii)** create representations of the system in another form or higher level of abstraction.

Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, data flow, data structures, and control flow) intended for a human audience. Redocumentation is the simplest and oldest

form of reverse engineering, and can be considered to be an unintrusive, weak form of restructuring.

Design Recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.

Reverse Design is a synonym to design recovery.

Program Understanding or Program Comprehension is a related term to reverse engineering. Program understanding implies always that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high level descriptions of the design. The science of program understanding includes the cognitive science of human mental processes in program understanding. Program understanding can be achieved in an ad hoc manner and no external representation has to arise. While reverse engineering is the systematic approach to develop an external representation of the subject system, program understanding is comparable with design recovery because both of them start at source code level.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject's system external behaviour, i.e. functionality and semantics.

Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. The process of *reengineering* computing systems involves three main steps: *reverse engineering*, *restructuring* and *forward engineering*.

Reverse Specification is a kind of reverse engineering where a specification is abstracted from the source code or design description. Specification in this context means an abstract description of what the software does. In forward engineering, the specification tells us what the software has to do. But this information is not included in the

source code. Only in rare cases, it can be recovered from comments in the source code and from the people involved in the original forward engineering process.

Re-code is changes to implementation characteristics. Language translation and control flow restructuring are source code level changes. Other possible changes include conforming to coding standards, improving source code readability, renaming program items, etc.

Re-design is changes to design characteristics. Possible changes include restructuring a design architecture, altering a system's data model as incorporated in data structures, or in a database, improvements to an algorithm, etc.

Re-specify is changes to requirements characteristics. This type of change can refer to changing only the form of existing requirements. For example, taking informal requirements expressed in English and generating a formal specification expressed in a formal language such as Z. This type of change can also refer to changing system requirements, such as the addition of new requirements, or the deletion or alteration of existing requirements.

Figure 2.2 presents a general model of reverse engineering, and Figure 2.3 presents a general model of reengineering.

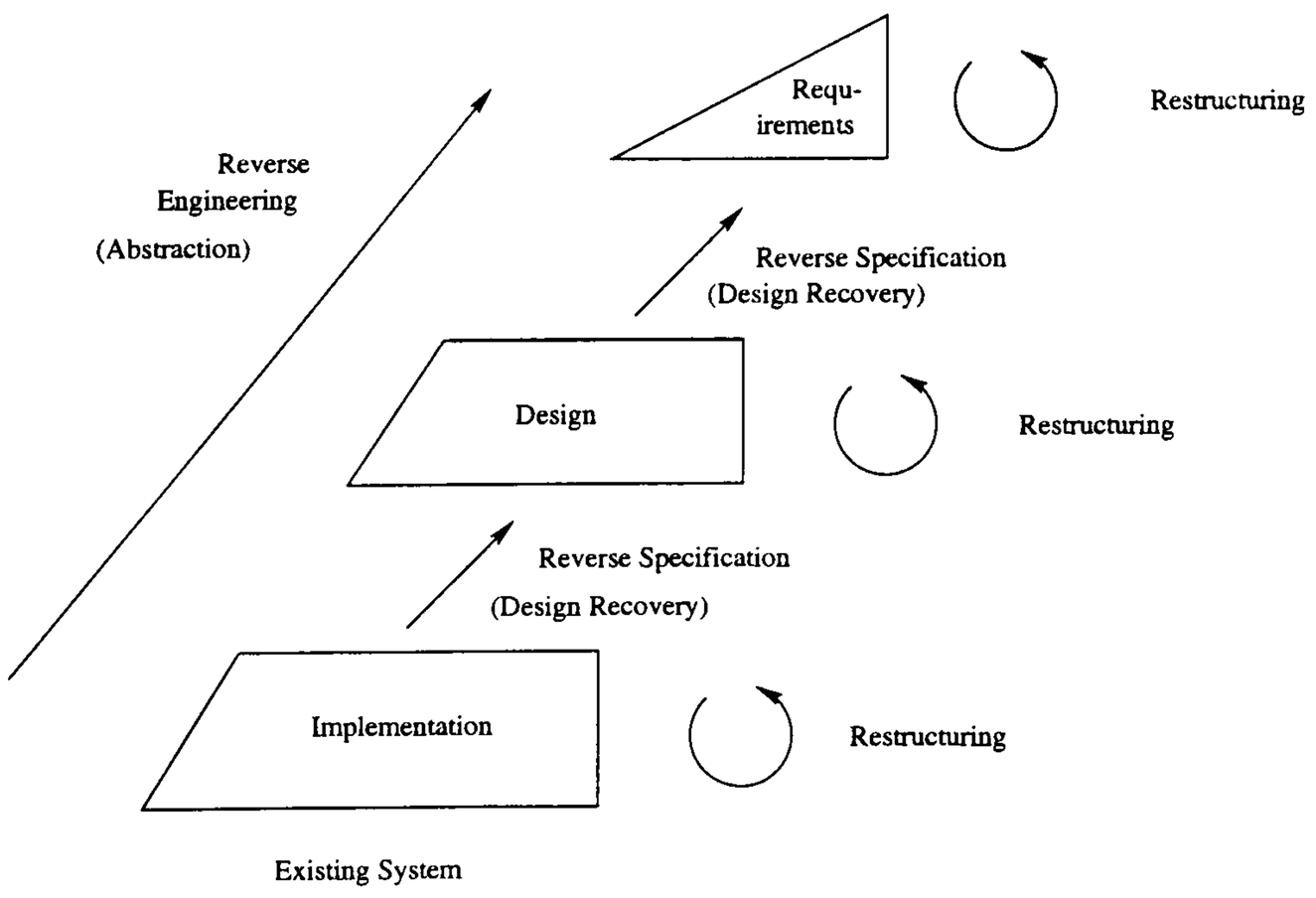

Figure 2.2: General Model for Reverse Engineering.

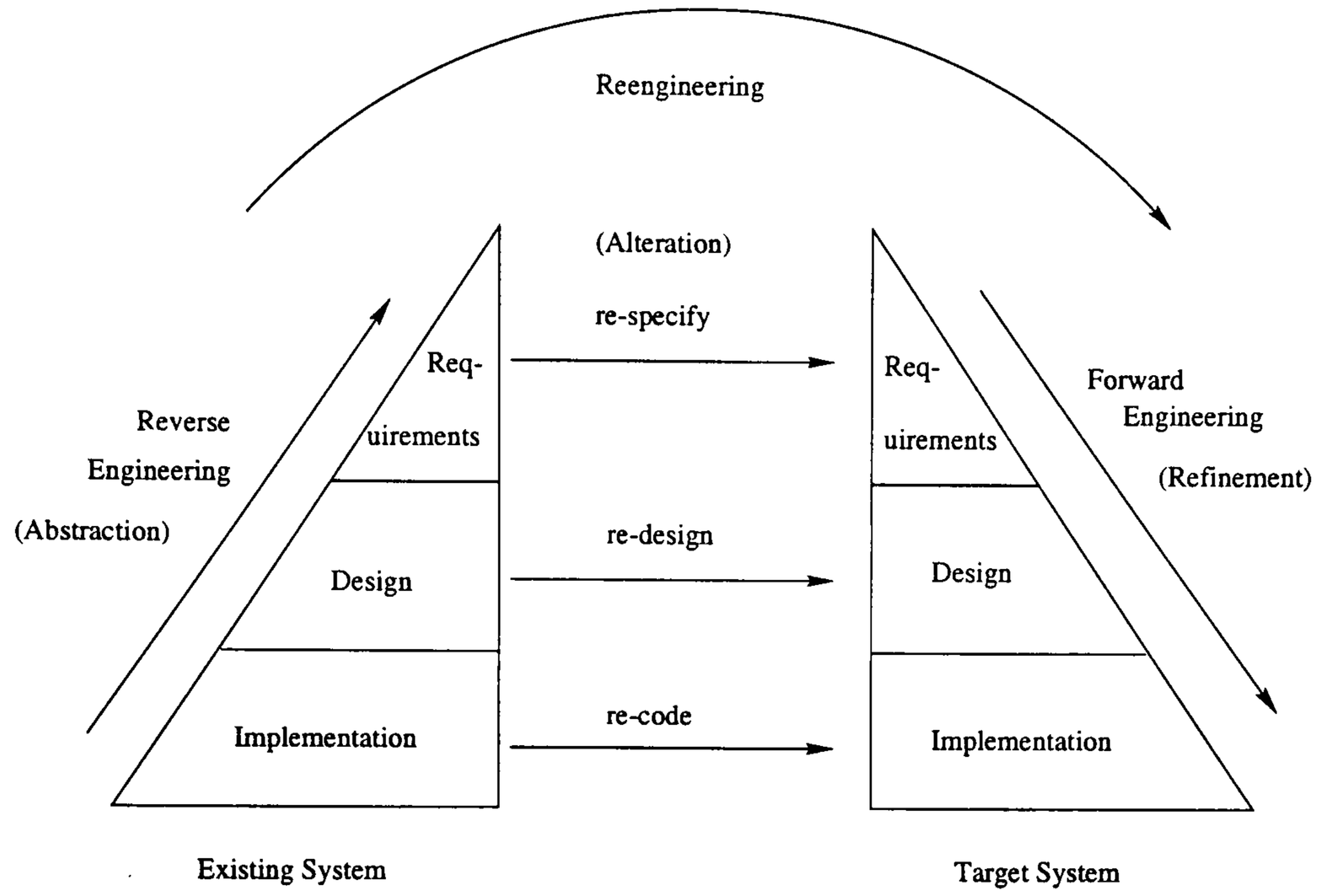

Figure 2.3: General Model for Software Reengineering.

2.4 Possible Research Issues

Compared with forward engineering, reverse engineering has been poorly addressed for many years. The approaches and tools of forward engineering are far more well studied. In this section, possible research issues of reverse engineering are discussed .

Cognitive Processes in Human Program Understanding This research issue focuses on how a human reader can understand unknown source code. The study of human cognitive processes can show where we can support human understanding effectively. It may, but need not, be a model of how automated understanding can work [150, 139].

Intermediate Representation of Source Code In large programs, it is more efficient to preprocess the source code and store the program information in an intermediate representation that allows fast queries instead of querying the source code directly [51]. There are two main topics in this research direction:

- **Use of data & knowledge bases** to store and retrieve source code information. Relevant artificial intelligence technologies can be used to facilitate and optimise the process of reverse engineering [9, 36, 57, 127, 132, 119, 66].

- **Using graphs** to represent source code information [48, 112, 99].

Reverse Specification Reverse specification is intended to extract a description of what the examined system does. The description is made in terms of application domains [149, 31, 45, 51].

On one hand, this process must be bottom-up, since the only reliable description of the behaviour of software is its source code. To support this bottom-up process, methods and tools are useful which generate abstract models from the source code, e.g. the formal description model of any other suitable formal languages. The technique of software animation can be applied to visualise program behaviour or to animate the generated models, since the underlying description, namely the source code, is operational.

On the other hand, the result of reverse specification, i.e. the description model derived from the source code, should be top-down structured. This is to comply with the basic cognitive rules for a human to recognise complicated systems.

The possible research issues in reverse specification includes [51, 91, 7, 14, 36, 45, 48, 55, 149] :

- Domain Analysis, Domain Models.

- Description Model Generating, Software Animation. Software can be animated to the maintainer to help him understand the program. This research can be classified into the following sub-issues:

  - Visualisation of Parallel and Distributed Programs.
  - Visualisation for Program Understanding and Debugging.
  - Visualisation for Object-Oriented Programs.

  - Algorithm Animation.
  - Information Visualisation and Visualisation of Large Systems.

- Requirement Traceability. Software maintainers often have to trace requirements in old code. In other words, they need to answer the question: "In which parts of this program is functionality X implemented?".

Reverse Design, Design Recovery The objective of reverse design is to get a design description out of the source code, i.e. to abstract from coding details [50, 52, 53]. There are two strategies to achieve this [81, 91, 159, 147, 104, 146, 87, 1]:

- Tools present the source code in such a way that a maintainer can make the abstraction. Since the computer only retrieves information that is entirely included in source code this strategy is called **fundamental.**

- Tools make the abstraction on their own. They are analysing the source code in using information from a knowledge base. This strategy is called **knowledge-based.**

Here, we make out the most popular approaches in reverse design.

- Code Views. Code views are representations of source code which cover the same information as the code or part of it but in a manner that accelerate the comprehension process. Examples are program slices, call graphs, data flow, definition-use-graphs, or control dependencies.

- Reformatting and Markup Languages. Reformatting is the functional equivalent transformation of source code which changes only the structure to improve readability. Markup languages are languages for annotations of source code to simply improve the source code's appearance with the means of bold-faced key words, slanted comments, etc.

- Hypertext. Hypertext methods and tools can be used to help browsing the source code.

- Source Code Analysis and Transformation Rules. This is the most important method in reverse design. Concrete transformation rules and inference algorithms are defined according to concrete source code language. By these rules and certain form of knowledge & data bases, source code can be transformed into a higher level of abstraction [112, 5, 42, 44].

- Data Centred Program Understanding. Instead of focusing on the control structure of a program, such as call graphs, control flow graphs and paths, data centred program understanding focuses on data and data relationships [95, 166].

- Program Slicing. A program slice is a fragment of a program in which some statements are omitted that are not necessary to understand a certain property of the program [71, 28].

2.5 Current State of Formal Methods in Reengineering

The debate about the use and relevance of formal methods in the development of computing systems has always attracted a considerable attention and is continually doing so. One school of thought (the protagonists) claims that formal techniques offer a complete solution to the problems of system development. Another school claims that formal methods have little, or no, use in the development process (at least due to the cost involved). There is a third view point, that we share, which states that formal methods are *both over-sold and under-used.*

Nonetheless, whatever school of thought one prescribes to, it is important to realise that as the complexity of building computing systems is continually growing, a disciplined, systematic and rigorous methodology is essential for attaining a "reasonable" level of dependability and trust in these systems. The need for such a methodology increases as "fatal" accidents are attributable to software errors.

In response to this, an intense research activity has developed resulting in the production of formal development techniques together with their associated verification

tools that have been successfully applied in forward engineering such systems. For example, assertional methods, temporal logic, process algebra and automata, have all been used with some degree of success.

In the area of reverse engineering, formal methods have also been put forward as a means to

1. formally specify and verify existing systems in particular those already operating in safety-critical applications;

2. introduce new functionalities and/or

3. take advantage of the improvement in systems design techniques.

We attempt to review a large class of formal methods that have been suggested in the reengineering process of computing systems. We shall also discuss some of their benefits and limitations. But first, it is necessary to lay some terminological groundwork and to consider current practices.

The term *formal methods* is used to refer to methods with sound basis in mathematics. These should be distinguished from *structured methods* which are well defined but do not have sound mathematical basis to describe system functionalities [67]. Formal methods allows system functionalities to be precisely specified whilst structured methods permit the precise specification of systems structure. However, recently, there have been substantial research activities to

- integrate formal and structured methods, for example the formal specification language Z [3, 152] has been integrated with the structured method known as SSADM and

- extend some formal methods allowing the treatment of non-functional requirements such as timing and probability [39, 124, 125, 38, 80, 144].

We take the view that a formal method should consist of some essential components: a semantic model, a specification language (notation), a verification system/refinement calculus, development guidelines and supporting tools:

1. The *semantic model* is a sound mathematical/logical structure within which all terms, formulas and rules used have a precise meaning. The semantic model should reflect the underlying computational model of the intended application.

2. *The specification language* is a set of notations which are used to describe the intended behaviour of the system. This language must have a proper semantics within the semantic model.

3. *Verification system/refinement calculi* are sound rules that allow the verification of properties and/or the refinement of specifications.

4. *Development Guidelines* are steps showing the use of the method.

5. *Supporting tools* involve proof assistant, syntax and type checker, animator, and prototyper.

Formal methods can be applied in two different ways.

1. The production of specifications which are then the basis for a conventional system development. In this case, specifications are used as a precise documentation medium which has the advantages of manipulability, abstraction and conciseness. Consistency checks and automatic generation of prototypes could be performed at this stage with the aid of the associated supporting tools.

2. The production of formal specification, as above, can then be used as a basis against which the correctness of the system is verified or as a basis to derive the verified system through correctness preserving refinement rules. This will give the developed system a degree of certainty and trustworthiness.

2.6 Classification of Formal Methods

Formal methods can be classified into the following five classes or types, i.e., *Model-based, Logic-based, Algebraic, Process Algebra* and *Net-based (Graphical)* methods.

In the following subsections we will briefly discuss each of these approaches.

2.6.1 Model-based Approach

General A system is modelled by explicitly giving definition of states and operations that transform the system from a state to another. In this approach, there is no explicit representation of concurrency. Non-functional requirements (such as temporal requirement) can be expressed in some cases.

Examples

- **Z** [3, 152]. With the first version proposed in 1979, the Z notion is based on predicate calculus and Zermelo Fraenkel set theory. A Z specification is written in terms of "schemas", each of which contains a signature part which declares items of interest and a predicate part which places a logical constraint on them.

- **VDM** [97, 30, 96]. VDM (the Vienna Development Method) is a formal method for *rigorous* computing system development. It is similar to Z in most aspects, although not as popular as Z. VDM supports model composition and decomposition, which facilitate both the forward and reverse engineering a lot.

  Although the semantics and proofs in predicate calculus are complete and rather complete in set theory, the functions, operations, compositions and decompositions in VDM makes its semantic and proof system much more complicated to be "accomplished". Therefore, similar problems happen with Z and VDM: a complete formal semantics does not exist yet, and as the consequence, the automated support tools, such automated prover, do not exist yet. Moreover, lacking of formal semantics will also limit the potentials for automation in the reengineering approach which adopts Z or VDM as its formal foundation.

  Time is not a part of VDM notation. When trying to apply VDM to real-time domain, novel features have to be added to VDM. VDM also keeps developing:

VDM^{++}, as a new version of VDM integrated with object-oriented idea, is a rather mature product now.

- **B-Method** [103, 102, 162]. The B-method uses the Abstract Machine Notation to support the description of the target systems. The most eminent success of B method is that it already has a strong and quite mature tool B Toolkit, to support and automate the development of application systems. The B-Method is "complete" in the sense that it provides abstract machine specifications and their proofs, refinements and their proofs, and compositions and their proofs. The development method of B matches the typical top-down forward engineering well. A complete development may be performed and recorded. Changes may be accommodated using the replay tools. Refinement, implementation and composition steps have precise notions of correctness and mechanical generation of proof obligations. By animator, test may be performed. The final implementation step may be mechanised for common languages (e.g. C and Ada) and for some specification constructs.

  In B-Method, no guidance is provided regarding (i) design decisions or their recording, (ii) testing or inspection methodology, (iii) presentation of specifications. B toolkit is still evolving, not 'very' mature now. B method has no time feature. Novel feature has to be added when using B for real-time systems. The main users of B are found in UK.

Sample Description Z is described as a sample here.

- **Syntax and Semantics**. The conceptual basis of Z is typed set theory, and the method is oriented to constructing models. Text and graphical representation are used.

  The basic elements of Z are types, sets, tuples and bindings. There is no universal set to which all elements belong, but a universe of disjoint sets called types

(which contain basic types and composite types). A set in Z is an unordered collection of different elements of the same type, and there is a concept of infinite sets supported in Z. A tuple is an ordered collection of elements that are not necessarily the same type. A binding is a finite mapping from names to elements, not necessarily of the same type.

The main representational form is the "schema" which is a set of bindings depicted in a special "axiomatic box" syntactical form including a signature(or Schema Name) and a property (made up of two parts—the declaration and axiomatic constraints).

The **semantics** of Z is based on a version of Zermelo-Fraenkel set theory that does not include the replacement and choice axioms.

- **System Specification**. Operations can be specified in several ways in Z. One way is through the use of "axiomatic descriptions", which are unnamed schemas that introduce one or more global variables, and constraints on those variables. These specifications are called "loose specifications" by Z practitioners, who stress the use of schemas to specify. A specifier uses these to indicate a function or constant has certain properties without giving it a value.

A Z specification is basically composed of ordered collections of schema definitions and axiomatic descriptions. There are complex scoping and naming rules, but the most important specification structuring mechanism is called "schema inclusion". The name of a defined schema may be referred to in any other schema of axiomatic description after its definition, but entities within that schema may be referred to only if the schema is included in the signature of the following schema or axiomatic description.

Analysis of Z specifications usually means performing consistency and completeness checks, which validate the specification for accuracy and completeness, style, feasibility (sometimes called viability, seeing if a system state exists which

satisfies the constraints specified in the initial condition), and expected properties. This analysis is performed by review by other specifiers who perform a "walk through" much like a code "walk through". Proofs are also used to analyse a Z specification, which is primarily done by hand, as there is no reliable automated prover for Z because a formal semantics does not exist yet.

- **Assessment**. Z is good at identifying errors that result from misconceptions in the model of a system. Z supports designing through the use of constructing models, and Z does support a refinement approach to developing systems. It is good at determining and specifying relationships between different levels of specification and design. There is also the ability to re-use Z schemas, especially those that are generic. Since the principles and stages of refinement approach in forward engineering are correspondent to those of abstraction approach in reverse engineering, Z can be also competent in being a good formal foundation of a reengineering approach.

As mentioned before, although the semantics of Z is based on a version of Zermelo-Fraenkel set theory, it is not complete or sufficient for the whole Z notations when including schemas, tuples, binding, etc. So, a formal semantics of Z does not exist yet. As a consequence, the automated support tools, such automated prover, do not exist yet. Moreover, lacking of formal semantics will also limit the potentials for automation in reengineering which adopts Z as formal foundation.

Time is not a part of Z notation. When trying to apply Z to real-time domain, novel features have to be added to Z. However, because of the rich expressibility of Z, Z has been used in a number of real-time applications, such as timed Z [114].

The main users of Z are found in UK and other European countries. Generally speaking, Z has been applied to a large amount of applications, some of which are rather large-scaled. It is one of the few formal methods that have been proved successful in industrial applications.

In recent years, some forms of improved Z with new technology such as object orientation has been developed, for example, Z^{++} and Object-Z.

2.6.2 Logic-based Approach

General In this approach logics are used to describe system desired properties, including low-level specification, temporal and probabilistic behaviours. The validity of these properties is achieved using the associated axiom system of the used logic. In some cases, a subset of the logic can be executed, for example the Tempura system [125]. The executable specification can then be used for simulation and rapid prototyping purposes.

Logic can be augmented with some concrete programming constructs to obtain what is known as wide-spectrum formalism. The development of systems in this case is achieved by a set of correctness preserving *refinement steps*. Examples of these forms are TAM [144] and the Refinement Calculus [142].

Examples

- **ITL** [39, 124, 125, 38]. ITL (Interval Temporal Logic) has been developed in [39, 128]. This kind of logic is based on intervals of time, thought of as representing finite chunks of system behaviour. An interval may be divided into two contiguous subintervals, thus leading to *chop* operator.

- **Duration Calculus** [40, 41]. Duration Calculus was introduced in [40] as a logic to specify and reason about requirements for real-time systems. It is an extension of Interval Temporal Logic where one can reason about integrated constraints over time-dependent and Boolean valued states without explicit mention of absolute time. Several rather large-scale case studies have shown that Duration Calculus provides a high level of abstraction for both expressing and reasoning about specifications.

- **Hoare Logic** [83, 84, 85]. Hoare Logic has a long history; it may be viewed as an extension of First-order Predicate Calculus [59] that includes inference rules for reasoning about programming language constructs.

  Hoare Logic provides a means of demonstrating that a program is consistent with its specification. Hoare Logic is not capable of specifying a system at high levels, however, it has distinct advantages in the low level specifications. These two features make Hoare Logic a suitable means in the first stage of reverse engineering, i.e., from source code program to an abstraction at very low level. Some research has been done in this area, such as the development of the reverse engineering tool AutoSpec [43, 73].

  There is no real-time feature in Hoare Logic. Some extension can be added to make Hoare Logic more suitable for real-time domain. A Real-time Hoare Logic has been proposed [88].

  Hoare Logic is one of the mathematical pillars for program verification and formal methods. Hoare Logic and its variants are used in numerous formal methods tools.

- **WP-Calculus** [58, 59]. Weakest Precondition Calculus was first proposed by E. W. Dijkstra in 1976. A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. By using the semantics of predicate logic and other suitable formal logics, WP-Calculus has been proven to be formalism suitable for reverse engineering of source code, especially at the low abstraction levels.

- **Modal Logic** [117, 46]. Modal logic is the study of context-dependent properties such as necessity and possibility. In modal logic, the meaning of expressions depends on an implicit context, abstracted away from the object language. Temporal logic can be regarded as an instance of modal logic where the collection of contexts models a collection of moments in time. A modal logic is equipped with

modal operators through which elements from different contexts can be combined. Two most popular modal operators are the necessity operator □ and the possibility operator ◇. There are several approaches to the semantics of modal logic, such as 'neighbourhood' semantics. Until now, there is no application of modal logic in software reverse engineering area.

- **Temporal Logic** [138]. Temporal logic has its origins in philosophy, where it was used to analyse the structure or topology of time. In recent years, it has found a good value in real-time application.

  In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special *temporal* logic.

  However, philosophers found it useful to introduce special temporal operators, such as □ (henceforth) and ◇ (eventually), for the analysis of temporal connectives in languages. The new formalism was soon seen as a potentially valuable tool for analysing the topology of time. Various types of semantics can be given to the temporal operators depending on whether time is linear, parallel or branching. Another aspect is whether time is discrete or continuous [115].

Temporal logic is *state-based.* A structure of states is the key concept that makes temporal logic suitable for system specification. Mainly, the types of temporal semantics include *interval semantics*, *point semantics*, *linear semantics*, *branching semantics* and *partial order semantics* [115].

The various temporal logics can be used to reason about *qualitative* temporal properties. Safety properties that can be specified include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness. Fairness properties include scheduling a given process infinitely often, or requiring that a continuously enabled transition ultimately fire.

Various proof systems and decision procedures for finite state systems can be used to check the correctness of a program or system.

In real-time temporal logics, *quantitative* properties can also be expressed such as periodicity, real-time response (deadline), and delays. Early approaches to real-time temporal logics were reported in [131, 25]. Since then, real-time logics have been explored in great detail.

- **TAM** [145, 144, 143]. TAM (Temporal Agent Model) aims to be a realistic software development method for real-time systems. It has striven to support a computational model which is amenable both to analysis by run-time execution environment software, and to efficient implementation. In doing so, TAM has not shared any of the simplifying assumptions that other techniques promote, e.g., the maximum parallelism hypothesis, and the instantaneous communication assumption.

  The TAM real-time logic is used both as a language in which to express requirements specifications, and as a formalism in which to define the semantics of the TAM language. It is constructed as a conservative extension to first-order predicate logic, and this enables the developer to use the standard first-order proof system. The logic formalise the concept of a *timed variable* which are used to represent real-time program variables and *shunts*. Time is represented by positive integers, and a timing function is used to represent the values found in variables and shunts at a specific time. Specifications are therefore constrains on the relationship between time-stamps and values found in shunts during the lifetime of the system. Additional free variables are also provided which represent the release and termination time of the system; these variables may be predicated over in the usual way and therefore provide a mechanism for specifying duration.

  Concurrency and communication are also provided to describe multi-tasking systems. However, there is no attempt to apply TAM in reverse or reengineering field yet.

- **RTTL** [129, 130]. RTTL (Real-Time Temporal Logic) uses a distinguished temporal domain, the ESM (Extended State Machine) state variables, and the set of ESM transitions to form temporal formula. These are then proven using an axiomatisation of the system's ESM trajectories.

  RTTL has a complex and non-compositional proof system. All of the ESMs have to be designed before any theorems that may be proved about them. There is a decision procedure for finite ESMs but due to the undecidability of predicate logic, a procedure for infinite ESMs can never be found. There is a method for RTTL, but it is basic and contains informal steps. Time is global and there is no maximum parallelism model.

  Perhaps more importantly, RTTL has a very "expressive" syntax, the user can choose either temporal domain expressions or operators. This flexibility may result in "cleaner" specifications.

  No special development method is proposed in RTTL or required by RTTL. If applied to reverse engineering area, RTTL has a flexibility to fit different methodologies.

- **RTL** [94]. RTL is a real-time logic with four basic concepts: *actions* which may be composite or primitive, *state predicates* which provide assertions regarding the physical system state, *events* which are markers on the (sparse) time line, and *timing constraints* which provide assertions about the timing of events.

  Work is presently being carried out on finding an efficient general decision procedure for RTL formulas, presently it is a time consuming exercise to verify safety and liveness assertions using standard deductive proofs. Also, a design method is mentioned which may provide an environment for the engineering of large real-time systems, Jahanian and Mok suggest that RTL may form a unified basis for a theory of decomposition.

  RTL's event occurrence function allows for a rich expression of periodic and non-

periodic real-time properties. However, unstricted RTL is undecidable. It does not treat data structures or infinite state systems. RTL formulas impose a partial order on computational actions which is useful for representing high level timing requirements.

RTL has been used with some success in industrial applications and it is also being used in a major IBM project called "ORE" which is integrating RTL with a real-time programming language. There is a feeling of confidence with RTL due to its pragmatic nature.

- **TPCTL** [80]. Timed Probabilistic Computation Tree Logic (TPCTL) deals with real-time constraints and reliability. Formulas of TPCTL are interpreted over a discrete time extension of Milner's Calculus of Communication Systems called TPCCS. Probabilities are introduced by allowing two types of transitions, one labelled with actions and the other labelled with probabilities.

  The semantics of TPCTL is defined over the reactive transitions of TPCCS processes. TPCTL is a logic essentially extending the branching time modalities of CTL [49] with time and probabilities. Since formulas are interpreted over TPCCS processes, which are observed through actions that label transitions, the semantics of TPCTL is defined in terms of transitions rather than states.

  TPCTL is one of the few logics that can express both hard and soft real-time deadlines, and it is possible to represent levels of criticality in TPCTL.

  Because of the action-based nature of TPCTL, it is difficult to specify state-based properties such as "henceforth, if the train is at the crossing then the gate must be down". Propositions such as "the gate is down" must be encoded indirectly through actions that change the state of the model, in which case the specification becomes unnecessarily complicated.

  TPCTL has no special development method. However, no practice has been carried out that using TPCTL as an independent tool to specify real-time systems.

Sample Description ITL is used as a sample here.

- **Syntax and Semantics**. An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to one less than the number of states in the interval (e.g., a one state interval has length 0). The syntax of ITL is defined as following, where i is a constant, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

Expressions:

$$exp ::= i \mid a \mid A \mid g(exp_1, ..., exp_n) \mid \iota a : f$$

Formulae:

$$f ::= p(exp_1, ..., exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1;\ f_2 \mid f^*$$

The informal semantics of the most interesting constructs are as following:

  - $\iota a : f$: the value of a such that f holds. If there is no such an a then $\iota a : f$ take an arbitrary value from a's range.
  - $\forall v \bullet f$: for all v such that f holds.
  - skip: unit interval(length 1).
  - $f_1;\ f_2$: holds if the interval can be decomposed("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
  - f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

The formal semantics is as followings: Let $\mathcal{X}$ be a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ

and σ' are identical with the possible exception of their mapping for the variable v.

- $\mathcal{E}_\sigma[\![v]\!] = \sigma_0(v)$
- $\mathcal{E}_\sigma[\![g(exp_1, ...exp_n)]\!] = \hat{g}\ (\mathcal{E}_\sigma[\![exp_1]\!], ..., \mathcal{E}_\sigma[\![exp_n]\!])$
- $\mathcal{E}_\sigma[\![\imath a : f]\!] = \begin{cases} \mathcal{X}(u) & \text{if } u \neq \{\} \\ \mathcal{X}(\mathrm{Val}_a) \text{ otherwise} \end{cases}$
  where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \wedge \mathcal{M}_\sigma[\![f]\!] = \mathrm{tt}\}$
- $\mathcal{M}_\sigma[\![p(exp_1, ..., exp_n)]\!] = \mathrm{tt}$ iff $\hat{p}\ (\mathcal{E}_\sigma[\![exp1]\!], ..., \mathcal{E}_\sigma[\![exp_n]\!])$
- $\mathcal{M}_\sigma[\![\neg f]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f]\!] = \mathrm{ff}$
- $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f_1]\!] = \mathrm{tt}$ *and* $\mathcal{M}_\sigma[\![f_2]\!] = \mathrm{tt}$
- $\mathcal{M}_\sigma[\![\forall v \bullet f]\!] = \mathrm{tt}$ iff for all σ' s.t. $\sigma \sim_v \sigma'$, $\mathcal{M}'_\sigma[\![f]\!] = \mathrm{tt}$
- $\mathcal{M}_\sigma[\![\mathsf{skip}]\!] = \mathrm{tt}$ iff $\mid \sigma \mid = 1$
- $\mathcal{M}_\sigma[\![f_1; f_2]\!] = \mathrm{tt}$ iff

  (exists a k, s.t. $\mathcal{M}_{\sigma_0 ... \sigma_k}[\![f_1]\!] = \mathrm{tt}$ and

  ($(\sigma$ is infinite and $\mathcal{M}_{\sigma_k ...}[\![f_2]\!] = \mathrm{tt})$ or

  (σ is finite and $k \leq \mid \sigma \mid$ and $\mathcal{M}_{\sigma_k ... \sigma_{\mid\sigma\mid}}[\![f_2]\!] = \mathrm{tt}$)))

  or (σ is infinite and $\mathcal{M}_\sigma[\![f_1]\!]$)
- $\mathcal{M}_\sigma[\![f^*]\!] = \mathrm{tt}$ iff

  if σ is infinite then

  (exist $l_0, ..., l_n$, s.t. $l_0 = 0$ and

  $\mathcal{M}_{\sigma_{l_n} ...}[\![f]\!] = \mathrm{tt}$ and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)

  or

  (exists an infinite number of l_i s.t. l_0 and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)

  else

  (exist $l_0, ..., l_n$ s.t. $l_0 = 0$ and $l_n = \mid \sigma \mid$ and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)

- **Assessment**. ITL was first proposed by Moszkowski [124]. ITL avoids the proliferation of time variables in specifications, as do all temporal logics. ITL is sufficiently general to express any discrete computation. An executable subset of ITL, called Tempura [125], is well developed. Zedan and Cau proposed a set of new refinement rules for ITL [38], which gives ITL a strong ability to describe all the popular possible features of real-time systems. Since ITL has an executable subset Tempura, its verification and simulation can be largely facilitated. The development method of ITL fits popular reengineering methodologies well. Generally speaking, ITL is a formal logic with enough expressibility of real-time systems and suitable for reengineering methodologies.

2.6.3 Algebraic Approach

General In this approach, an explicit definition of operations is given by relating the behaviour of different operations without defining states. Similar to the model-based approach, there is no explicit representation of concurrency.

Examples

- **OBJ**[75, 76]. OBJ is a wide spectrum first-order functional language that is rigorously based on equational logic. This semantics basis supports a declarative, specificational style, facilitates program verification, and allows OBJ to be used as a theorem prover.

- **LARCH** [79]. The Larch family of algebraic specification languages was developed at MIT and Xerox PARC to support the productive use of formal specifications in programming. One of its goals is to support a variety of different programming, including imperative languages, while at the same time localising programming language dependencies as much as possible. Each Larch language is composed of two components: the *interface* language which is specific to the particular programming language under consideration and the *shared language*

which is common to all programming languages. The interface language is used to specify program modules using predicate logic with equality and constructs to deal with side effects, exception handling and other aspects of the given programming language. The shared language includes specification-building operations inspired by those in CLEAR, although these are viewed as purely syntactic operations on lists of axioms rather than as semantically non-trivial as in CLEAR.

Sample Description OBJ is used as a sample here.

OBJ [75] is a broad spectrum algebraic specification/programming language based on order sorted equational logic. It is a specification language in which an algebra is defined using *objects*. Objects are carrier sets along with operations, and equational theories which are treated by the OBJ interpreter as re-write axioms. Each object is built from primitive *sorts* and enrichments of existing objects.

Proofs of equivalence are achieved automatically in OBJ by rewriting processes into their normal forms and testing for syntactic equivalence.

There are now a number of enhanced OBJ interpreters, including OBJ1, OBJ2 and OBJ3. Here we prefer using the most up-to-date one: OBJ3.

OBJ3 is based on *order sorted equational logic*, which provides a notion of *subsort* that rigorously supports multiple inheritance, exception handling and overloading. OBJ3 also provides *parameterised programming*, which gives powerful support for design, verification, reuse, and maintenance. This approach uses two kinds of module: *objects* to encapsulate executable code, and in particular to define abstract data types by initial algebra semantics; and *theories* to specify both syntactic structure and semantic properties for modules and module interfaces. Each kind of module can be parameterised, where actual parameters are modules. For parameter instantiation, a *view* binds the formal entities in an interface theory to actual entities in a module, and also asserts that the target module satisfies the semantic requirements of the interface theory.

2.6.4 Process Algebra Approach

General In this approach, explicit representation of concurrent processes is allowed. System behaviour is represented by constraints on all allowable observable communications between processes.

Examples

- **CSP** [82, 86]. The Communicating Sequential Processes (CSP) formal specification notation for concurrent systems was first introduced in [82]. Since this original proposal did not include a proof method, a complete version of CSP was proposed in [86].

- **CCS** [121, 123]. Calculus of Communicating Systems (CCS) was proposed by Milner in 1989. It is a formalism similar to CSP. CCS is also suitable for distributed and concurrent systems. At present, several variations of CCS has been developed, which forms a CCS family. CCS family includes CCS, CCS+, CCS*, SCCS, TCCS and TPCCS [64].

Two underlying concepts of CCS are *agents* and *actions*. A CCS model consists of a set of communicating processes (agents in CCS terminology). CCS adopts operational semantics.

CCS is a successful formalism to build system models with respect to concurrency and distribution. Compared with CSP, the emphasis of CCS is on defining a series of equivalencies (bisimulations), each equivalence defining a different model of concurrency. Thus certain processes that might be considered identical in CSP, would be different in CCS. CCS has a form of modal logic to specify the observable behaviours of processes. CSP has a richer set of laws than CCS allowing for optimising design and implementations. CCS concentrates on a minimal set of operators needed for the full expression of non-deterministic concurrency and its resulting equivalences.

CCS is not a real-time formalism either. Some extensions of CCS with real-time feature have been developed, such TCCS, SCCS, and TPCCS.

- **ACP** [26, 15]. Algebra of Communicating Processes (ACP) was proposed by J.A. Bergstra in 1984. Until now, a rather large variety of ACP has been proposed, such as Real Time ACP(ACP_ρ), Discrete Time ACP. ACP is also an action-based process algebra, which may be viewed as a modification of CCS. However, ACP is an executable formalism. ACP is equipped with a process graph semantics, and adopts bisimulation proof system. ACP allows a variety of communication paradigms, including ternary communication, through the choice of the communication function.

- **LOTOS** [93, 113]. LOTOS (Language Of Temporal Ordering Specification) was developed to define implementation-independent formal standards of OSI services and protocols. LOTOS has two very clearly separated parts. The first part provides a behavioural model derived from process algebra, principally from CCS but also from CSP. The second part of LOTOS allows specifiers to describe abstract data types and values, and is based on the abstract data type language ACT ONE.

  By combining the two formalism, CCS/CSP and ACT ONE integrally, LOTOS has a strong ability to describe both the "data" and "control" of the systems, i.e., ACT ONE for the data part and CCS/CSP-based language for the control part. LOTOS is able to capture a relatively complex temporal pattern of events, involving non-determinism, concurrency and synchronisation, by means of small algebraic expression built by using few conceptually simple operators [156].

  LOTOS has formally defined syntax, static semantics and dynamic semantics. The static semantics are defined by an attributed grammar [93] and the dynamic semantics are described operationally in terms of inference rules.

  Since LOTOS has an operational semantics, it is possible to implement these

semantics in an interpreter. LOTOS has "a number of" various support tools, which are although not mature or narrow-aspected, do have some successful points [157].

LOTOS does not support real-time specifications. Although a Timed LOTOS has been proposed, it is not proven a suitable formalism for real-time systems.

LOTOS has problems in specifying distributed systems - it does not support dynamic reconfiguration which is an important and interesting characteristics in those systems. Its model of concurrency is based in the known "interleaved semantics" in which an observer can see one event a time and concurrency is represented sequentially. Many models based on "true concurrency semantics" have been proposed although it seems that none of them will be present in the next version of LOTOS. This is a weak point in represent distributed processing where in many situations things happen simultaneously and no ordering between events can be established. Also, LOTOS has weak data specification mechanisms and cannot express time explicitly.

- **TCSP** [136]. Timed CSP is an extension of Hoare's CSP, with a dense temporal model providing a global clock. A delay operator is included along with some extended parallel operators. There is an assumption of a minimum delay between any two dependent action occurrences, but no minimum delay on any two independent actions. The semantics of TCSP is given by timed traces, and a specification relation *sat* is provided for verifying predicates over traces.

  Processes in Timed CSP are built from sequences of communication actions. The semantic model of TCSP is based on observation and refusal *timed traces*.

It is important in specification languages for real-time systems that the temporal relationships between actions are maintained through the manipulation of the specification. In a non-real-time process algebra, the concurrency operators are usually *conservative*, i.e., they degenerate into non-deterministic interleaves of

the constituent processes' actions. In TCSP, the concurrency operators are non-conservative, they do not degenerate. Instead, the algebraic rules for concurrent processes define the effect of composition on the temporal domain. For example, in the process where there is concurrent composition of two *WAIT* processes, the result is a process that waits for the maximum of two delays.

There exist no tools for the manipulation of specifications written in TCSP.

- **TPCCS** [80]. Timed Probabilistic Calculus of Communicating Systems (TPCCS) [80] is essentially an extension of Milner's CCS with discrete time and probabilities. To increase the description ability, a logic named Timed Probabilistic Computation Tree Logic (TPCTL) is proposed to describe the logic of and between TPCCS processes. Therefore TPCCS, together with TPCTL, forms a framework for specification and verification of real-time and reliability in distributed systems. TPCCS, as a process algebra, is used for modelling the operational behaviour of distributed real-time systems; and TPCTL, as a logic, is used for expressing properties of the systems. A verification method for automatically proving that a system described in TPCCS satisfies properties formulated in TPCTL, is also well defined [80].

The main advantage of TPCCS is that it has a powerful description ability for real-time distributed systems. TPCCS can reason about both time and probabilities in distributed systems. In particular, TPCCS

- extends CCS with probabilities by adding a probabilistic choice operator and by introducing a probabilistic transition relation,
- adds discrete time to the extended CCS where the timing model is based on a minimal delay assumption, i.e., communications must occur as soon as possible,
- defines a strong bisimulation equivalence for which a sound and complete axiomatisation is given.

TPCCS has very formally defined syntax and semantics, which bring lots of convenience in the automation of specification and verification. However, the calculation of probabilities is not mentioned in TPCCS and TPCTL. A tool named Timing and Probability Workbench (TPWB) has been developed. TPWB Partially supports automatic verification of TPCCS.

Sample Description CSP is used as a sample here.

- **Syntax and Semantics of CSP** A CSP specification is a hierarchy of processes. A complete specification can be viewed as a single process which is composed of sub-processes, each of which is decomposed into component processes.

  The CSP notation has three primitive processes for input, output and assignment:

  $A!e$ Output the value e over channel A;

  $B?x$ From channel B input to x;

  $x ::= e$ Assign x the value e.

A number of operators exist for combining processes, for example:

$P||Q$ Processes P and Q operate in parallel;

$P \sqcap Q$ Either P or process Q operates. The choice is non-deterministic;

$P;\ Q$ Process P operates followed by Q.

The basic concept in CSP considers a process as a mathematical abstraction of interactions between the system and its environment. Recursion is used to describe long lasting processes. The second feature is to use traces to record the sequence of actions a process has carried out. The abstract description is then given a more concrete explanation using algebraic law, and the last step is the implementation.

The notation for CSP uses *first order logic* symbols plus some additional symbols for *traces, functions*, etc.

There is a family of increasingly sophisticated models for providing CSP specification semantics. These computational models include the *counter model*, the *trace model*, the *divergences model*, the *readiness model* and the *failure model*.

- **Assessment of CSP**

  The main contribution of CSP is as a programming language for parallel processing, principally in the area of synchronising communications.

  CSP supports an event model that enables the description of entities that have properties and relationships that vary over time. It allows us to model a dynamic reality, to specify systems that perform various actions in particular orderings, and to express timing constraints between these actions and on the synchronisation of various system components.

  CSP specifications may be viewed quite simply as a system of processes executing independently, communicating over unbuffered unidirectional channels, and synchronising on particular events.

  Specifications may be manipulated through the application of a number of algebraic laws, and combined by means of a small number of operators which are known to be sound. Various semantic models allow proposed properties to be proven, and to demonstrate that particular requirements have been satisfied.

  These features make CSP a suitable formalism in the area of concurrency. However, like many other methods/languages, timing constraints associated with *real-time* operations cannot be handled, or have to be in a clumsy and inefficient way. CSP is not good at handling asynchronous events, such as interrupts.

  Tools for CSP keep emerging. The Occam Transformation System developed by Oxford University's Programming Research Group is an automated tool to assist in carrying out algebraic transformation. Since Occam follows the main principles of Hoare's CSP, this tool may bring some convenience to CSP, too. FDR (Failures-Divergence Refinement) was the first commercially available tool for CSP and played a major role in driving the evolution of CSP from a blackboard notation to a practical language. As a prover tool, FDR allows the checking of a wide range of correctness conditions of finite state systems, including deadlock and livelock freedom as well as general safety and liveness properties.

When these conditions are not satisfied, the reasons can be investigated. FDR is a product of Formal Systems, a consultancy firm specialising in the industrial application of formal methods.

2.6.5 Net-Based Approach

General Graphical notations are popular notations for specifying systems as they are easier to comprehend and, hence, more accessible to non-specialists. In this approach, graphical languages with a formal semantics are used, which bring special advantages in system development and reengineering.

Examples

- **Petri Net** [137, 133]. Petri Net theory is one of the first formalisms to deal with concurrency, nondeterminism and causal connections between events. According to [122], it was the first unified theory, with levels of abstraction, in which to describe and analyse all aspects of computer in the context of its environment.

  Petri nets provide a graphic representation with formal semantics of system behaviour. Until now, a large amount of varieties of Petri Net Theory has been proposed. Generally, petri nets can be classified into ordinary (classic) petri nets and timed petri nets.

- **Timed Petri Net** [118, 63, 27, 29, 105, 135]. Petri Net theory was the first concurrent formalisms to deal with real-time. Two basic timed versions of Petri nets have been introduced: Timed Petri Nets [134] and Time Petri Nets [118]. Both have been used in recent work [63, 27, 29, 105, 135]. There are two questions that arise when time is introduced to net theory: (i) the location of the time delays(at places or transitions) and (ii) the type of delay (fixed delays, intervals or stochastic delays).

  Timed Petri Nets are derived from classical Petri nets by associating a finite firing duration (a delay) with each transition of the net. The transition is disabled from

occurrence for the delay period, but is fired immediately after becoming enabled. These nets are used mainly in performance evaluation.

Time Petri Nets (TPNs) are more general than Timed Petri Nets. A Timed Petri Net can be simulated by a TPN, but not vice versa. Both a lower and an upper bound are associated with each transition in a TPN. A state in the reachability graph is a tuple consisting of a marking, and a vector of possible firing intervals of enabled transitions in that marking.

- **Statecharts** [89, 90]. Statecharts provides an abstraction mechanism based on finite state machine. It represents an improved version of the structured methods. A graphic tool called "Statemate" [4] exists to implement the formalism. Methods similar to that of Statecharts may be found in [68].

  Statecharts have been proved to be at least as expressive as state machines, and the succinct justification for them is provided by the following "equation":

  $$Statecharts = state\text{-}transitions + depth + orthogonality + broadcast\,communication.$$

  In statecharts, conventional finite state machines are extended by AND/OR decomposition of states, interlevel transitions, and an implicit intercomposition broadcast communication. Statecharts denote composition of state machine into super-machines which may execute concurrently. The state machines contain transitions which are marked by enabling and output events. It is assumed that events are instantaneous, and a global discrete clock is used to trigger sets of concurrent events. Statecharts are hierarchical, and may be composed into complex charts. The semantics of Statecharts is given by maximal computation histories. An axiomatic system is presented.

  Statecharts supports typical structural top-down system development method. It does not fit the procedures of reverse engineering, which abstracts specifications from source code. Real time is incorporated in Statecharts by having an implicit clock, allowing transitions to be triggered by timeouts relative to this clock and

by requiring that if a transition can be taken, then it must be taken immediately [151].

Sample Description Petri Net is used as a sample here.

- **Syntax and Semantics of Petri Nets**

The classic Petri Nets model is a 5-tuple (P, T, I, O, M). P is a finite set of places (often drawn as circles), representing conditions. T is a finite set of transitions (often drawn as bars), representing events. I and O are sets of input and output functions mapping transitions to bags of places(the incidence functions). M is the set of initial markings.

Places may contain zero or more tokens (often drawn as black circles). A marking (or state) of the Petri nets is the distribution of tokens at a moment in time, i.e. $M : P \rightarrow N$ where N is the non-negative integers. Tokens in Petri nets model dynamic behaviour of systems. Markings change during execution of the Petri nets as the tokens "travel" through the net.

The execution of the Petri nets is controlled by the number and distribution of the tokens (the state). A transition is enabled if each of its input places contains at least as many tokens as there exists arcs from that place to the transition. When a transition is enabled it may fire. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

Given an initial state (distribution of tokens), the reachability set is the set of all states that result from executing the Petri net. Properties such as boundness, liveness, safety and freedom from deadlock can be checked by analysing the reachability graph. The reachability graph is usually constructed using an interleaving operational semantics.

In Petri nets causal dependencies and independencies in some set of events are explicitly represented. It is therefore easy to provide a non-restrictive partial or-

der semantics. Events which are independent of each other are not projected onto a linear time scale. Instead a non-interleaving partial order relation of concurrency is introduced.

- **Assessment of Petri Nets**

The advantages of using Petri nets are numerous. They are easy to comprehend due to their graphical form, they can be used to model hardware, software and human behaviour, and they allow formal reasoning of system behaviour. Some experts suggest using both Petri nets and formal logic for developing systems and the former to model the system, the latter to verify it.

Ordinary Petri nets have been criticised for not being able to deal with fairness and data structures, e.g. the data in a measure header, although the number of tokens at a particular place in the net can simulate a local program variable. Structuring mechanisms such as composition operators are not inherently part of the theory, and there is no calculus to transform a net into a real-time programming language. Unlike state machine, a "place" in a Petri net cannot easily be identified with a place in the corresponding program code. A further problem is that the reachability graph suffers from state explosion as Petri nets become larger, thus impacting on the ability to scale up analysis to larger systems. Ordinary Petri nets are still an object of intense research aimed at putting Petri nets theory on firm mathematical ground. However, practically speaking, such standard nets are not up to the task of modelling complex systems. For this reason, higher level nets (coloured nets) and stochastic nets have been introduced to extend the modelling power of Petri nets.

2.7 Criteria and Results

In this section, we summarise a wide spectrum of existing formal methods from the point of view of software reengineering. Generally speaking, some of them already

have a rather good advantage in certain aspects, such as ITL for real-time systems, and TPCCS & TPCTL for systems with reliability and probabilities. However, all of them have certain flaws or weakness in some aspects as described in section 2.6.

We list our findings through the review in forms of tables according to the following criteria:

- Temporal Model – Temporal model is the model of time used by the formal methods. A sparse model has discrete instances of time and there is a minimum granularity. A dense model is not discrete, between any two instances in time there is an infinite number of other instances.

- Automated Tools – This criterion refers to whether the formal method has relevant automated tools to support its development, such as checking syntax, verifying semantics and auto-execution.

- Reliability – This criterion refers to the reliability of the formalism.

- Proof System – This refers to whether there is any proof system and what the type of the system is (when there is one).

- Industrial Strength – This criterion refers to the potential of the formal method for large-scale/industrial applications.

- Methods of Verification – This criterion refers to the existing methods of verification of the formal method. Normally, there are two types of the methods of verification: model checking and theorem proving.

- Concurrency – This criterion refers to the explicit representation and reasoning of *concurrency*.

- Communication – This criterion refers to the explicit representation and reasoning of *communication*.

- Reverse Engineering – This criterion refers to whether the formal method has been applied in any reverse engineering domain.

Criteria	Z	VDM	B
Temporal Model	none	none	none
Automated Tools	a few	none	good
Reliability	good	good	good
Proof System	semi-axiomatic	semi-axiomatic	axiomatic
Industrial Strength	great	some	great
Methods of Veri.	model-checking	model-checking	both
Concurrency	none	none	none
Communication	none	none	none
Reverse Eng.	yes	no	no

Table 2.1: Model/State-Based Formalisms

Criteria	HL	WP-Calc.	TL	ML
Temporal Model	none	none	dense/sparse	none
Automated Tools	some	some	some or few	few
Reliability	good	good	good	good
Proof System	axiomatic	axiomatic	axiomatic	axiomatic
Industrial Strength	some	some	great	great
Methods of Veri.	theorem proving	theorem proving	both	both
Concurrency	none	none	norm exist	none
Communication	none	none	norm exist	none
Reverse Eng.	yes	yes	no	no

Table 2.2: Logic-Based Formalisms

The above five categories are corresponding to subsections of section 2.6. We believe we should use the sixth category in order to better summarise those so-called "combined" approaches.

Through reading these tables, we can draw the following conclusions of the current situation of formal methods for reengineering:

- Some formalisms are rather good in certain aspects of software development while others are good in other aspects. For example, ITL has a strong ability

Criteria	ITL	DC	TAM	RTTL	RTL
Temporal Model	sparse	dense	sparse	sparse	sparse
Automated Tools	few	none	none	few	none
Reliability	good	good	good	good	good
Proof System	axiomatic	axiomatic	axiomatic	axiomatic	axiomatic
Industrial Strength	great	some	great	some	some
Methods of Veri.	theorem prov.	theorem prov.	theorem prov.	theorem prov.	theorem prov.
Concurrency	par. comp.	none	exist	interleaved	interleaved
Communication	sync./async.	none	exist	sync.	none
Reverse Eng.	no	no	no	no	no

Table 2.3: Logic-Based Formalisms

Criteria	OBJ	Larch
Temporal Model	none	none
Automated Tools	few	some
Reliability	good	good
Proof System	axiomatic	axiomatic
Industrial Strength	some	great
Methods of Veri.	theorem prov.	theorem prov.
Concurrency	interleaved	interleaved
Communication	sync.	sync
Reverse Eng.	no	no

Table 2.4: Algebraic Formalisms

for representing and reasoning of most features of real-time systems. TPCCS & TPCTL is good at dealing with systems with reliability and probability features. Z is capable for large-scale industrial applications. B has a comprehensive automated toolkit. DC has advantages for its ability of dealing with dense temporal models. Various process algebras are excellent for their abilities of representing and reasoning of concurrency and communication. Finally, the most important features of net-based formalisms are their graphical representations: concise, easy to understand, and very clear.

- Only a very few formalisms have been applied as the theoretical foundation of

Criteria	CSP	CCS	ACP	LOTOS	TCSP
Temporal Model	none	none	none	none	dense
Automated Tools	some	none	good	some	none
Reliability	good	good	good	good	good
Proof System	axiomatic	bisimulation	bisimulation	bisimulation	axiomatic
Industrial Strength	some	some	some	great	some
Methods of Veri.	both	both	both	model-checking	both
Concurrency	interleaved	interleaved	interleaved	interleaved	both
Communication	sync/async.	sync.	sync.	sync.	sync.
Reverse Eng.	no	no	no	no	no

Table 2.5: Process Algebra Formalisms

Criteria	Petri Nets	Timed Petri Nets	Statecharts
Temporal Model	none	dense/sparse	sparse
Automated Tools	some	none	none
Reliability	good	good	good
Proof System	reachability	reachability	axiomatic
Industrial Strength	some	some	some
Methods of Veri.	model-checking	model-checking	model-checking
Concurrency	interleaved	interleaved	exist
Communication	sync.	sync.	sync.
Reverse Eng.	yes	no	no

Table 2.6: Graphic-Based Formalisms

reverse engineering;

- Although some formalisms are suitable for certain stages of reverse engineering, there is not any formalism covering all reverse engineering stages. For example, Hoare Logic can cope with the low-level abstraction of program source code, but not high level abstraction. This also happens to the formalisms such as Wp-Calculus and predicate logic. Therefore, a new wide spectrum formalism is needed for the reengineering process, e.g., an ITL-based wide spectrum language with real-time features.

It is not hard to see that most existing formal methods were not designed for reverse

Criteria	TPCCS + TPCTL	Petri Nets + Predicate
Temporal Model	sparse	sparse/dense
Automated Tools	none	none
Reliability	good	good
Proof System	axiomatic	reach. ps axiom.
Industrial Strength	some	unknown
Methods of Veri.	theorem proving	model-checking
Concurrency	interleaved	interleaved
Communication	sync.	sync.
Reverse Eng.	no	no

Table 2.7: Table 6: Combined Formalisms

engineering as well as reengineering. This urges that research into suitable formal methods for reengineering should be established..

2.8 Analysis and Conclusion

This review is conducted in the view of developing a practical approach for the reengineering of existing system including real-time critical application. Reengineering generally consists of three stages, i.e., restructuring, reverse engineering and forward engineering. Because most existing formal approaches were developed for forward engineering, whether a formal approach has been used for reverse engineering is specially used as a criterion.

Through the review, we found that using formal methods in reverse engineering existing systems (real-time systems, in particular) is still a research area that has not been addressed properly, because (1) there are formal methods for reverse engineering; (2) a new wide spectrum formalism supporting various abstraction levels is needed for the reengineering process, and (3) even if a formal method can cope with reverse engineering well, it is still a problem whether this formal method can be integrated with an existing matured forward engineering formal method.

Graphical notations are also popular notations for reverse engineering (understand-

ing) existing systems. The Petri Net is useful for building a graphical model for reengineering.

Another factor that should be taken into consideration when reengineering computing systems is recent rapid development of object-oriented technology. We believe that an approach that integrates formal methods, particular system domain features and object-oriented techniques can contribute to improve reengineering:

- existing software can be easily understood and reengineered with the help of a successfully extracted semantics-oriented specification. An approach with a full consideration about the features of the system being reengineered will be more effective and efficient.

- object-oriented techniques, which have been recognised as the best way currently available for structuring software systems, can help maintenance in grouping together data and operations performed on them, thereby encapsulating the whole system behind a clean interface, and organising the resulting entities in a hierarchy based on specialisation in functionalities;

- formal methods can provide a solid theoretical foundation for the correctness and unambiguity of the approach, meanwhile give more potentials to the automation of the approach, hence, a practical software maintenance and reengineering tool becomes feasible.

Therefore our goal is to devise a uniform coherent semantic theory that enables a comprehensive formal understanding of reengineering model when applied to the analysis and the development of complex computing systems in real applications. This should allow diverse kinds of formalisms to be developed and integrated.

The unified theory provides a formal basis within which an object-oriented formal notation will be developed that unifies existing widely-used formalisms. In addition, sound transformational calculi together with verification and validation techniques will be developed. Our approach to this is to build a wide spectrum language in which

concrete and abstract (e.g. specification statement) system notations could be easily intermixed. The developed calculi will then allow us to transfer from one form of specification/program to another.

The novel aspects of this proposal are in the incorporation of the outcome of extensive research in a number of key areas of software maintenance into a formally unified semantic model.

We intend to use a wide spectrum language approach to the proposed formal reengineering of existing computing systems, particularly real-time systems. Our extensive experience with the design and use of the Wide Spectrum Language (WSL) [22, 167, 47] and TAM [145, 38, 170] have illustrated the practical use of such an approach. In our next research stage, we therefore aim to:

1. develop a single "wide spectrum" language in which both abstract specifications written in our extended logic and executable code may be intermixed in the representation of the target system;

2. define a refinement and abstraction relation on specifications and programs described in the language;

3. develop a family of sound refinement and abstraction calculi to serve for both forward and backward refinement, and

4. treat real-time systems with parallelism as specific domain.

Chapter 3

Related Work

3.1 Maintainer's Assistant

The project, as the main part of a larger project the **ReForm** project funded by IBM and the DTI/SERC, addresses the reengineering of installed software to bring it to a state in which modern software engineering techniques can be applied via the application of formal transformations. Maintainer's Assistant is developed in Software Maintenance Center in Durham University, UK [163, 160]. The structure of the system is shown in the figure below:

The aim of the ReForm project is to create a code analysis tool–the **Maintainer's Assistant** [163, 160, 32, 33], aimed at helping the maintenance programmer to understand and modify a given program. Program transformation techniques are employed by the Maintainer's Assistant both to derive a specification from a section of code, and to transform a section of code into a logically equivalent form. The aim is to provide a tool with features such that:

- It acts, initially, on existing program code as a tool to aid comprehension (possibly by producing specifications);
- Only the program code is required;

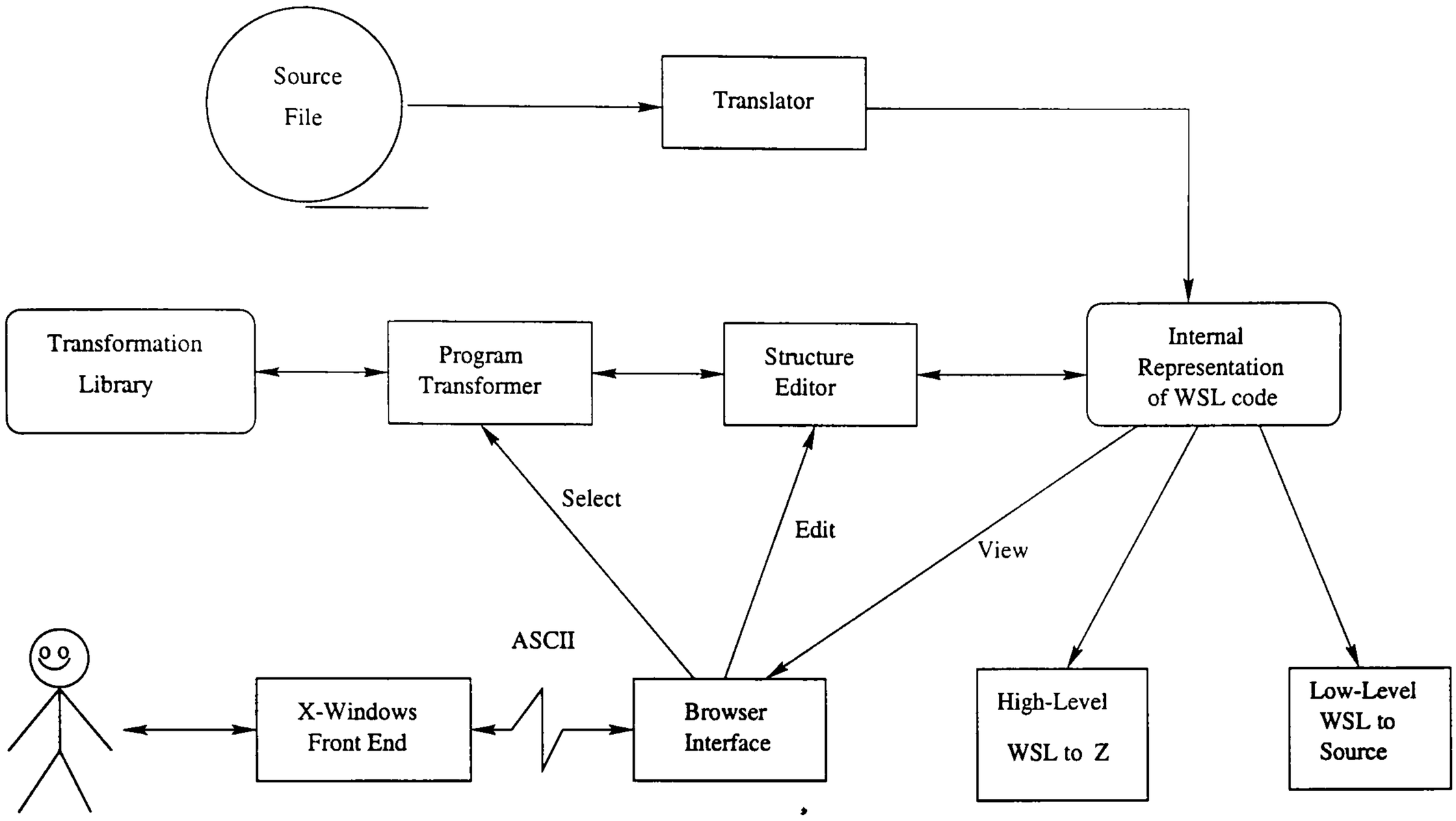

Figure 3.1: Maintainer's Assistant System Architecture.

- The system can work with any language by first translating, i.e., with a stand-alone translator into WSL;
- Changes are made to the WSL program by means of transformation;
- The system incorporates a large, flexible catalogue of transformations;
- The applicability of each transformation is tested before it can be applied;
- The system is interactive and incorporates an X-Windows front end and pretty-printer called the Browser;
- The system includes a database structure to store information about the program being transformed, such as the variables assigned to within a given piece of code;
- The system includes a facility to calculate metrics for the code being transformed.

One of the most important successes of Maintainer's Assistant is that it is based on a wide spectrum language whose syntax and semantics are formally defined. Maintainer's Assistant is a successful case of applying wide spectrum languages in reengineering area. However, Maintainer's Assistant focused on transformations rather than abstraction. It involved very little in how to use multi-leveled abstractions and relevant abstraction rules to reach a good system reengineering, especially reverse engineering. The Wide Spectrum Language in Maintainer's Assistant is sequential and non-timed, which limits its application in domains such as real-time systems.

3.2 CStar and Elbereth

Both CStar [77] and Elbereth [77, 78] are the results of the research conducted at the Software Engineering Laboratory of University of California at San Diego.

Elbereth is a Java Reengineering Tool based on Star Diagrams. It provides powerful ways to view all the uses of a variable, method or class in the context in which it is used. It also supports the recording and recall of plans for system-wide changes, meaning that the tool not only provides visualisations of a program, but of a programmer's work as well.

CStar is a C Reengineering Tool based on Star Diagrams. It provides much the same functionality as Elbereth, but for C. It is more mature in some ways, supporting capabilities such as building a star diagram for all variables of a particular type.

The work at the Software Engineering Laboratory, Department of Computer Science and Engineering, University of California at San Diego, is one of the representative examples of research that are related to our work. They based their approach to reverse engineering on abstraction, and identified three kinds of abstractions: problem domain, structural, and logical. Problem domain abstractions correspond to concepts from a program's application area. Structural abstractions are used to eliminate implementation details and redundant information. Logical abstractions are properties that can be logically derived from code. The goal in logical abstraction is not to generate

abstract program description, but to be able to determine the validity of specified properties of a program's context/action pairs. Logical abstractions can be thought of as properties that can be derived from the adopted event/state model (ESM).

A method for generating functional specifications is described, which incorporates the abstraction techniques. It has been applied to a variety of COBOL programs and been found to generate 'natural' abstract program descriptions. An analysis tool is being constructed that will be used to help verify the approach and to assess its complexity and computational requirements [91]. A prototype system of program understanding and reverse engineering called Function And Context Extraction Tool was under construction.

However, the work at University of California at San Diego is based on the *action/context* paradigm rather than wide spectrum languages, and as a consequence, the approach is not formalised, and there do not exist consistent multiple abstraction levels with an integrated formal semantics. These limit the accuracy and power of their approach. Although a comprehensive and general description was given about their approach [91], more actual work needs to be carried out, especially abstraction rules, i.e., rules to reach the proposed abstraction.

3.3 PRISME

PRISME is a reverse engineering tool based on functional abstraction developed in the Department of Information Technology, University of Paris [17, 16]. The developers propose to re-document programs with outlines. The interest of outlines is that they allow to contract, as in a zoom, the amount of information necessary to understand programs, easing walking through them to localise given computations or to identify the role of a piece of code. As a first stage toward a framework of program outlines, a model is defined which is suitable to the representation of computations performed within loops. The main feature of the outlines is that they are both formal and conceptual: they are represented within frames which are semantically equivalent to the outlined loops

and help understanding what is computed by revealing how this is computed. PRISME is a system for program re-documentation, it is able to automatically construct outlines of a subset of Lisp looping functions.

However, the abstraction in PRISME is function-based instead of semantics-based.

PRISME does not involve wide spectrum languages. It does not engage a mature formal method to specify the target system, therefore, PRISME can only extract simple 'signatures' as pieces of outline description of the system, no complete specification can be extracted in PRISME. Moreover, the notations in PRISME lack of integrated semantic foundation.

PRISME does not contain any abstraction rule to carry out its proposed abstraction for re-documentation, and consequentially, it involves no formal definition of abstraction and relevant rules.

PRISME is only capable for a narrow subset LISP looping functions, not for a variety of real computing systems, both procedural and object-oriented.

PRISME does not consider any real-time or object-oriented systems.

3.4 AUTOSPEC

In the Software Engineering Research Centre (SERC) of the Department of Computer Science, Michigan State University, efforts of using formal methods to reverse engineering and reengineering have been made [42, 43, 44, 74, 72, 73].

The project involves a two-phase approach to reverse engineering that integrate a process for abstracting formal specifications from program code with a technique for identifying candidate objects in program code. Thus far, a set of procedures for abstracting formal specifications from program code by translating basic programming constructs into equivalent formal representations (predicate logic) has been developed. Specially, they have developed procedures to handle assignments, alternatives and iteratives. In all the cases, the weakest precondition (*wp*) as defined by Dijkstra and Gries is used in the abstraction process. For all programming statements, there is a *wp*

predicate transformer that is used to define the semantics of the statement with respect to a postcondition R. The wp is the set of all states in which a given statement S can begin execution and upon termination, postcondition R is true. The abstraction process begins with the programming statement S and seek the postcondition R using wp definitions to guide the derivation. Until now, a preliminary prototype *AUTOSPEC* has been developed to apply these procedures to program code.

Apparently, this project only deals with the first abstraction step of reverse engineering, i.e. it only extracts an abstraction at the lowest level of specification, in the form of predicate logic as a notation of the source code. Therefore, AutoSpec only considered the initial step in the whole process of reversing source code into a system specification. There is no multiple levels or high levels of abstraction in AutoSpec.

3.5 Other Related Software Reengineering Projects

Here we list some other software reengineering projects we have found, however, none of these project are directly related with formal methods.

Chopshop Project The Chopshop project is carried out in the School of Computer Science, Carnegie Mellon University. It aims at providing practical analysis and visualisation tools to assist with real software engineering task. Chopshop is guided by a number of aspirations:

- To focus on commonly-used programming languages.
- To provide analysis that are efficient even when applied to very large systems.
- To provide analyses with firm theoretical foundations—the results of the analyses should be translated into claims about the behaviour of analysed programs.
- To present results at different levels of abstraction appropriate for the task at hand.

A program slicing tool Chopshop has been built, which computes the dataflow dependencies of C code and displays them using a variety of abstraction mechanisms. Chopshop is a reverse engineering tool to help programmers understand unfamiliar C code. A new dataflow analysis technique is developed, which is a modular generalisation of static program slicing. It gives more understandable results than standard formulations of slicing. The user can select several sources and sinks of information, and Chopshop shows how data flows from the sources to the sinks.

DARPA EDCS project [60] The Evolutionary Design of Complex Software (EDCS) Program, sponsored by the Defence Advanced Research Projects Agency (DARPA), USA, addresses the need for military systems to evolve over extended lifetimes. The program is based on the observation that the most likely way to make an existing system adapt to changes in its operational environment is through changing its software. The program examines the ways that software can be created to be more easily evolved, defines methods for incremental adaptation of systems through software changes, and seeks ways to migrate the currently installed base of military systems to more evolutionary systems.

Rigi project [120, 155] This project is carried out at the University of Victoria, Canada. Rigi is a Software Engineering project being conducted by researchers in the Department of Computer Science at the University of Victoria. The current focus of the group is visualisation support for the understanding and reverse engineering of legacy systems. This support is embodied in the form of a general Rigi graph model and realised in an editor called RigiEdit. Inherent in the model is the notion of nested subsystems that encapsulate detail and provide high level overviews of software systems. Recent work has generalised both the model and the tool to allow the use of the graph editor in other domains and to allow user defined extensions to the built-in Rigi Command Library (RCL).

The RevEngE program understanding project [35, 126] The objective of this three-year project, carried out at University of Victoria in conjunction with IBM Canada Ltd.'s Centre for Advanced Studies (CAS), is to develop an integrated environment offering tools for subsystem identification and discovery to support reverse engineering processes, using a common software repository. In particular, the project addresses issues in the areas of software analysis technology, algorithms to extract system abstractions, integration technology applicable to CASE, user-interface technology to model, browse, and search large collections of software artefacts and reverse engineering process models interactively.

The objectives for the first year of the project are two-fold: to design and build a prototype environment for reverse engineering consisting of a software repository and a set of reverse engineering tools, and to investigate specific reverse engineering problem domains. These goals have been met: the three systems on display by McGill University, the University of Toronto, and the University of Victoria highlight the contributions made by the research partners, both individually and collectively, towards addressing the challenges of reverse engineering.

PURE project [8, 65] The Program Understanding and Reengineering (PURE) project is at IRST, Italy. The goal is to develop technologies to analyse software systems or sub-systems, either at a fine-grained level (control and data dependencies) or at a more coarse grained level (systems' high level structure and behaviour, i.e. software architecture), aiming at evaluating system characteristics, supporting user-assisted migration or restructuring, and more generally increasing software artefacts quality.

First code analysis activity is focused on supporting program understanding, maintenance, quality evaluation and assurance. The main effort will be in the area of interprocedural analysis among which data dependence, control dependence, slicing, pointers and arrays are analysed. An intermediate language representation allows being independent from the source programming language, given a front-end which translates the code in the intermediate language. Results can be saved in textual form, or a user

interaction with the analyses is supported by a customised version of the text editor EMACS.

Analysis of source code at the architectural level is motivated by the fact that the first activity performed by maintenance programmers when approaching the task of understanding a system is often trying to discover its high level structure, that is, identifying its subsystems and their relations: in few words, the software architecture of the system. First, software architecture goal is to identify architectural patterns and styles for distributed and object-oriented systems and to develop technologies to identify components and relations according to the defined patterns, evaluate the quality of extracted design and to support sound architecture recovery and migration across different architectural styles.

Type-based analysis of C programs The research is carried out at the program analysis group at Microsoft Research, USA. One of the goals is to investigate whole-program analysis of large programs, which mean industrial size large programs rather than academic size large programs. The research aims at programs consisting of around a million lines of C or C++ code.

The project has found that flow-sensitive inter-procedural data-flow based analysis algorithms often do not scale well. The current state-of-the-art algorithms are not able to compute results for large programs in reasonable time given reasonable memory constraints, even given generous definitions of reasonable.

The project is currently investigating type inference based methods as an alternative to data-flow methods. Several results for performing points-to analyses (or alias analyses) by type inference methods has been achieved. The research is still under active investigation. However, *there are not publishable results yet*.

RENAISSANCE project The RENAISSANCE project at Lancaster University is an ESPRIT funded research project into software reengineering and software evolution. The principle business objectives of the RENAISSANCE partners are to improve

their capability to offer commercial services in the area of system evolution and to increase their return on investment in their software assets. To meet these objectives, the RENAISSANCE project has established the following technical objective:

- Support application evolution from centralised to distributed client-server architectures.
- Support the recovery of system family designs and subsequent evolution using existing CASE tools.
- Support evolution through the reuse of sub-systems recycled from existing systems.
- Provide a method for project managers to assess the costs, risks and benefits of evolution options.
- Integrate all of this support into a systematic method to support system evolution.

An integrated RENAISSANCE evolution method is proposed to guide the process of system evolution. This will be distinguished from other reengineering projects by its focus on architectural evolution and the recovery of designs of system families in 4GLs rather than the more common COBOL or FORTRAN.

Grasp project [54, 56] The development of GRASP has been supported by research grants from NASA Marshall Space Flight Centre, the Department of Defence Advanced Research Projects Agency (ARPA) and the Defence Information Systems Agency (DISA). The GRASP Project has successfully created and prototyped a new algorithmic level graphical representation for Ada software: the Control Structure Diagram (CSD). The primary impetus for creation of the CSD was to improve the comprehension efficiency of Ada source code and, as a result, improve software reliability and reduce software costs.

GRASP provides the capability to generate CSDs from Ada 95 source code in both a reverse and forward engineering mode with a level of flexibility suitable for professional application. As of release 4.3, GRASP has been integrated with GNAT, GNU's Ada 95 compiler. This has resulted in a comprehensive graphical based development environment for Ada 95. The user may view, edit, and print, and compile source code as CSDs with no discernible addition to storage or computational overhead.

TAMPR project TAMPR is a transformation system developed by the Software Reengineering Group at Queen's University, UK. The objective of the Reverse Engineering tool developed at Queen's University is to translate COBOL into a structured notation, called Standard Form. The transformations used to achieve that translation are applied by the TAMPR transformation system. Each transformation is a rewrite rule, consisting of a pattern and a replacement defined using a wide-spectrum grammar.

CORET and ARES Both the projects belong to the reuse and reverse engineering group at the Vienna University of Technology, Austria.

- ESPRIT IV Project: ARES [61, 13]. The Architectural Reasoning for Embedded Systems (ARES) project enables software developers to explicitly describe, assess, and manage architectures of embedded software families. To reach this goal they select, extend or develop a framework of methods, processes and prototype tools for incorporating architectural reasoning along the life-cycle of embedded software families. Results of this project will help to design reliable systems with embedded software, that satisfy important quality requirements, evolve gracefully and may be built in-time and on-budget. Partners are Nokia, Philips, ABB, Imperial College, and Technical University of Madrid.

- FWF Project: CORET [69, 70]. The FWF-funded project Object-Oriented Reverse Engineering (CORET) aims at transforming old data processing software systems to a modern, object-oriented architecture. It focuses on guiding such a

transformation process by different kinds of patterns on different levels of abstraction thereby integrating human expertise in order to overcome typical limits of automated reverse engineering methods. The feasibility and the effectiveness of the approach will be evaluated by building a prototype toolset.

The TXL language work TXL developed at Queens University, Canada is a programming language and rapid prototyping system specifically designed to support transformational programming, . The basic paradigm of TXL involves transforming input to output using a set of structural transformation rules that describe by example how different parts of the input are to be changed into output. Each TXL program defines its own context free grammar according to which the input is to be structured, and rules are constrained to preserve grammatical structure in order to guarantee a well-formed result.

3.6 Sources Retrieved

- The following journals of most recent six years (January 1993 to January 1999) are searched manually:

    1. IEEE Transaction on Software Engineering
    2. IEEE Software
    3. ACM Transaction on Software Engineering and Methodology
    4. ACM Software Engineering Notes
    5. Software Maintenance: Research and Practice

- BIDS database, including:

    1. Science Citation Index(SCI), which includes all the international and important national journals of most recent six years.

2. Index to Scientific and Technical Proceedings, which includes all the proceedings of international conferences of the most recent five years.

- Internet. Internet is searched thoroughly through the following agents:

1. Excite
2. Infoseek
3. Lycos
4. Yahoo

3.7 Conclusion

Although significant work has been carried out on many aspects of reverse engineering, using formal abstraction rules to extract formal specifications from source code is rarely addressed, especially in real-time domain. Maintainer's Assistant [163, 160], PRISME [17, 16], AUTOSPEC [42, 43, 44, 74, 72, 73] and the work of the Software Engineering Laboratory, University of California at San Diego [91] solved some closely-related problems, such as transformation and part of informal abstraction. However, none of them engages in extracting semantics-oriented formal specifications from source code through abstraction. Formal abstraction rules for reverse engineering have never been developed.

Chapter 4

An Integrated Framework for Reengineering

4.1 Characteristics of Legacy Systems

4.1.1 Typical Problems

Legacy systems present a fundamental challenge to those who own and operate them: those systems have begun to age but continue to provide vital services [141, 148]. They were designed to follow requirements and an implementation approach that existed earlier in the organisation's life cycle. Then they were released into environments possibly different from those planned or changed significantly over years. Presently, though years and decades later, they are still expected to operate efficiently, solve problems, and incorporate changes in technology and business practices for many years to come [2].

Because legacy software systems are so critical to an organisation's survival, they are not retired or substituted with newly developed systems without compelling reasons. Major changes require huge investment in new technology, with significant risk that the new systems may fail to deliver the required services. Therefore, organisations maintain functionality, correct defects, and upgrade legacy systems to keep up with

changing business or technical conditions.

Legacy systems share many negative characteristics, or in another word, problems. Some of the worst, and lamentably typical ones are as follows:

- Legacy systems are large, with hundred thousands or even millions of lines of codes.
- They are geriatric, often more than ten years old.
- They are written in a legacy language like COBOL.
- They are built around a legacy environment, e.g., IBM's IMS (a DBMS from IBM).
- They are autonomous. Applications operate independently, with little or no interface with other applications. If interfaces are present, they are often badly designed, haphazard at best according to present criteria. For example, some interfaces were based on export/import models or lack of data consistency.

To complicate matters, these legacy systems are often *mission-critical*, i.e., essential to the organisation's business and must be operational at all time.

4.1.2 Structure and Data Dependency

A legacy system is, under most circumstances, composed of nested procedures and functions. In what follows, we use the term 'component' to mean **procedure** or **function**. If the system is monolithic, then we apply various re-structuring techniques. According to the nested structure, these components have different *visibility* (scope) levels. The components which nest at the top layer (i.e., components with no parent, such as *main*() in C programs) are assumed to have the highest visibility level 0. This means that those components are in a most general position in the whole system. Similarly, the direct sub-procedures and sub-functions of a level 0 component have the visibility level 1. And for a component of level i, the visibility level of its direct sub-procedures

and sub-functions is level $i + 1$. A depiction of visibility levels is given in Figure 4.1. All data items are associated with the same component's visibility (scoping) level at which they were first declared. Therefore, the top global components and their data items have the highest visibility level which is marked *level 0*, and those at the *n*th nested layer have the *n-1* visibility level. In ideal cases, a component at level i only has direct access to components at level $i + 1$. Components distributed over several levels can be treated as a system composed of sub-components at different levels.

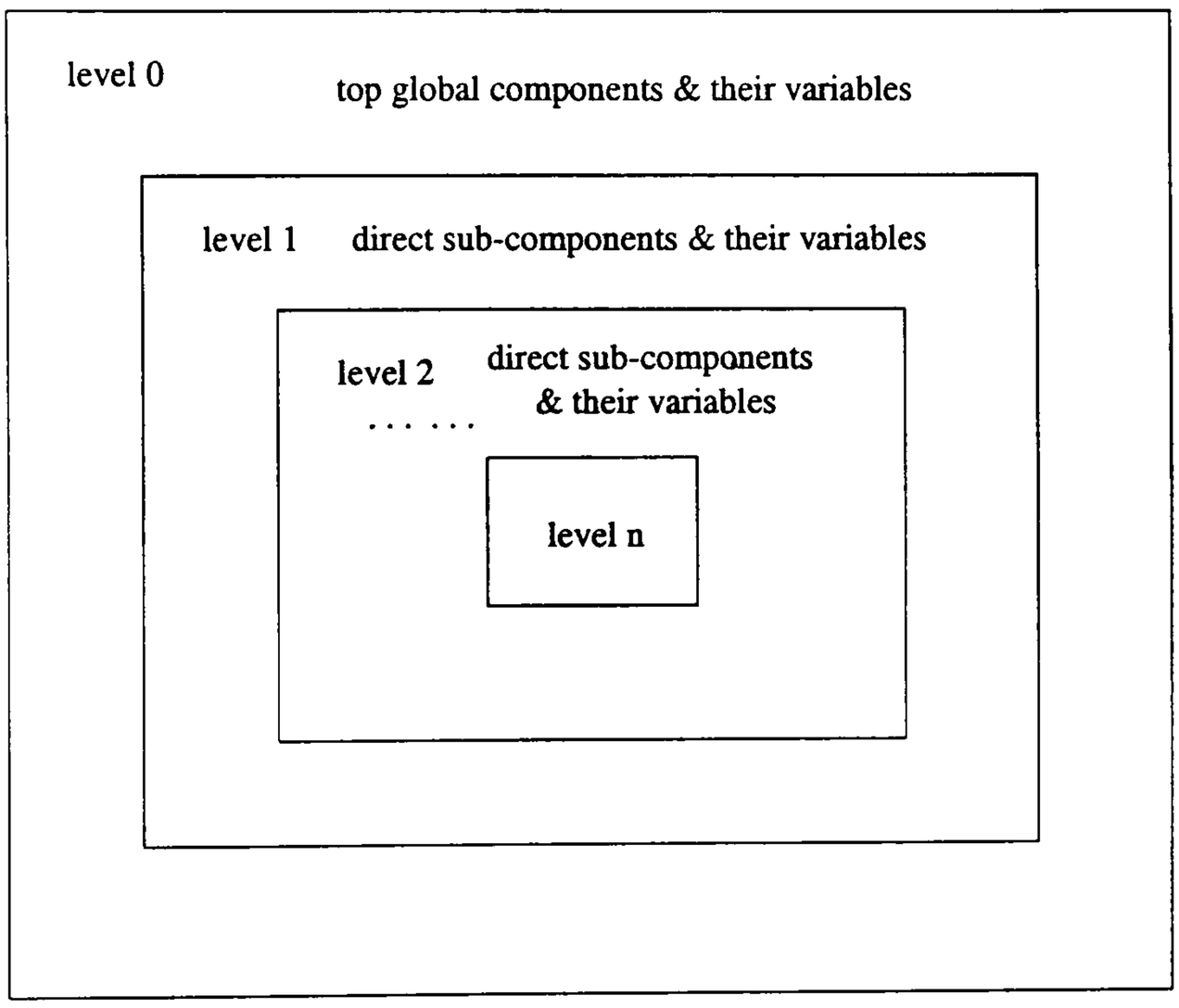

Figure 4.1: Visibility Levels of System Components and Their Data Items.

The functional effect of a component can be interpreted as changing part or all of the data items at a higher level than this component (i.e., *relative global variables* to this component) into new values with the aid of data items belonging to this component (i.e., this component's *local variables*). Therefore, a component can be viewed as a mapping function between the new and the original values of its relative global data items. The effect of a component is embodied in the change of its relative global variables. Figure 4.2 shows this mapping relation. The rectangle with rounded corners represents a component, the circles represent local variables of the component, and the rectangles represents global variables to the components. Variables at the left side

are the original states and those at the right are the new states after invocation of the component.

For an object-oriented system, the data fields of an object and those accessible to the object can be considered as global data items, and other data items used by its methods are local ones of the object.

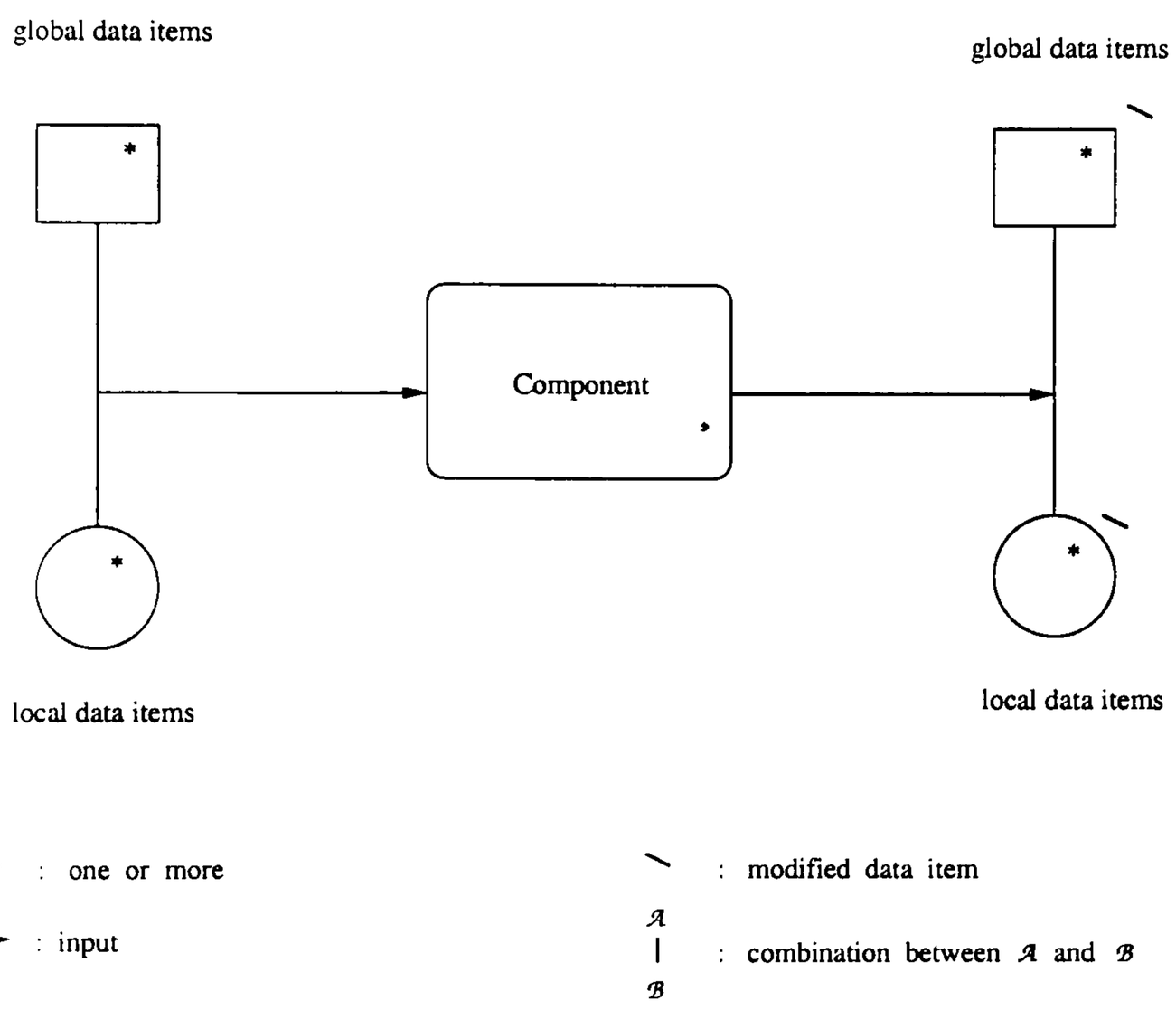

Figure 4.2: Functional Mapping of a System Component.

Based on the above concepts of visibility level and mapping function, Figure 4.3 shows the typical structure and data dependency of a legacy system. Here, *primary data items* are the global data items with visibility level 0, and *secondary data items* are those data items whose visibility level is deeper than 0. The primary data items at the top are initial states of the legacy system, and those at the bottom are the final states. Those in the middle are intermediate states. The nested rectangles are the components at various visibility levels in the legacy system.

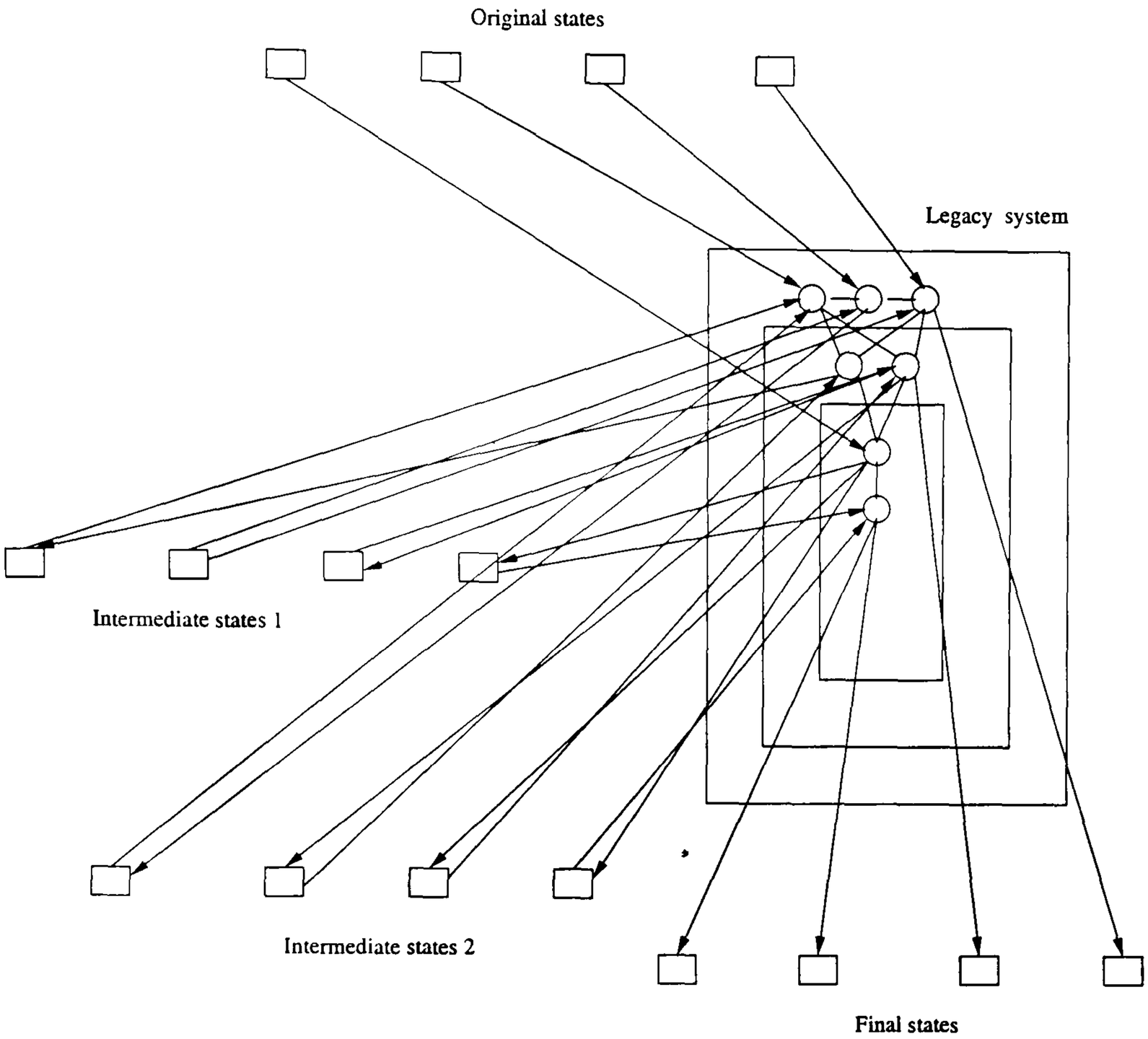

Figure 4.3: Global and Local Data Dependency.

4.2 The Approach

4.2.1 General Method

Using Wide Spectrum Language

The study in previous chapters has shown [110] that using a wide spectrum language is the most suitable and efficient approach to the reengineering of computing systems because of its various abstraction levels and the integrity of these levels.

Based on the characteristics of legacy systems, a unified approach for software reengineering is proposed. The approach is based on the construction of a wide spectrum language, known as RWSL, which enjoys a sound formal semantics. An integrated

framework for reengineering is built to support the proposed approach.

As the reverse engineering part, we endeavour to extract a formal specification from the legacy source code. There are several reasons to support this idea:

- Specification is more compact than source code, it is expressed in a more problem-oriented notation and is easier for the software engineer to understand. Therefore the extracted specification could greatly facilitate software engineers' understanding of the legacy system, both in efficiency and accuracy, and therefore facilitate further re-design and re-specification of the original system. The benefit is worth the cost, especially for critical legacy systems.

- From the new specification, executable code can potentially be generated automatically or semi-automatically. Using formal notations could assure more precise system description and increase the automation of the whole reengineering process.

In this section, we discuss the architecture and working flow of RWSL— the main points of our approach are object extraction rules and abstraction rules [109, 106, 108, 107, 168, 111]. Object extraction rules deal with the transformation of legacy systems at code level to object-oriented systems. And abstraction rules help to extract system specifications from the code.

Using Abstraction

Central to our approach is the notion of **abstraction** and its **rules** ('information' hiding; a precise definition of which will be given in chapter 6). In our approach, abstraction is performed *systematically* at both data and structural levels in such a way that the underlying computation is not disturbed whilst functional abstraction is also performed, that is, some functionality considered "trivial" is abstracted away from the code.

The approach firstly identifies all data items and their 'visibility' levels, where visibility level 0 is the highest. Thereafter it makes the subject system more abstract by removing some data items (those of visibility level > 0) whilst expressing their contribution to the overall functional behaviour of the system with the remaining data items. Such a contribution will be expressed (encoded) within the specification statement of RWSL, which is in ITL formulae (see next chapter for detail).

The approach therefore can be described as follows:

1. Identify all components in a system. There is an obvious correlation between the structure of the legacy code and the structure of the resulting formal specification. The more structured the formal specification is, the easier it is to understand, to improve, and to be used as an appropriate starting point for forward engineering. If the system is very monolithic or unstructured, then engage existing restructuring techniques [20, 78, 100] to decompose the system into subsystems and structure them. For example, [78] proposed a meaning-preserving program restructuring tool.

2. Associate 'visibility' levels for each component (e.g., the ith component has level l_i). These levels reflect the nesting structure in the system, see Figure 4.1.

3. All data-items are associated with the same component's visibility level at which they were first declared.

4. Identify the central data structure and items of the system (i.e. those with level l_0).

5. For each ith-level component and $i > 0$ do

   (a) Identify all data items local to the component

   (b) Record the effect of the data item, identified in step (a), on any data items in levels Q with $Q < i$, in a specification statement of RWSL, introduce a procedure definition if necessary. Elementary abstraction rules are mostly

used, and the procedure name should reflect the functionality of the procedure as exact as possible. *Avoid* introducing new procedures whenever possible.

(c) Abstract away unnecessary implementation details and trivial functionality description within the generated specification. This will be done with corresponding further abstraction rules.

The *correctness* is achieved through the soundness of the applied abstraction rules.

Abstraction Pattern

Abstraction is a process of generalisation, removing restrictions, eliminating detail and removing nonessential information [158]. Unlike transformation which keeps the semantics unchanged, abstraction endeavours in weakening the original semantics of system implementation. Thus the abstractions cannot be applied without a clear idea of which information contained in the program refers simply to the implementation, and not to the function of the program. In general case, this information cannot be determined automatically within the system, so user guidance is needed at this stage.

To solve this problem, a set of abstraction patterns are proposed based on the developed further abstraction rules as an efficient means to let the software reengineer inform the computer system about his/her observations of the legacy system. And then the computer system will perform abstraction with the aid of these observations and the relevant abstraction rules. These abstraction patterns appear in RWSL and the supporting tool as *abstraction pattern assertions*. Details are described in Chapter 6 and Chapter 7.

4.2.2 Architecture of RWSL

RWSL is a multi-layered wide spectrum language with sound formal semantics. Due to the distinct advantage of Interval Temporal Logic (ITL) [124, 125, 39, 38, 170], we

use it as the semantic foundation of RWSL.

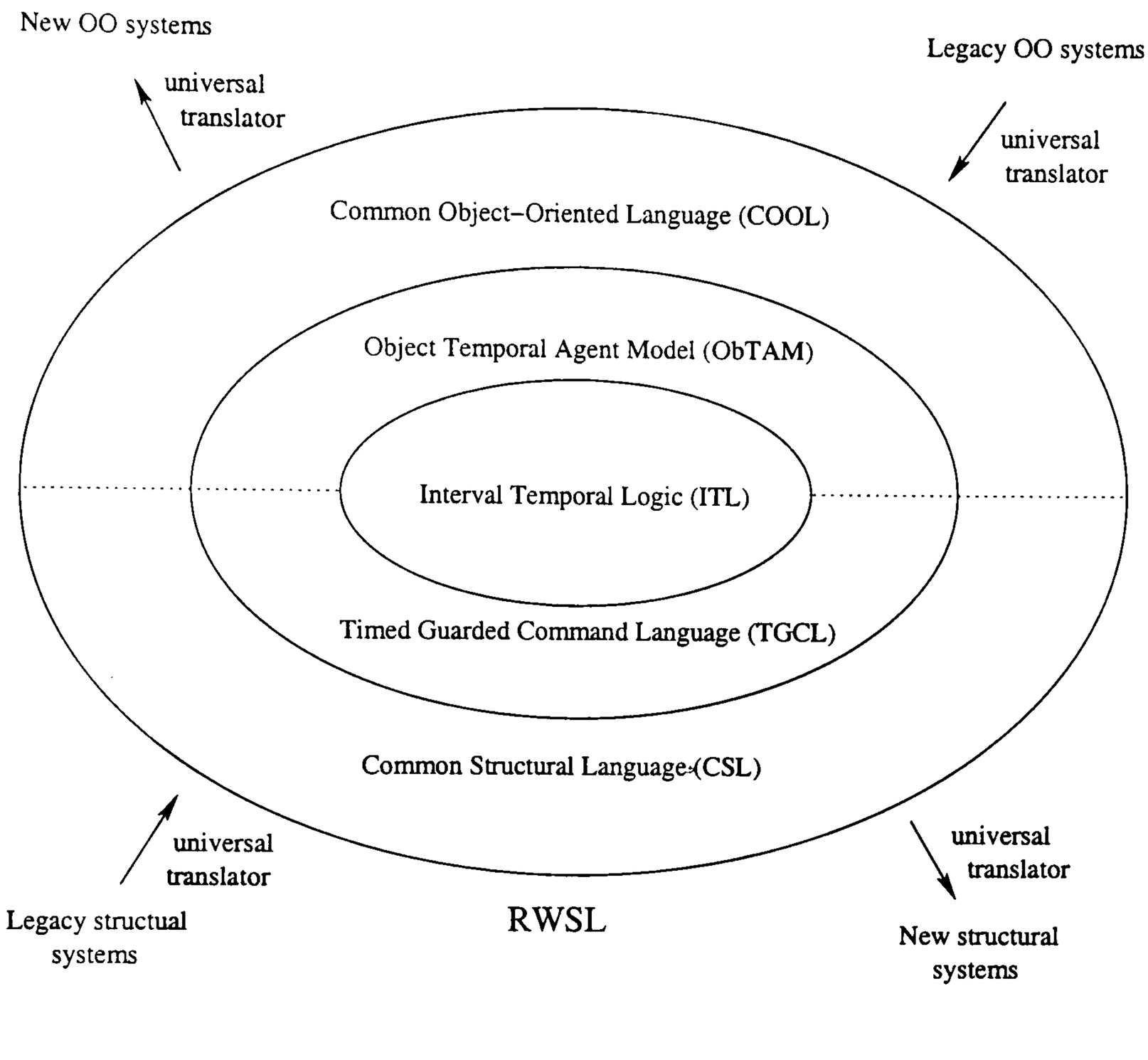

Figure 4.4: RWSL: General Architecture.

Figure 4.4 shows the architecture of RWSL. The top part is the object-oriented section, which includes three layers, namely ITL Specification, Object-Oriented Temporal Agent Model (ObTAM) and Common Object-Oriented Language (COOL). ObTAM is an extension of Temporal Agent Model (TAM) language [144, 142, 145] with object-oriented features. The most concrete layer of the object-oriented section is Common Object-Oriented Language, which provides structures as those in an ordinary OO language.

The bottom part is the structural (procedural) section, which also includes three layers: ITL Specification, Timed Guarded Command Language (TGCL) and Common Structural Language (CSL). TGCL is an extension of Dijkstra's Guarded Command Language [58, 59] with time and concurrency feature. Both TGCL and CSL are at the code level, while in CSL operators and concepts are implemented in common program-

ming elements, such as shunts.

Both the object-oriented and procedural systems will be specified with ITL formulae. The semantics of other layers of RWSL, together with the abstraction and object extraction rules will be defined in ITL.

4.2.3 Working Flow of RWSL

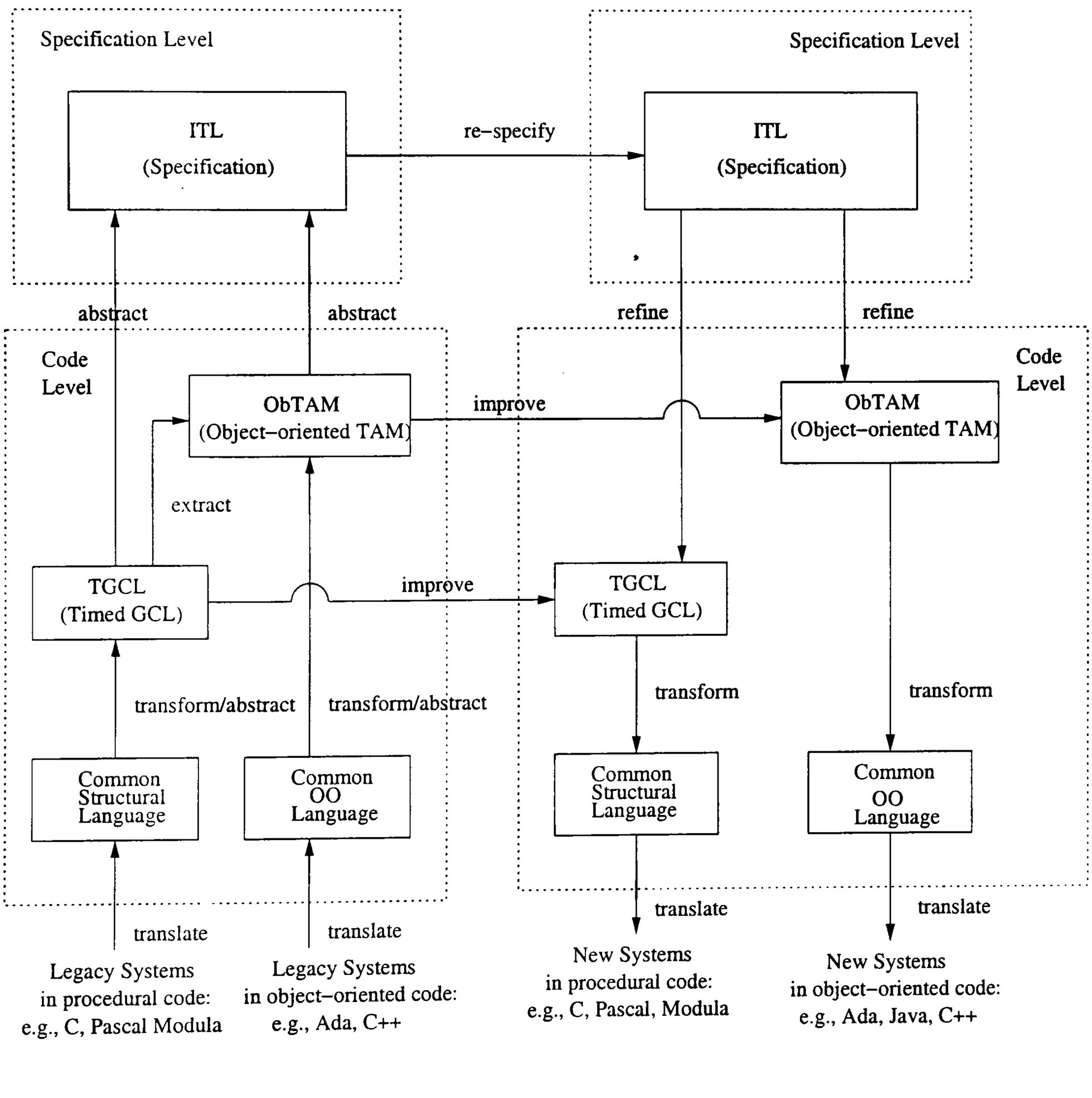

Figure 4.5: RWSL: Working Process in Reengineering.

Figure 4.5 shows the possible process when using RWSL to reengineer legacy systems. The approach may be used as follows: the source code of a procedural or object-

oriented legacy system is first translated into CSL or COOL through a translator [1]. Such a translation ensures standardisation, since legacy systems may have come in various languages, such as C, C++, Modula or COBOL. This is followed by transformation to TGCL or ObTAM through successive application of correctness-preserving transformation rules.

There are three possible paths for reengineering (Figure 4.5):

1. TGCL and ObTAM code can be improved/extended by adding the required extra functionalities. The TGCL and ObTAM code can be then transformed into an equivalent programming language (either through transformation or straight forward translation). In this path, the procedural nature of a procedural legacy system is kept.

2. If the object-oriented paradigm is sought, object extraction is performed to obtain an equivalent ObTAM code from the procedural TGCL code. Then the ObTAM code is extended/improved. Subsequently, this is transformed to an object-oriented language, such as ADA, JAVA and C++.

3. If high level of abstract specification is needed, then following the construction of TGCL code or/and ObTAM code, the semantics calculation is performed to produce an ITL specification. The reason for this step is that specification is more compact than code and is expressed in a more problem-oriented notation, and therefore easier for software engineers to understand. The benefit is worth the cost, especially for critical legacy systems. The specification will be subsequently used as a basis for forward engineering through refinement.

ITL Specification is abstract enough for the software maintainer to do re-design and re-specification of the target system. Therefore, at the specification level improvements

[1]The "universal translator", as shown in Figure 4.4, translates between a source/target language to/from RWSL (e.g., a COBOL-to-RWSL Translator [101]). This translator must be written for each source/target language and is simply a one-to-one mapping, to ensure semantics equivalence.

(e.g., addition of new functions/services) will be introduced to make the legacy system more 'suitable' for the new requirements. After these improvements, forward engineering can be carried out, i.e., using refinement rules to refine the new target system into a new concrete form, for example, in ADA.

Chapter 5

Reengineering Wide Spectrum Language

5.1 Interval Temporal Logic

ITL forms the most abstract and logical layer in our language. It is used to give a specification oriented semantics for TGCL and ObTAM. Furthermore, all transformation, object extraction, abstract and refinement relations and rules are precisely described and proved within ITL. The choice of ITL is based on a number of reasons. It is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offer powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulae in a slightly modified version of ITL [38]. Tempura [125], an executable subset of ITL, provides an executable framework for developing, analysing and experimenting with suitable ITL specifications.

In addition, Zedan and Cau have provided a refinement calculus for ITL [38], which

takes ITL to calculate a refined concrete portion in Tempura.

5.1.1 Syntax

An interval σ is considered to be a (in)finite sequence of states $\sigma_0\sigma_1...$, where a state σ_i is a mapping from the set of variables *Var* to the set of values *Val*. The length $\mid \sigma \mid$ of an interval $\sigma_0...\sigma_n$ is equal to n (one less than the number of states in the interval, i.e., a one state interval has length 0).

ITL syntax is defined as follows, where i is a constant; a is a static variable (doesn't change within an interval); A is a state variable (can change within an interval); v a static or state variable; g is a function symbol and p is a predicate symbol.

Expressions:

$$exp ::= i \mid a \mid A \mid g(exp_1, ..., exp_n) \mid \iota a : f$$

Formulae:

$$f ::= p(exp_1, ..., exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1;\ f_2 \mid f^*$$

The informal semantics of the most interesting constructs are as following:

- $\iota a : f$: the value of a such that f holds. If there is no such an a then $\iota a : f$ take an arbitrary value from a's range.
- $\forall v \bullet f$: for all v such that f holds.
- skip: unit interval(length 1).
- $f_1;\ f_2$: holds if the interval can be decomposed("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

Frequently used abbreviations are listed in table 5.1.

Table 5.1: Frequently used abbreviations

$\bigcirc f$	$\widehat{=}$ skip ; f	next
more	$\widehat{=} \bigcirc true$	non-empty interval
empty	$\widehat{=} \neg more$	empty interval
inf	$\widehat{=}$ *true* ; *false*	infinite interval
isinf (f)	$\widehat{=}$ *inf* $\wedge f$	is infinite
finite	$\widehat{=} \neg inf$	finite interval
isfin (f)	$\widehat{=}$ *finite* $\wedge f$	is finite
$\Diamond f$	$\widehat{=}$ *finite* ; f	sometimes
$\Box f$	$\widehat{=} \neg\Diamond\neg f$	always
$\textcircled{w} f$	$\widehat{=} \neg \bigcirc \neg f$	weak next
$\Diamond\!\!\!\!\cdot f$	$\widehat{=} f$; *true*	some initial subinterval
$\boxdot f$	$\widehat{=} \neg(\Diamond\!\!\!\!\cdot \neg f)$	all initial subintervals
$\Diamond\!\!\!\!\cdot\!\!\!\!\cdot f$	$\widehat{=}$ *finite* ; f ; *true*	some subinterval
$\boxdot\!\!\!\!\cdot f$	$\widehat{=} \neg(\Diamond\!\!\!\!\cdot\!\!\!\!\cdot \neg f)$	all subintervals
$\Diamond^+ f$	$\widehat{=} \Diamond(more \wedge f)$	
$\boxdot^+ f$	$\widehat{=} \neg(\Diamond^+ \neg f)$	
halt f	$\widehat{=} \Box$(empty $\equiv f$)	terminate interval when
keep f	$\widehat{=} \boxdot$(skip $\Rightarrow f$)	all unit subintervals
f^ω	$\widehat{=}$ *isinf* $(isfin\ (f)^*)$	infinite chopstar
$\bigcirc exp$	$\widehat{=} \imath a: \bigcirc(exp = a)$	next value
fin exp	$\widehat{=} \imath a: fin\ (exp = a)$	end value
$A := exp$	$\widehat{=} \bigcirc A = exp$	assignment
stable *exp*	$\widehat{=}$ *exp gets exp*	stability

5.1.2 Semantics

The formal semantics is as followings: Let $\mathcal{X}$ be a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mapping for the variable v.

- $\mathcal{E}_\sigma[\![v]\!] = \sigma_0(v)$
- $\mathcal{E}_\sigma[\![g(exp_1, ...exp_n)]\!] = \hat{g}\ (\mathcal{E}_\sigma[\![exp_1]\!], ..., \mathcal{E}_\sigma[\![exp_n]\!])$
- $\mathcal{E}_\sigma[\![\imath a : f]\!] = \begin{cases} \mathcal{X}(u) & \text{if } u \neq \{\} \\ \mathcal{X}(\mathrm{Val}_a) \text{ otherwise} \end{cases}$
  where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \wedge \mathcal{M}_\sigma[\![f]\!] = \mathrm{tt}\}$

- $\mathcal{M}_\sigma[\![p(exp_1, ..., exp_n)]\!] = \text{tt}$ iff $\hat{p}\ (\mathcal{E}_\sigma[\![exp1]\!], ..., \mathcal{E}_\sigma[\![exp_n]\!])$
- $\mathcal{M}_\sigma[\![\neg f]\!] = \text{tt}$ iff $\mathcal{M}_\sigma[\![f]\!] = \text{ff}$
- $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] = \text{tt}$ iff $\mathcal{M}_\sigma[\![f_1]\!] = \text{tt}$ *and* $\mathcal{M}_\sigma[\![f_2]\!] = \text{tt}$
- $\mathcal{M}_\sigma[\![\forall\, v \bullet f]\!] = \text{tt}$ iff for all σ' s.t. $\sigma \sim_v \sigma', \mathcal{M}'_\sigma[\![f]\!] = \text{tt}$
- $\mathcal{M}_\sigma[\![\text{skip}]\!] = \text{tt}$ iff $\mid \sigma \mid = 1$
- $\mathcal{M}_\sigma[\![f_1; f_2]\!] = \text{tt}$ iff

  (exists a k, s.t. $\mathcal{M}_{\sigma_0 ... \sigma_k}[\![f_1]\!] = \text{tt}$ and

  $((\sigma$ is infinite and $\mathcal{M}_{\sigma_k ...}[\![f_2]\!] = \text{tt})$ or

  $(\sigma$ is finite and $k \leq \mid \sigma \mid$ and $\mathcal{M}_{\sigma_k ... \sigma_{\mid\sigma\mid}}[\![f_2]\!] = \text{tt})))$

  or $(\sigma$ is infinite and $\mathcal{M}_\sigma[\![f_1]\!])$
- $\mathcal{M}_\sigma[\![f^*]\!] = \text{tt}$ iff

  if σ is infinite then

  (exist $l_0, ..., l_n$, s.t. $l_0 = 0$ and

  $\mathcal{M}_{\sigma_{l_n} ...}[\![f]\!] = \text{tt}$ and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$)

  or

  (exists an infinite number of l_i s.t. l_0 and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$)

  else

  (exist $l_0, ..., l_n$ s.t. $l_0 = 0$ and $l_n = \mid \sigma \mid$ and

  for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, ..., \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$)

5.1.3 Specification

Let W be a set of state variables then $frame(W)$ denotes that only the variables in W can possibly change, i.e., the variables outside the frame do not change. The semantics is defined as follows:

– $\mathcal{M}_\sigma\ [\![frame(W)]\!] = \mathrm{tt}$ iff for all $v \in Var - W, \mathcal{M}_\sigma [\![\textbf{stable}(v)]\!]$

The syntax of specification statement is $W : f$ where W is a set of variables and f an ITL formula. The specification statement represents a blackbox description of the behaviour of the required system. When we specify agents that require a minimum execution interval, care must be taken as regard to the feasibility of the specification. This is to ensure that the written specification indeed conforms with whatever restricted computational (executable) model chosen.

The semantics of the specification statement is simply given as

$$W : f \triangleq frame(W) \wedge f$$

5.2 Timed Guarded Command Language

Based on the basic structures of Dijkstra's Guarded Command Language [58, 58], Timed Guarded Command Language introduces time, concurrency, and communication. This gives TGCL the necessary power for tackling time critical concurrency systems.

5.2.1 Syntax

Let $\mathcal{A}$ denotes a TGCL program, x denotes a variable, e denotes an expression, then the syntax of TGCL is as the following:

$\mathcal{A}$::=$x := e$

| $\mathcal{A}$;$\mathcal{A}'$

| $if \underset{i \in I}{\square} g_i\ then\ \mathcal{A}_i\ fi$

| $while\ g\ do\ \mathcal{A}'\ od$

| $(x, y) \leftarrow s$

| $x \rightarrow s$

$$
\begin{aligned}
&| \; T = \{x_i : T_i\} \\
&| \; x : T \\
&| \; proc\ P(In\ pin_i : T_i,\ Out\ pout_j : T'_j)\{\mathcal{A}'\} \\
&| \; P(In\ e_i,\ Out\ x_j) \\
&| \; parbegin\ \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel, ..., \parallel \mathcal{A}_n\ parend \\
&| \; [t]\mathcal{A}' \\
&| \; \mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2 \\
&| \; delay\ n \\
&| \; skip
\end{aligned}
$$

A TGCL variable can be the following:

$$v ::= v_{sig} \mid v_{str} \mid x.d$$

where v_{sig} is an atomic variable, v_{str} is a structural variable, and $x.d$ is a data field of a structural variable.

TGCL also adopts the concept of 'shunt' in TAM. *Shunts* are shared variables via which communications between agents is performed. In TGCL, a TAM agent is implemented with an executable program segment. A shunt contains two values: the first one is a stamp which records the time of the most recent write, and the second one is the value which was most recently written.

The informal semantics of TGCL is described as following:

- General elements.
  - $x := e$ evaluates the result of expression e to variable x.
  - $\mathcal{A}\ ;\mathcal{A}'$ means the sequential composition of $\mathcal{A}$ and $\mathcal{A}'$.
  - *if* $\square_{i \in I} g_i$ *then* $\mathcal{A}_i$ *fi* is a conditional statement. If any guard g_i is true then the corresponding $\mathcal{A}_i$ will be executed.
  - *while g do* $\mathcal{A}'$ *od* is the loop statement.
  - *skip* is empty operation statement.

- Procedural elements.
  - $T = \{x_i : T_i\}$ is the structure building declaration. It defines a structure named T, which has data fields x_i of type T_i, $i \in 1..n$.
  - $x : T$ means defining x as a variable of type T. T can be a simple data type or a structure.
  - $proc\ P(In\ pin_i : T_i,\ Out\ pout_j : T'_j)\{\mathcal{A}'\}$ defines a procedure in TGCL. The procedure is named P, which has pin_i as its input parameters, and $pout_j$ as its output parameters. The input parameter passing convention is *call by value*, which means that the values of the practical parameters are passed into the procedure; and the output parameter passing convention is *call by reference*, which means that the address references of the practical parameters are passed into the procedure, and therefore any change made will take effect on practical parameters themselves. $\{\mathcal{A}'\}$ is the procedure body of P.
  - $P(In\ e_i,\ Out\ x_j)$ means the invocation of procedure P with parameters p_i, while e_i are input parameters and x_j are output parameters.
  - $x.d$ is field selection. x is a structure, and d is a field of x.
- Real time elements.
  - $parbegin\ \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel, ..., \parallel \mathcal{A}_n\ parend$. Here '$\parallel$' is introduced as the parallel operator. This statement means that $\mathcal{A}_1, ..., \mathcal{A}_n$ execute concurrently, and terminates until all the $\mathcal{A}_i$ terminate.
  - $[t]\mathcal{A}'$ means that the execution of $\mathcal{A}'$ should be completed within t time units (deadline).
  - $\mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2$. The given shunt s is treated as a signal, and is monitored from the release time for t time units. If s is written to in that interval then the agent $\mathcal{A}_2$ is released with a release time equal to the end of the interval, otherwise the agent $\mathcal{A}_1$ is released at the end of the interval.

- *delay n* will cause a delay of the system for n time units.
- $(x, y) \leftarrow s$ is the input statement with time feature, which reads the timestamp and value from a shunt s at the same time. The timestamp is read into x, and the value into y.
- $x \rightarrow s$ is the output statement with time feature, which writes the value given into shunt s.

5.2.2 Semantics

Assuming that $\mathcal{P}[\![\mathcal{A}]\!]$ defines the semantics of $\mathcal{A}$, Δt represents a duration of t time unit, i.e., $\mathcal{P}[\![\Delta t]\!] \triangleq len = t$, the formal semantics of TGCL is defined as following:

- $\mathcal{P}[\![x := e]\!] \triangleq \bigcirc x = e$
- $\mathcal{P}[\![\mathcal{A}\,;\,\mathcal{A}']\!] \triangleq \mathcal{P}[\![\mathcal{A}]\!]\,;\,\mathcal{P}[\![\mathcal{A}']\!]$
- $\mathcal{P}[\![\mathit{if}\ \Box_{i\in I} g_i\ \mathit{then}\ \mathcal{A}_i\ \mathit{fi}]\!] \triangleq (\bigvee_{i\in I}(g_i \wedge \mathcal{P}[\![\mathcal{A}_i]\!])) \vee (\bigwedge_{i\in I}\neg g_i)$
- $\mathcal{P}[\![\mathit{while}\ g\ \mathit{do}\ \mathcal{A}'\ \mathit{od}]\!] \triangleq (g \wedge \mathcal{P}[\![\mathcal{A}']\!])^* \wedge \mathit{fin}(\neg g)$
- $\mathcal{P}[\![(x, y) \leftarrow s]\!] \triangleq x = \surd s \wedge y = \mathsf{read}(s)$

  where $\mathsf{read}(s) \triangleq \Pi_2(s)$
- $\mathcal{P}[\![x \rightarrow s]\!] \triangleq \mathsf{skip} \wedge \bigcirc s = (\surd s + 1, x)$
- $\mathcal{P}[\![\mathit{proc}\ P(\mathit{In}\ \mathit{pin}_i : T_i,\ \mathit{Out}\ \mathit{pout}_j : T'_j)\{\mathcal{A}'\}]\!] \triangleq \mathcal{P}[\![\mathcal{A}']\!]$

  $\wedge(\forall\, \mathit{pin}_i \bullet \mathit{pin}_i \in W \wedge \mathsf{stable}(\mathit{pin}_i)) \wedge (\forall\, \mathit{pout}_j \bullet \mathit{pout}_j \in W)$

  where $\mathsf{stable}\ \ \mathit{exp} \triangleq \exists\ a \bullet \Box(\mathit{exp} = a)$
- $\mathcal{P}[\![P(\mathit{In}\ e_i,\ \mathit{Out}\ x_j)]\!] \triangleq \mathcal{P}[\![\mathcal{A}']\!](\mathit{pin}_i/e_i, \mathit{pout}_j/x_j)$

  where $\mathit{proc}\ P(\mathit{In}\ \mathit{pin}_i : T_i,\ \mathit{Out}\ \mathit{pout}_j : T'_j)\{\mathcal{A}'\}$
- $\mathcal{P}[\![x : T]\!] \triangleq f_T(x)$, i.e. the feature of type T

- $\mathcal{P}[\![T = \{x_i : T_i\}]\!] \triangleq \forall x \in T \bullet f_T(x) = \bigwedge_{i \in I} f_{T_i}(x_i)$
- Assume $T = \{x_i : T_i\}$ and $x : T$, then $\mathcal{P}[\![x.d]\!] \triangleq d \in \bigcup_{i \in I} x_i$, i.e., d is a data field of structural variable x.
- $\mathcal{P}[\![parbegin\ \mathcal{A} \parallel \mathcal{A}'\ parend]\!] \triangleq \mathcal{P}[\![\mathcal{A}]\!] \wedge \mathcal{P}[\![\mathcal{A}']\!]$
- $\mathcal{P}[\![[t]\mathcal{A}']\!] \triangleq \Delta t \wedge (\mathcal{P}[\![\mathcal{A}']\!] ;\ true) \wedge (\mathcal{P}[\![\mathcal{A}']\!] \supset len <= t)$
- $\mathcal{P}[\![delay\ n]\!] \triangleq len = n$
- $\mathcal{P}[\![\mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2]\!] \triangleq (\Delta t \wedge \mathsf{stable}(\surd s) ;\ \mathcal{P}[\![\mathcal{A}_1]\!]) \vee (\Delta t \wedge \neg\, \mathsf{stable}(\surd s) ;\ \mathcal{P}[\![\mathcal{A}_2]\!])$
- $\mathcal{P}[\![skip]\!] \triangleq \mathsf{empty}$

5.3 Object-Oriented Temporal Agent Model

TAM (Temporal Agent Model) aims to be a realistic software development method for real-time systems. It has sufficient power for time, concurrency and communication. ObTAM extends TAM with object-oriented features, e.g., object hierarchy and inheritance.

5.3.1 Syntax

ObTAM syntax is the same as the syntax of TGCL less the procedural part, but with the following additional object-oriented portion:

$$
\begin{aligned}
\mathcal{A} ::= & x : T \\
& \mid T <_{sub} T' \\
& \mid T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T'_l)[\mathcal{A}_j]\} \\
& \mid x.d \\
& \mid x.m(In\ e_k : T_k,\ Out\ pout_l : T'_l)
\end{aligned}
$$

A variable of ObTAM can be the following:

$$v ::= v_{sig} \mid v_{obj} \mid x.d$$

where v_{sig} is an atomic variable, v_{obj} is an object variable, and $x.d$ is a data field of an object variable.

The informal semantics of ObTAM is described as following:

- General elements: same as TGCL.
- Object-oriented elements.
  - $x : T$ means defining x as a variable of type T. T can be a simple data type or a class.
  - $T <_{sub} T'$ can be used to build the object hierarchy. It declares that class T is a subclass of class T'. As the consequence, T will inherit all the data fields and methods in T' if they are not redefined in T. On the other hand, all the data field and methods in T' will be overridden with the counterparts in T if they are redefined in T.
  - $T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T'_l)[\mathcal{A}_j]\}$ is the class building declaration. It defines a class named T, which has data fields x_i of type T_i, $i \in 1..n$, and methods m_j, $j \in 1..r$. The behaviour of a class is a sequence of method invocations. pin_{j_k} stands for the input parameters of method m_j, and $pout_{j_l}$ stands for the output parameters of method m_j. The input parameter passing convention is *call by value*, and the output parameter passing convention is *call by reference*. $\mathcal{A}_j$ is the methods body of method m_j.
  - $x.d$ is object field reference. x is an object, and d is a field of x.
  - $x.m(In\ e_k : T_k,\ Out\ pout_l : T'_l)$ is method invocation. It invocates the method m in object x.
- Real time elements: same as TGCL.

5.3.2 Semantics

Similarly, the formal semantics of the above elements of ObTAM is defined as follows:

- $\mathcal{P}[\![x : T]\!] \triangleq f_T(x)$, i.e. the feature of type T

- $\mathcal{P}[\![T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T'_l)[\mathcal{A}_j]\}]\!] \triangleq$

  $x : T = W_x : f$ where

  $W_x = \bigcup_{i \in I} x_i$

  $f = \bigwedge_{i \in I} f_{T_i}(x_i) \wedge (\bigvee_{j \in J} (\mathcal{P}[\![\mathcal{A}_j]\!] \wedge (\forall\, pin_{j_k} \bullet stable(pin_{j_k}))))^*$

- Assume $T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T_l)[\mathcal{A}_j]\}$ and $T' = \{y_{i'} : T'_{i'},\ m'_{j'}(In\ pin_{j'_{k'}} : T'_{k'},\ Out\ pout_{j'_{l'}} : T'_{l'})[\mathcal{A}'_{j'}]\}$, then:

  $\mathcal{P}[\![T <_{sub} T']\!] \triangleq x : T = W_x : f$

  where $W = \bigcup_{i \in I} x_i \cup \bigcup_{i' \in I'} y_{i'}$ iff for all $x_i\ i \in I$, $y_{i'} \neq x_i$

  $f = \bigwedge_{i \in I} f_{T_i}(x_i) \wedge \bigwedge_{i' \in I'} f_{T_{i'}}(y_{i'}) \wedge \bigvee_{j \in J} \Phi_j^* \wedge \bigvee_{j' \in J'} (\Phi'_{j'})^*$

  iff for all $x_i\ i \in I$, $y_{i'} \neq x_i$, and iff for all $\Phi_j\ j \in J$, $\Phi'_{j'} \neq \Phi_j$

  $\forall\, j \in J \bullet \Phi_j = \mathcal{P}[\![\mathcal{A}_j]\!] \wedge stable(pin_{j_k})$

  $\forall\, j' \in J' \bullet \Phi'_{j'} = \mathcal{P}[\![\mathcal{A}'_{j'}]\!] \wedge stable(pin_{j'_{k'}})$

The above means that T inherits all the data fields and methods of T' if they are not redefined in T, and all the data fields and methods in T' are overridden with the counterparts in T if they are redefined in T.

- Assume $T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T_l)[\mathcal{A}_j]\}$ and $x : T$, then:

  $\mathcal{P}[\![x.d]\!] \triangleq d \in \bigcup_{i \in I} x_i$, i.e., d is a data field of object x.

- Assume $T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T_l)[\mathcal{A}_j]\}$ and $x : T$, then:

$\mathcal{P}[\![x.m(In\ e_k,\ Out\ y_l)]\!] \triangleq$ there exists an m_s s.t.,

$$m_s \in \bigcup_{j \in J} m_j \wedge m_s = m \wedge \mathcal{P}[\![\mathcal{A}_s]\!] (pin_{s_k}/e_k, pout_{s_l}/y_l)$$

5.4 Common Structural Language

CSL is developed to enrich the statements in TGCL and make RWSL compatible to WSL in MA. Statements in CSL are more program-like. CSL can be viewed as an extension of WSL in MA with time, concurrency and type, or a variation of TGCL with a more program-like format and diversity in statements. CSL is the most concrete procedural layer of RWSL.

CSL syntax and semantics (defined in TGCL) is as follows:

1. Assignment

   x:=e $\triangleq$ $x := e$

   This statement evaluates the result of expression e to variable x.

2. Sequential Composition

   $\mathcal{A}$; $\mathcal{B}$ $\triangleq$ $\mathcal{A}$; $\mathcal{B}$

   This statement compose program segments $\mathcal{A}$ and $\mathcal{A}'$ sequentially.

3. Deterministic Iteration

   while g **do**

   $\mathcal{A}$ $\triangleq$ *while g do* $\mathcal{A}$ *od*

   od

   This statement is an iteration with g as loop condition and $\mathcal{A}$ as loop body.

4. Input Statement

read x, y **from** s $\triangleq$ $(x, y) \leftarrow s$

This is the input statement with time feature, which reads the timestamp and value from a shunt s at the same time. The timestamp is read into x, and the value into y.

5. Output Statement

write x **to** s $\triangleq$ $x \rightarrow s$

This is the output statement with time feature, which writes the value given into shunt s.

6. Structure Definition

struct T

{

T_i: x_i $\triangleq$ $T = \{x_i : T_i\}$

}

This is the structure declaration statement. It defines a structure named T, which has data fields x_i of type T_i, $i \in 1..n$.

7. Typed Variable

T: x $\triangleq$ $x : T$

This statement defines x as a variable of type T. T can be a simple data type or a structure.

8. Procedure Definition

proc P(In pin_i:T_i, Out $pout_j$:T'_j)

{

$\mathcal{A}$ $\triangleq$ *proc* $P(In\ pin_i : T_i,\ Out\ pout_j : T'_j)\{\mathcal{A}'\}$

}

This statement defines a procedure named P, which has pin_i as its input parameters, and $pout_j$ as its output parameters. The input parameter passing convention is *call by value*, and the output parameter passing convention is *call by reference*. $\{\mathcal{A}'\}$ is the procedure body of P.

9. Procedure Invocation

P(In e_i, Out x_j) $\triangleq$ $P(In\ e_i,\ Out\ x_j)$.

This statement invocates procedure P with parameters p_i, while e_i are input parameters and x_j are output parameters.

10. Parallel Composition

parbegin

$\mathcal{A}_1$

parallel with $\triangleq$ *parbegin* $\mathcal{A}_1 \parallel \mathcal{A}_2$ *parend*

$\mathcal{A}_2$

parend

This statement defines that program segments $\mathcal{A}_1$ and $\mathcal{A}_2$ execute in parallel, and terminates until all $\mathcal{A}_i$ terminate.

11. Deadline

duration t **in**

$\mathcal{A}$ $\triangleq$ $[t]\mathcal{A}$

end

This statement means that the execution of $\mathcal{A}$ should be completed within t time units (deadline).

12. Signal

wait **on** s **for** t **do**

$\mathcal{A}$

else $\triangleq \mathcal{A} \trianglerighteq_s^t \mathcal{B}$

$\mathcal{B}$

end

The given shunt s is treated as a signal, and is monitored from the release time for t time units. If s is written to in that interval then the agent $\mathcal{A}_2$ is released with a release time equal to the end of the interval, otherwise the agent $\mathcal{A}_1$ is released at the end of the interval.

13. Delay

delay n $\triangleq$ *delay n*

This statement will cause a delay of the system for n time units.

14. Local Variable

var x:=e:

$\mathcal{A}$ $\triangleq$ $P()$ *where proc* $P()\{x : T;\ x := e;\ \mathcal{A}\}$

end

This statement defines x to be a local variable within block $\mathcal{A}$.

15. Actions

actions : lab_i:

$\text{lab}_{i\in I} == \mathcal{A}_{i\in I}$; **call** $\text{lab}_{j\in I}$ | **call Z**. $\triangleq$ $\mathcal{A}_i\ ;\ (\mathcal{A}_j \vee skip)$

end_actions

This statement defines an *action system* composed of a sequence of *actions*. An action is a parameterless procedure acting on global variables, which is defined as $\text{lab}_{i\in I} == \mathcal{A}_{i\in I}$, where lab_i is a statement variable (the name of the action) and $\mathcal{A}_i$ is the action body. In the above action system, action lab_i is executed first, i.e., execution of its action body. After that, if $\mathcal{A}_i$ is succeeded with a **call** $\text{lab}_{j\in I}$ statement, action lab_j will be executed; if **call Z** is the successive statement the action system exits.

16. Typed Array Declaration

T: **array** ayname[e] $\hat{=}$ $x_1, x_2, ..., x_{e-1}, x_e : T$ where $\text{x}_i \equiv$ ayname[i]

This statement declares that ayname is an array with e elements of type T, where e is an integer expression.

17. Untyped Array Declaration

array name[e] $\hat{=}$ $x_1, x_2, ..., x_{e-1}, x_e$ where $\text{x}_i \equiv$ name[i]

This statement declares that ayname is an array with e elements, where e is an integer expression. The element type is not defined.

18. Assertion

g $\hat{=}$ an ITL formula

Assertions state that a certain condition is true at a particular point in a program.

19. Parallel Assignment

$< \text{x}_1{:=}\text{e}_1, \text{x}_2{:=}\text{e}_2 >$ $\hat{=}$ $x_1 := e_1 \parallel x_2 := e_2$

This statement defines that the assignments inside the brackets are carried out in parallel. x_1 and x_2 must be two different variables.

20. Comment

comment: "text" $\triangleq$ *skip*

This statement presents a comment of the program which is enclosed in the quotation marks.

21. If-Else Conditions

if g_1 **then** $\mathcal{A}_1$		*if* g_1 *then* $\mathcal{A}_1$ *fi*;
else $\mathcal{A}_2$	$\triangleq$	*if* $\neg g_1$ *then* $\mathcal{A}_2$ *fi*
fi		

This is a typical two-branched condition statement.

22. Nested Conditions

if g_1 **then** $\mathcal{A}_1$		
elsf g_2 **then** $\mathcal{A}_2$		*if* g_1 *then* $\mathcal{A}_1$
else $\mathcal{A}_3$	$\triangleq$	*else if* $\neg g_1 \wedge g_2$ *then* $\mathcal{A}_2$ *else* $\mathcal{A}_3$ *fi*
fi		*fi*

This is a typical nested condition statement.

23. Exit

exit $\triangleq$ $\neg g \wedge$ *skip*

where g is the condition of the loop in which **exit** locates.

This statement should locate in a loop, and when it is reached the program leaves the loop and continues execution from immediately after the end of the loop.

24. Infinite Loop

do

$\mathcal{A} \triangleq$ *while true do* $\mathcal{A}$ *od*

od

This statement defines a loop whose iteration condition is alway true.

25. Counted Repetition

for x:=e_1 **to** e_2 **step** e_3 **do**

$\mathcal{A}$ $\triangleq$ for $x \notin Var(\mathcal{A})$ (i.e. for fresh x),

od $x := e_1$; *while* $x \leq e_2$ *do* $\mathcal{A}$; $x := x + e_3$ *od*

This is the standard "for loop" in a programming language.

26. Empty Operation

skip $\triangleq$ *skip*

This statement performs an "empty operation".

27. Dijkstra Constructs

(a) Guarded conditions

d_if

$g_{i \in I} \rightarrow \mathcal{A}_{i \in I} \triangleq$ *if* $\square_{i \in I}$ g_i *then* $\mathcal{A}_i$ *fi*

fi

This is a non-deterministic conditional statement. The program may execute *any* one branch which has a true guard. This contrasts with the ordinary conditional statement where it is the first branch with a true guard that is executed.

(b) Guarded iteration

d_do

$$g_{i\in I} \rightarrow \mathcal{A}_{i\in I} \triangleq \textit{while true do if } \Box_{i\in I}\ g_i \textit{ then } \mathcal{A}_i \textit{ fi od}$$

od

This statement is equivalent to a "D_If" inside a "while" loop. While at least one of the guards is true, the program will execute the loop.

28. Local Procedure Definition

begin

$\mathcal{A}_1$

where proc P(pin$_i$ **var** pout$_j$)==$\mathcal{A}_2$. $\triangleq$ $\mathcal{A}_1 \wedge P \in \mathcal{A}_1$

end

The procedure P is *local* because it belongs to $\mathcal{A}_1$. pin$_i$ are the input parameters which are passed by *value*, and pout$_j$ are output parameters which are passed by *reference*. The procedure body is defined as $\mathcal{A}_2$.

29. Local Procedure Invocation

P(e$_i$ **var** x$_j$) $\triangleq$ $P(In\ e_i,\ Out\ x_j)$

This statement invokes local procedure, while e$_i$ are input parameters and x$_j$ are output parameters.

30. External Procedure Invocation

!xp pname(expn$_i$) $\triangleq$ $P'(In\ e_i,\ Out\ x_j)$

assume that the function of external procedure pname can be simulated by procedure P' in TGCL.

External procedures are used in RWSL in order to deal with code modules which call procedures that are not explicit in the program, i.e., defined outside the system border. The function of the external procedure can be simulated with procedure P'. A (possibly empty) list of expressions is passed by value or by reference to the procedure.

31. Function Definition

f (pin_i: T_i): T

{

$\mathcal{A}$ $\triangleq$ *proc* P_f(*In* $pin_i : T_i$, *Out* $x : T$){$\mathcal{A}$}

}

The statement defines a function named f, the returned value is of type T, and all parameters are passed by value.

32. Untyped Function Definition

f (pin_i)

{

$\mathcal{A}$ $\triangleq$ *proc* P_f(*In* pin_i, *Out* x){$\mathcal{A}$}

}

In this definition, the parameters and the returned value of function f are not typed.

33. Function Invocation

x := f (e_i) $\triangleq$ P_f(*In* e_i, *Out* x) where P_f is the procedure equivalent to function f.

Functions are invoked as expressions wherever an expression can be used.

34. External Function Invocation

x := !f (e$_i$) $\triangleq$ $!P_f(In\ e_i,\ Out\ x)$

where $!P_f$ is the procedure equivalent to function !f.

An external function invocation is an expression which is evaluated by a function which is not explicit in the program. There are obvious similarities between external functions and external procedures.

5.5 Common Object-Oriented Language

The syntax of COOL is the same as the syntax of CSL less the procedural part, but with the following additional object-oriented portion:

1. Class Definition

class T

{

T$_i$: x$_i$;

m$_j$(In pin$_{j_k}$:T$_k$, Out pout$_{j_l}$:T$'_l$)

$\{\mathcal{A}_j\}$

}

$\triangleq T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T'_l)[\mathcal{A}_j]\}$

This statement is the class building declaration. It defines a class named T, which has data fields x$_i$ of type T$_i$, $i \in 1..n$, and methods m$_j$, $j \in 1..r$. pin$_{j_k}$ stands for the input parameters of method m$_j$, and pout$_{j_l}$ stands for the output parameters of method m$_j$. The input parameter passing convention is *call by value*, and the output parameter passing convention is *call by reference*. $\mathcal{A}_j$ is the methods body of method m$_j$.

2. Class Hierarchy

T extends T′ $\triangleq$ $T <_{sub} T'$

This statement is used to build the object hierarchy. It declares that class T is a subclass of class T'. Therefore, T inherits the properties of T'.

3. Field Reference

x.d $\triangleq$ $x.d$

This is object field reference. x is an object, and d is a field of x.

4. Method Invocation

x.m(In e_k, Out y_l) $\triangleq$ $x.m(In\ e_k,\ Out\ y_l)$.

This invokes the method *m* in object *x*.

5. Object Declaration

T : x $\triangleq$ $x : T$

This statement defines x as a variable of type T. If T is a class, x will be an object of class T.

Chapter 6

Abstraction: Taxonomy and Rules

6.1 Introduction

An implementation (code), a design and a specification of a software system are usually at different levels of abstraction. To move from code to design and then to specification involves a process of crossing levels of abstraction. Usually a specification is more abstract than its implementation, and therefore the above process can be also represented as:

$$concrete \rightarrow less\ abstract \rightarrow more\ abstract$$

Abstraction is the crucial technique to reverse engineering. Without tackling abstractions properly, any design or specification recovery methodology can not succeed. To achieve correct and practical abstraction, two fundamental problems need to be solved:

1. First of all, it is necessary to identify *what abstraction is*. Although abstraction technology was used in quite a few research projects [62, 91, 22, 16, 42, 73], the definition of abstraction remains a disputed issue. Most existing definitions adopt *ad hoc* methods and only covers special aspects of the problem. This results in the definitions of abstraction that are ambiguous, incomplete, and incorrect in

some cases. In this chapter, a taxonomy of abstraction is proposed. Within this taxonomy, abstractions are formally defined under different conditions in reverse engineering environment. Monotonicity and relations between these abstractions are discussed and then described in a formal notation. Healthiness obligations are developed as axioms to guarantee correct and sensible abstraction during reverse engineering.

2. Once abstractions are identified in reverse engineering, the next question is how to perform abstraction, i.e., how to cross levels of abstractions. This research issue has not been properly addressed, and practical solutions with precisely defined semantics are urgently needed. To solve this problem, a group of abstraction rules for conducting abstraction in the above process are proposed. These rules aim at extracting formal specification from legacy source code, and are formally defined and proven sound in ITL, which assures precision and correctness.

6.2 Definitions

In a software system, the specification is different from source code in the following aspects:

- Source code has more implementation details which need not to exist in a specification;
- Implementation is focused on *how to do*, while specification is focused on *what to do*;
- There is much more non-determinism in a specification than in an implementation.

In a broad sense, abstraction corresponds to weakening in semantics and this weakening is due to the following:

- inessential design/implementation details are omitted;
- non-determinism is increased; and
- "how to do" is substituted by "what to do".

The simplest interpretation of the notion of abstraction is to hide irrelevant details. Although simple, it leaves open to wider interpretation to what constitutes "irrelevant". For this reason, we have decided to categorise abstraction in a way that hopefully makes it clear. We classify abstraction as follows: *Weakening Abstraction (WA), Hiding Abstraction (HA), Temporal Abstraction (TA), Structural Abstraction (SA)* and *Data Abstraction (DA)*. These five kinds of abstraction form a rather complete taxonomy of abstraction. The formal definition of abstraction is as follows, and special cases will be discussed in the next five subsections.

The implementation of a software system is known as the concrete form of the system, e.g., source code, and the specification is known as the abstract description. To unify terminology, we use the term *representation* for both abstract and concrete forms. Therefore, an abstraction relation $\succeq$ is defined as a function relating two representations of one single system. A representation $\mathcal{B}$ is an abstraction of representation $\mathcal{A}$, written as $\mathcal{A} \succeq_f \mathcal{B}$ (read as $\mathcal{B}$ is an abstraction of $\mathcal{A}$ in respect of f) is defined as:

$$\mathcal{A} \succeq_f \mathcal{B} \triangleq f(\mathcal{A}, \mathcal{B})$$

where f is defined according to the type of abstraction, namely WA(Weakening Abstraction), HA(Hiding Abstraction), TA(Temporal Abstraction), SA(Structural Abstraction) and DA(Data Abstraction).

6.2.1 Weakening Abstraction

Weakening abstraction is quite broad in sense. Here, "weakening" refers to semantics weakening of representations during abstraction. If some information is taken out from

the original representation, and the new result representation has not any contradiction with the original, that is, the semantics of the original representation implies that of the new representation, then a semantics weakening sequence is built and the new representation is a weakening abstraction of the original one.

Corresponding formal definition is as follows:

$$\mathcal{A} \succeq_{WA} \mathcal{B} \triangleq [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$$

The above definition means that representation $\mathcal{B}$ is a weakening abstraction of representation $\mathcal{A}$ on the condition that the semantics of $\mathcal{A}$ implies that of $\mathcal{B}$. Obviously, weakening abstraction is the *inverse of functional refinement*.

Assuming we have a representation segment as follows:

$$\phi = (x \geq 0 \wedge y = x!) \vee (x < 0 \wedge y = \text{-}100)$$

The software reengineer identifies that the last part $(x < 0 \wedge y = \text{-}100)$ is purely implementation detail to assure a smooth execution by exceptional state test and handling, actually y can be arbitrary when $x < 0$, and therefore decides to get rid of it to reach a more concise representation:

$$\phi' = (x \geq 0 \wedge y = x!) \vee x < 0$$

Since $\phi \Rightarrow \phi'$, $(x \geq 0 \wedge y = x!) \vee x < 0$ is a weakening abstraction of the original representation $\phi = (x \geq 0 \wedge y = x!) \vee (x < 0 \wedge y = \text{-}100)$. In further reverse engineering steps, the original representation can even be made more concise by weakening abstraction to:

$$\phi' = (x \geq 0 \Rightarrow y = x!)$$

In fact, the above result is the core function of the original representation.

6.2.2 Hiding Abstraction

Hiding abstraction focuses on the simplification of data space. It emphasises that a part of the data space of the original representation is to be considered as irrelevant or unnecessary and is therefore omitted from the representation. However, the resulting representation should still be a semantic weakening of the original one. In practical reverse engineering, hiding abstraction is often used to get rid of local variables and hide internal communication channels. This is because details become unimportant or too "local" and should not be observed outside the blackbox when a software system is viewed from a more abstract point of view.

The corresponding formal definition is as follows:

$$\mathcal{A} \succeq_{HA} \mathcal{B} \triangleq (\exists x \bullet [\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$$

The above definition means that representation $\mathcal{B}$ is a hiding abstraction of representation $\mathcal{A}$ on the condition that a part of the data space of $\mathcal{A}$ (such as x) is hidden and $\mathcal{B}$ as the remaining part of $\mathcal{A}$ is implied by the original $\mathcal{A}$.

Assuming we have a representation fragment as follows:

```
Queue-body={
    string: field1;
    float: field2;
    }
Queue-body: array queue-body[maximum];
integer:queue-head;
integer:queue-tail;

proc Initialise()
    {
        integer: i;
        queue-head:=0;
        queue-tail:=0;
        for i:=0 to maximum step 1 do
```

```
            queue-body[i].field1:=";
            queue-body[i].field2:=0.0;
        od;
    }
```

The procedure initialises an array-based queue: setting the queue head and tail to position 0 and all queue elements to the nil value. It has a local variable *i* which is used as loop control variable to initialise the queue elements one by one. Obviously, the **for** loop involves implementation details that should not be seen at a higher abstraction level. The software reengineer may record the effect of the **for** loop with an inner procedure "init-elements" and then hide its details by using the "init-elements" instead of the loop in further system representations:

$$\textit{init-elements} \succeq i := 0 \wedge (\textit{queue-body}[i].\textit{field}1 :='' \wedge \textit{queue-body}[i].\textit{field}2 := 0.0 \wedge i := i+1)^{\textit{maximum}}$$

The new representation appears as:

```
Queue-body={
    string: field1;
    float: field2;
    }
Queue-body: array queue-body[maximum];
integer:queue-head;
integer:queue-tail;

proc Initialise()
{
    queue-head:=0;
    queue-tail:=0;
    init-elements();
}
```

Therefore, with the above hiding abstraction local variable *i* is hidden and its related

details are blocked out.

6.2.3 Temporal Abstraction

Temporal abstraction is abstraction which relates to time. It is useful and popular when tackling the reverse engineering of real-time systems. For the representation of a fragment of software systems, namely $\mathcal{A}$, its duration is defined as the time span from the beginning of its execution to the end of its execution. Temporal abstraction reflects the variation of this duration while abstraction is conducted.

Let the duration of $\mathcal{A}$ be denoted $T(\mathcal{A})$. It can be defined as $T(\mathcal{A}) = \{x \in Time \mid \mathcal{A} \wedge len = x\}$, that is, $T(\mathcal{A})$ is the set of execution (durations) times of $\mathcal{A}$. The formal definition of temporal abstraction is as follows:

$$\mathcal{A} \succeq_{TA} \mathcal{B} \triangleq ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]) \wedge R_T(T(\mathcal{A}), T(\mathcal{B}))$$

In the above definition, $R_T(T(\mathcal{A}), T(\mathcal{B}))$ is a relation between the execution times of $\mathcal{A}$ and $\mathcal{B}$. In temporal abstraction, the execution time of the new result representation ($\mathcal{B}$) could be either speeded up or slowed down compared with that of the original representation ($\mathcal{A}$). However, in either cases the new semantics should only be a weakening of the original.

For example, consider an experimental temperature control system, in which there is a heater, an electric fan and a thermo-sensor installed in the experimental box. The heater can be switched on to increase the temperature within the box or switched off to let the box cool off. The electric fan is used to speed up the cooling of the box. The thermo-sensor keeps testing the temperature in the box. If it exceeds the high limit, then the heater is switched off and the fan is switched on. If it exceeds the low limit, then the heater is switched on and the fan is switched off. Assume we have a representation fragment as follows:

$$\textit{temp-control} \succeq (\textit{temp} > \textit{high-limit} \wedge (\Delta 5ms \wedge \textit{heater} = \textit{off} \ ; \ \Delta 5ms \wedge \textit{fan} = \textit{on}))$$

$$\vee(\mathit{temp} < \mathit{low\text{-}limit} \wedge (\Delta 5\mathit{ms} \wedge \mathit{fan} = \mathit{off}\ ;\ \Delta 5\mathit{ms} \wedge \mathit{heater} = \mathit{on}))$$

The software reengineer finds that it is not necessary to switch the heater and the fan sequentially. In this case, the sequential composition operator can be substituted with a parallel composition operator and the duration can be adjusted to the sum of the sequences. Therefore with temporal abstraction the above representation is changed to the follows:

$$\mathit{temp\text{-}control} \succeq (\mathit{temp} > \mathit{high\text{-}limit} \wedge (\Delta 5\mathit{ms} \wedge (\mathit{heater} = \mathit{off} \wedge \mathit{fan} = \mathit{on})))$$
$$\vee(\mathit{temp} < \mathit{low\text{-}limit} \wedge (\Delta 5\mathit{ms} \wedge (\mathit{fan} = \mathit{off} \wedge \mathit{heater} = \mathit{on})))$$

The new representation is weaker in semantics because it only said the heater and fan be switched within 15 milliseconds no matter which one is dealt with first. The operation order in the representation is omitted as trivial detail.

6.2.4 Structural Abstraction

Structural abstraction is so named because it endeavours to make structural simplification in system representation. There are two kinds of composition structures: sequential composition and parallel composition. With structural abstraction, these compositions are reduced and their effects are recorded in a more abstracted representation. Two basic conditions determine whether a change in system representation is a structural abstraction: firstly, whether there is any sequential or parallel composition reduced in the new representation; secondly, whether the semantics of the new representation is a weakening of the original.

Structural abstraction is formally defined as follows:

1. Structural abstraction on sequential composition:

   $\mathcal{C}' \succeq_{SA} \mathcal{C} \triangleq [\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$ and $\#\mathit{seq\text{-}op}(\mathcal{C}') > \#\mathit{seq\text{-}op}(\mathcal{C})$

   where $\#\mathit{seq\text{-}op}(\mathcal{C})$ and $\#\mathit{seq\text{-}op}(\mathcal{C}')$ represent the number of sequential composition operators in $\mathcal{C}$ or $\mathcal{C}'$.

2. Structural abstraction on parallel composition:

   $\mathcal{C}' \succeq_{SA} \mathcal{C} \stackrel{\triangle}{=} [\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$ and $\#par\text{-}op(\mathcal{C}') > \#par\text{-}op(\mathcal{C})$

   where $\#par\text{-}op(\mathcal{C})$ and $\#par\text{-}op(\mathcal{C}')$ represent the number of parallel composition operators in $\mathcal{C}$ or $\mathcal{C}'$.

The above definition means that representation $\mathcal{C}$ is a structural abstraction of $\mathcal{C}'$ on the following conditions: 1. the semantics of $\mathcal{C}$ implies the semantics of $\mathcal{C}'$; 2. There is at least one sequential or parallel composition reduced in the new representation.

Structural abstraction is useful because a system is composed of simpler components or subsystems through the above two basic compositions no matter how complex it is.

For example, there is a program in charge of the switch on and off of motors in a streamline control system. When an interrupt to stop the streamline occurs, the four motors need to switch off within 5 milliseconds but the order is not important, i.e., the four motors can be switched off in any order. After analysing the program, the software reengineer gets the following representation:

$$interrupt = stop \wedge \Delta 5ms \wedge ((switchoff(motor1);\ switchoff(motor2)) \\ \| \ (switchoff(motor3);\ switchoff(motor4)))$$

The above formula not only requires switching off the four motors within 5ms but also defines the order—motor1 ahead of motor2, motor3 ahead of motor4. This happened because the developer composed the switch operations with sequential composition. Viewing the order as specific implementation detail, the software reengineer extracts a more high level representation by reducing unnecessary compositions:

$$interrupt = stop \wedge \Delta 5ms \wedge (switchoff(motor1) \wedge switchoff(motor2) \\ \wedge switchoff(motor3) \wedge switchoff(motor4))$$

In the new representation, the order is no longer defined, and the original sequential compositions disappear.

6.2.5 Data Abstraction

Data abstraction is a general technique by which one can change the state space in an abstraction. Data abstraction allows software engineer to extend and change the original data types in legacy code to more high level and proper data types. In the absence of data abstraction, data structure identified from legacy code remains unchanged during the whole reverse engineering process although it will help to acquire better specification if the data structure is mapped to a more suitable one. Data abstraction is a quite complex means to reverse engineering. Correct data abstraction can improve the resulting specification greatly, while improper data abstraction may result in degraded specification.

In a data abstraction, a *data abstraction relation* must be defined first, which maps the original data structures to new data structures and therefore the original data states to new data states. The condition of data abstraction is that the semantics of the new representation must be a weakening of the original representation. If it is difficult to judge, then the data states of the original representation needs to be mapped over the data abstraction relation.

The formal definition of data abstraction is as follows:

Assuming $\mathcal{A}$ and $\mathcal{B}$ are two representations, r is a data abstraction relation:

$$r = (\textit{states of } \mathcal{A} \longrightarrow \textit{states of } \mathcal{B})$$

or in a more formal format:

$$r = \{(x, y) : \ x \in X, \ y \in Y, \ X = \{\textit{states of } \mathcal{A}\}, \ Y = \{\textit{states of } \mathcal{B}\}\}$$

where a *state* of a representation consists of the values of all the variables in the frame of the representation. Therefore, $\mathcal{A}$ is data-abstracted to $\mathcal{B}$ on relation r, denoted $\mathcal{A} \succeq_{DA-r} \mathcal{B}$, is defined as:

$$\mathcal{A} \succeq_{DA-r} \mathcal{B} \stackrel{\wedge}{=} r([\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$$

The above definition means that $\mathcal{B}$ is a data abstraction of $\mathcal{A}$ on relation r on the condition that if the states of $\mathcal{A}$ are mapped to those of $\mathcal{B}$, then the semantics of $\mathcal{B}$ is a weakening of the mapped semantics of $\mathcal{A}$.

Assume we have a queue data structure implemented with an array. Here we only give the element insertion procedure as an example.

```
Queue-body={
    string: field1;
    float: field2;
    }
Queue-body: array queue-body[maximum];
integer:queue-head;
integer:queue-tail;

proc insert(In element: Queue-body)
    {
    if Abs(queue-head - queue-tail)+1≥maximum then Print("queue is full")
    else queue-tail:=(queue-tail+1) mod maximum;
        queue-body[queue-tail]:= element
    fi;
}
```

The procedure is a little bit complex because array "queue-body" has a capacity limit and therefore quite a lot attention was paid to the related error handling. We use data abstraction to make the above procedure more concise: mapping the array data structure to a list structure.

```
Queue-body={
    string: field1;
    float: field2;
    Queue-body: link;
    }
Queue-body: queue-head, queue-tail;
```

```
proc insert(In element: Queue-body)
    {
        queue-tail.link:=element;
        queue-tail:= queue-tail.link
    }
```

In the new procedure, all implementation details related to array capacity is left out. The remaining information is more concise and easy to understand.

6.3 Healthiness Obligation

Healthiness obligations are conditions that must hold in order to have "sensible" abstractions. Different abstractions have different healthiness obligations. These are similar to Dijkstra's healthiness conditions [58, 59] for his Guarded Command Language. One can think of them as axioms or invariants.

Hiding Abstraction

- Shared variables between different representations should not be hidden. These shared variables connect different representations and involve important design or functional information.

$$For\ all\ \mathcal{A} \succeq_{HA} \mathcal{B} \triangleq (\exists x \bullet [\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$$

there must be

$$x \notin shared\text{-}var(\mathcal{A}) \text{ and } x \notin shared\text{-}var(\mathcal{B})$$

which means that x is not a shared variable.

- Variables with visibility level of *zero* should not be hidden. This is because variables with visibility level of zero are global variables in structured legacy systems

and are crucial for the design and specification.

$$For\ all\ \mathcal{A} \succeq_{HA} \mathcal{B} \triangleq (\exists x \bullet [\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$$

there must be

$$visibility\text{-}level(x) \neq 0$$

Weakening Abstraction

- Any representation should not be abstracted to TRUE or FALSE (trivial specification or starting from scratch). Although abstraction throws away irrelevant or unimportant details, it does not make sense to throw away everything.

$$For\ all\ \mathcal{A} \succeq_{WA} \mathcal{B} \triangleq [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$$

there must be

$$\mathcal{B} \neq TRUE$$

Temporal Abstraction

- An infinite action cannot be performed in a finite interval.

$$For\ all\ \mathcal{A} \succeq_{TA} \mathcal{B} \triangleq ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]) \wedge R_T(T(\mathcal{A}), T(\mathcal{B}))$$

there must be

$$finite(\mathcal{A}) \Rightarrow finite(\mathcal{B})$$

- Any representation cannot be abstracted to an agent with negative time interval.

$$T(\mathcal{B}) \geq 0$$

Structural Abstraction

- Two finite representations in sequential or parallel composition can not be structurally abstracted to an infinite representation. This means that if there is any contention between the two representations, for example, resource deadlock, then the sequential or parallel composition can not be reduced.

$$For\ all\ \mathcal{C}' \succeq_{SA} \mathcal{C} \triangleq [\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$$

where $\#seq\text{-}op(\mathcal{C}') > \#seq\text{-}op(\mathcal{C})$ or $\#par\text{-}op(\mathcal{C}') > \#par\text{-}op(\mathcal{C})$

there must be

$$finite(\mathcal{C}') \Rightarrow finite(\mathcal{C})$$

Data Abstraction

- Recursion on data abstraction relation is forbidden. This means that the variable set of $\mathcal{A}$ should not be the same of $\mathcal{B}$, i.e., data abstraction relation should not map to itself. In that case, data abstraction turns into weakening abstraction because data abstraction relation is then just the identity relation.

$$For\ all\ \mathcal{A} \succeq_{DA-r} \mathcal{B}$$

there must be

$$W_{\mathcal{A}} \neq W_{\mathcal{B}}$$

6.4 Monotonicity of Abstraction Relations

Monotonicity is important to abstraction because any abstraction is actually performed on a part of the whole software system. If the abstractions defined are not monotonic within most common context of popular software systems, the usage of the abstractions will be limited to a quite great extent.

Let $\mathcal{CX}$ be a context, $\mathcal{CX}$ is monotonic with respect to $\succeq_f$ if $\mathcal{A}_i \succeq_f \mathcal{B}_i$, $i = 0, ..., k$,

then $\mathcal{CX}(\bar{\mathcal{A}}) \succeq \mathcal{CX}(\bar{\mathcal{B}})$ holds where $\bar{\mathcal{A}} = \mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_k$ and $\bar{\mathcal{B}} = \mathcal{B}_0, \mathcal{B}_1, ..., \mathcal{B}_k$.

The conclusion we have reached is that: for conjunction, disjunction, sequential composition, parallel composition and the conclusion part of implication, which cover all the normal context in logic and source code program, weakening abstraction and structural abstraction are monotonic, temporal abstraction is monotonic only if R_T is a kind of functions to which the above context is also monotonic, hiding abstraction is monotonic only over disjunction and sequential composition, and data abstraction is not monotonic. However, since weakening abstraction is the basic semantic foundation of all the five category of abstractions, temporal, hiding and data abstractions performed on local parts of a software system can be "inherited" by further abstraction process in the sense of weakening abstraction.

The formal definition is as follows:

For $\mathcal{CX} = \wedge \mid \vee \mid ; \mid \parallel \mid \Rightarrow$(conclusion part),

- $\mathcal{CX}$ is monotonic with respect to $\succeq_{WA}$;
- $\mathcal{CX}$ is monotonic with respect to $\succeq_{SA}$;
- $\mathcal{CX}$ is monotonic with respect to $\succeq_{TA}$ if $\mathcal{CX}$ is monotonic with respect to R_T;
- $\vee$ and ; are monotonic with respect to $\succeq_{HA}$, but $\wedge$, $\parallel$ and $\Rightarrow$ are not.
- $\mathcal{CX}$ is not monotonic with respect to $\succeq_{DA}$. Here is an counter example. Assume $\mathcal{A}_1 \succeq_{DA-r_1} \mathcal{A}'_1$, and $\mathcal{A}_2 \succeq_{DA-r_2} \mathcal{A}'_2$. However, r_1 is not applicable on $\mathcal{A}_2$ and r_2 is not applicable on $\mathcal{A}_1$, hence, $\mathcal{A}_1 \not\succeq_{DA-r_2} \mathcal{A}''_1$, and $\mathcal{A}_2 \not\succeq_{DA-r_1} \mathcal{A}''_2$. Here $\mathcal{A}''_1$ and $\mathcal{A}''_2$ represent any possible representations. Therefore, $\mathcal{A}_1;\ \mathcal{A}_2 \not\succeq_{DA-r_1} \mathcal{A}'_1;\ \mathcal{A}''_2$ and $\mathcal{A}_1;\ \mathcal{A}_2 \not\succeq_{DA-r_2} \mathcal{A}''_1;\ \mathcal{A}'_2$. Hence, both $\mathcal{CX}(\mathcal{A}_1, \mathcal{A}_2) \succeq_{DA-r_1} \mathcal{CX}(\mathcal{A}'_1, \mathcal{A}''_2)$ and $\mathcal{CX}(\mathcal{A}_1, \mathcal{A}_2) \succeq_{DA-r_2} \mathcal{CX}(\mathcal{A}''_1, \mathcal{A}'_2)$ are false.

The above conclusion has been proven sound in formal logic. Interested reader may refer to the appendix.

6.5 Relations between Abstractions

The partial ordering relations between the five categories of abstractions discussed in section 6.2 are shown in Figure 6.1. I.e.:

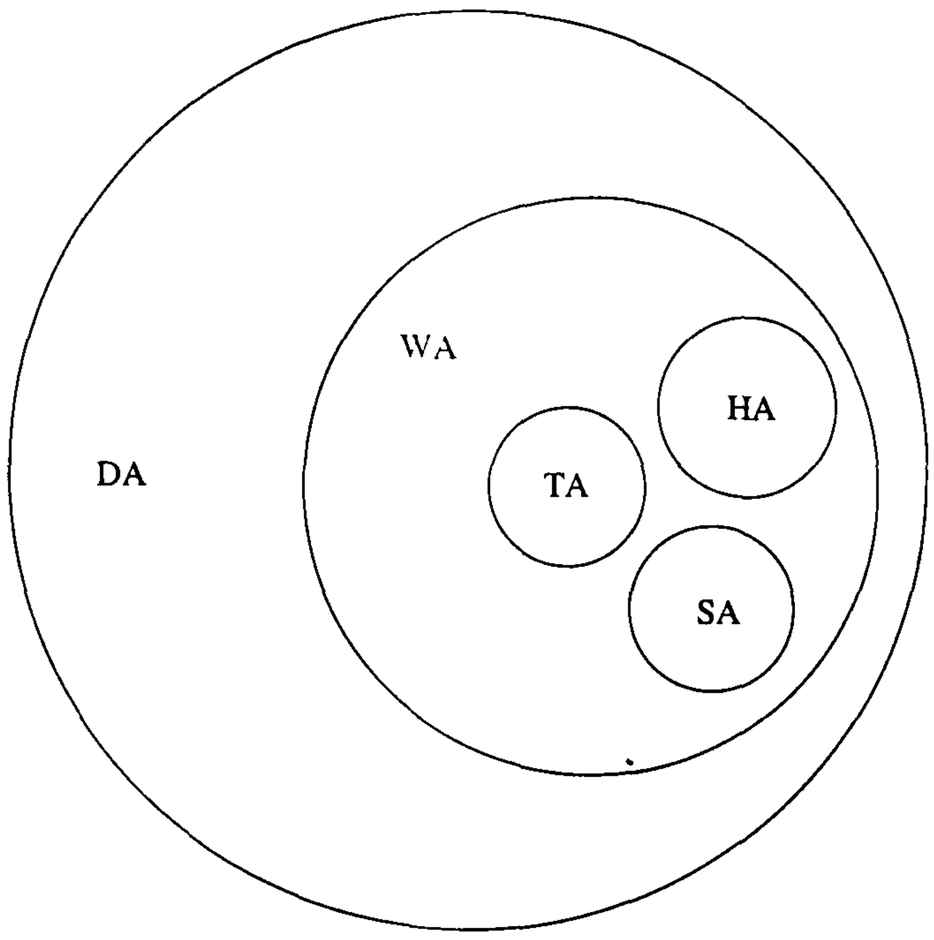

Figure 6.1: Partial Ordering Relations between Abstractions.

The following conclusion has been proven sound in formal logic. The proof is given in the appendix.

1. Temporal abstraction, structural abstraction and hiding abstraction are weakening abstraction, too. This means that weakening abstraction is the basis of these abstractions. In another words, temporal abstraction, structural abstraction and hiding abstraction are stronger in semantics than weakening abstraction. The reason is that semantics weakening is a part of the definitions of other abstractions. Abstraction is different from both transformation and restructuring, and there should be a consistence between the original semantics and the abstracted semantics.

2. Temporal abstraction, structural abstraction and hiding abstraction are independent to each other. There is no partial ordering or overlap between them.

3. Data abstraction is the most general. If the variable set of $\mathcal{A}$ remains to be the same of $\mathcal{B}$, i.e., the data abstraction relation r maps to itself, then data abstraction turns into weakening abstraction.

6.6 Elementary Abstraction Rules

A formal notation is used to describe abstraction rules in our study. We have classified the abstraction rules obtained through our study into two categories: *elementary abstraction rules*, rules to abstract source statements into logic formulae, which may be very redundant and specific; and *further abstraction rules*, which extract a more concise and abstract specification from the formulae through compositions and semantics weakening. Also, abstraction rules fall into different sections according to the domain that the rules deal with. For example, when dealing with an object-oriented (time-critical) system, the abstraction rules consist of *general abstraction rules*, *object-oriented abstraction rules* and *time critical rules*.

Abstraction rules in this category aim to abstract the statements in TGCL and ObTAM to formulae in ITL (the resultant formulae may be redundant, or even "too specific") and these rules can transform source statements into logic formulae, which is a kind of specification. So, in further abstraction, logic composition and semantic weakening will be applied through further abstraction rules to abstract these formulae to a more concise and abstract specification.

The statements in TGCL and ObTAM consist of two sets: *simple statements* such as assignment, input and output, and *composite statements* which are a composition of simple statements and composite statements through composition structures, such as condition, loop and procedure. Therefore, elementary abstraction rules fall into two sets correspondingly: the first set which is named *Primitive Abstraction Rules* converts the simple statements to ITL formulae, and the second set which is named *Compound Abstraction Rules* deals with the composite statements.

6.6.1 Primitive Abstraction Rules

Primitive Abstraction Rules aim at converting the simple statements in RWSL to ITL formulae. The formal definition of Primitive Abstraction Rules is as follows:

$$St \succeq Sp$$

where St denotes a simple statement in concrete code, and Sp is the abstract specification for St, that is, the semantics of St in logical form.

Proof:

$[\![St]\!] = Sp$ (definition of St)

Hence, $[\![St]\!] \Rightarrow Sp$

Hence, $St \succeq Sp$.

Rules listed in this subsection are instances of the Primitive Abstraction Rules and are proven sound in ITL based on the semantic weakening definition of abstraction. Due to similarity, the proofs of these rules are carried out in the proof of Primitive Abstraction Rules.

Assume $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}_i$, $\mathcal{B}_i$ are system representations, and $\Phi, \Psi, \Phi_i, \Psi_i$ are formulae, then we have the following primitive abstraction rules:

1. Assignment

$$x := e \succeq \{x\} : \bigcirc x = e$$

This rule extracts a logic formula of the assignment statement which assigns the value of expression e to variable x.

Example Consider a simple calculation:

$$y := x * y + 8 \succeq \{y\} : \bigcirc y = x * y + 8$$

2. Input Statement

$$(x, y) \leftarrow s \succeq \{x, y\} : x = \surd s \wedge y = \mathsf{read}\,(s)$$

This rule extracts a logic formula of the input statement which reads the value in shunt s to variable y and store the timestamp in x.

Example Consider reading temperature from a thermo sensor (shunt):

$$(tm, temp) \leftarrow sensor \succeq \{tm, temp\} : tm = \surd sensor \wedge temp = \mathsf{read}\,(sensor)$$

3. Output Statement

$$x \rightarrow s \succeq \{s\} : \mathsf{skip} \wedge \bigcirc s = (\surd s + 1, x) \quad .$$

This rule extracts a logic formula of the output statement which writes the value of variable or expression x to shunt s, and change the timestamp of s to the time when last write operation happened.

Example Consider setting the high limit to alarm of a water level sensor (shunt):

$$20m \rightarrow sensor \succeq \{sensor\} : \mathsf{skip} \wedge \bigcirc sensor = (\surd sensor + 1, 20m)$$

4. Type Definition

$$x : T \succeq \exists\, x \bullet f_T(x) \wedge scope(x)$$

The statement declares variable x of type T. This is expressed in logic as variable x has the feature of type T, which is described with function $f_T(x)$, and the valid scope of x is described with $scope(x)$ which depends on the definition context.

Example Consider variable *age* defined as an integer:

$$age : Integer \succeq \exists\, age \bullet Integer(age) \wedge scope(age)$$

5. Delay

$$delay\ n \succeq len = n$$

Delay means doing nothing during the specified period. The statement defines a delay lasting n time unit, which is expressed with the formula $len = n$.

Example Consider operations on a CPU are delayed 100 time units:

$$delay\ 100 \succeq len = 100$$

6.6.2 Compound Abstraction Rules

Compound Abstraction Rules aim at converting composite statements to ITL formulae.

The formal definition of Compound Abstraction Rules is as follows:

$$\frac{S_i \succeq \Phi_i}{\mathcal{C}(S_i) \succeq f_{\mathcal{C}}(\Phi_i)}$$

where $f_{\mathcal{C}}$ denotes logical construction corresponding to composition operator $\mathcal{C}$, and S_i denotes simple statements or composite statements.

Proof:

$[\![\mathcal{C}(S_i)]\!] = f_{\mathcal{C}}([\![S_i]\!])$ (from the definition of $f_{\mathcal{C}}(S_i)$)

Since $S_i \succeq \Phi_i$, hence $[\![S_i]\!] \Rightarrow \Phi_i$

Since $[\![S_i]\!] \Rightarrow \Phi_i$ and $f_{\mathcal{C}}(S_i)$ is monotonic with $\Rightarrow$ relation

Hence, $f_{\mathcal{C}}([\![S_i]\!]) \Rightarrow f_{\mathcal{C}}(\Phi_i)$

Hence, $[\![\mathcal{C}(S_i)]\!] \Rightarrow f_{\mathcal{C}}(\Phi_i)$

Hence, $\mathcal{C}(S_i) \succeq f_{\mathcal{C}}(\Phi_i)$

Rules listed in this subsection are instances of the Compound Abstraction Rules and are proven sound in ITL based on the semantic weakening definition of abstraction. Due to similarity, the proofs of these rules are carried out in the proof of Compound Abstraction Rules.

Assume $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}_i$, $\mathcal{B}_i$ are system representations, and $\Phi, \Psi, \Phi_i, \Psi_i$ are formulae, then we have the following abstraction rules:

1. Sequential Composition

$$\frac{\mathcal{A} \succeq \Phi \qquad \mathcal{B} \succeq \Psi}{\mathcal{A} \,;\, \mathcal{B} \succeq \mathsf{frame}(\Phi) \cup \mathsf{frame}(\Psi) : \Phi \,;\, \Psi}$$

If two representation fragments have a sequential composition relation, they can be abstracted separately, and the result representations should be composed with a sequential operator. The new frame is the union of both original frames.

Example Consider two delay operations in sequence:

delay x; delay y

Performing abstraction on the two delay statements separately, we got:

$$len = x \,;\, len = y$$

Then compose the two formulae with sequential operator:

$$len = x \,;\, len = y$$

By further abstraction we got the final result:

$$len = x + y$$

2. Conditional Statement

$$\mathcal{A}_i \succeq \Phi_i \text{ (for all } i \in I)$$

$$\textit{if } \square_{i \in I} g_i \textit{ then } \mathcal{A}_i \textit{ fi} \succeq \bigcup_{i \in I} \mathsf{frame}\,(\mathcal{A}_i) : (\bigvee_{i \in I}(g_i \wedge \Phi_i)) \vee (\bigwedge_{i \in I} \neg g_i)$$

This rule extracts a logic formula from a conditional statement. Each guarded branch can be abstracted separately and then composed together with disjunction. The new frame is the union of the frames of all branches.

Example Consider the symbolic function:

```
if x>0 then y:=1
else if x=0 then y:=0
     else if x<0 then y:=-1 fi
     fi
fi
```

We perform abstraction on the "else" branch of the outer "if" statement first:

```
if x>0 then y:=1
else {y} : (x = 0 ∧ y := 0) ∨ (x < 0 ∧ y := −1) ∨ (x > 0)
fi
```

Then we abstract the "if" branch of the outer "if" statement and compose the result together:

$$\{y\} : (x > 0 \wedge y := 1) \vee (x = 0 \wedge y := 0) \vee (x < 0 \wedge y := -1)$$

Further abstraction may be possible to make the above representation more abstract.

3. Iteration Statement

$$\frac{\mathcal{A} \succeq \Phi}{\textit{while } g \textit{ do } \mathcal{A} \textit{ od} \succeq \mathsf{frame}(\Phi) : (g \wedge \Phi)^* \wedge \textit{fin}(\neg g)}$$

This rule extracts a logic formula of an iteration statement. The iteration is mapped into "chopstar" formula in ITL, and the iteration body can be abstracted separately and then joined into the chopstar structure. The new frame equals the frame of the iteration body.

Example Consider a loop implementing factorial calculation:

```
while n>1 do y:=y*n; n:=n-1 od
```

With the above rule, it is abstracted to:

$$\{y, n\} : (n > 1 \wedge (\bigcirc y = y * n \,;\, \bigcirc n = n - 1))^* \wedge \textit{fin}(n \leq 1)$$

4. Procedure Definition

$$\frac{\mathcal{A}' \succeq \Phi}{\textit{proc } P(\textit{In } pin_i : T_i,\ \textit{Out } pout_j : T'_j)\{\mathcal{A}'\} \succeq \{pout_j\} \cup \mathsf{frame}(\Phi) : \Phi}$$

where Observables=$\{pin_i, pout_j$, global variables to P$\}$

Scope=$\{$local variables of P$\}$

A procedure definition is abstracted into a separate specification in ITL with its input parameters stable and output parameters possibly nonstable. The procedure body can be abstracted separately and then join the parameter part with conjunction. The new frame is the union of $pout_j$ and the frame of the procedure body. Observables are defined to include parameters and global variables of the procedure, which form the interface of the procedure. Local variables should be

deleted with their effects recorded in further abstraction because they are considered as implementation details.

Example Consider a procedure as follows:

```
proc calculator(In x integer; Out y:integer)
{
  integer: i;
  y:=1;
  for i:=1 to 8 step 1 do
     y:=y*x
  od
}
```

The observables of calculator consists of x and y, and the scope only consists of local variable i which should be hidden. In the first step, we abstract the procedure body with corresponding rules:

$$\{y\} : \exists\, i \bullet integer(i) \wedge y := 1\ ;\ (y := y * x)^8$$

Since i is no longer used in the representation, its declaration is left out. The complete result is as follows:

$$calculator \succeq \{y\} : (y := 1\ ;\ (y := y * x)^8)$$

More concisely, the final result is: $calculator \succeq \{y\} : y := x^8$.

5. Procedure Invocation

$$\frac{\mathcal{A}' \succeq \Phi}{P(In\ e_i,\ Out\ x_j) \succeq \{x_j\} : \Phi(pin_i/e_i, pout_j/x_j)}$$

where *proc* $P(In\ pin_i : T_i,\ Out\ pout_j : T'_j)\{\mathcal{A}'\}$

The invocation of a procedure equals the execution of the procedure's abstracted body with the input parameters' values passed in and output parameters returned.

Example Assume the procedure defined in "procedure definition" is invoked in a conditional statement:

```
if num>10 then calculator(In num, Out y)
```

The abstracted result should be: $\{y\} : num > 10 \wedge y := num^8$

6. Parallel

$$\mathcal{A} \succeq \Phi,\ \mathcal{B} \succeq \Psi$$

$$parbegin\ \mathcal{A} \parallel \mathcal{B}\ parend \succeq \mathsf{frame}(\Phi) \cup \mathsf{frame}(\Psi) : (\Phi \wedge \Psi)$$

Two concurrency or parallel representations can be abstracted separately and the results are composed through the conjunction operator. The new frame is the union of both original frames.

Example Assume there are two control procedure running concurrently, one to monitor the methane level in a mine, the other to monitor the water level:

```
parbegin
    methane-monitor() || water-level-monitor()
parend
```

With this rule, the program will be abstracted to the following:

$$methane\text{-}monitor() \wedge water\text{-}level\text{-}monitor()$$

7. Duration

$$\frac{\mathcal{A} \succeq \Phi}{[t]\mathcal{A} \succeq \mathsf{frame}\,(\Phi) : (\Delta t \wedge \Phi \ ; \ true) \wedge (\Phi \supset len <= t)}$$

Duration means that the execution of the specified representation should be finished within the indicated time duration. This rule extracts a logic formula from duration statement. It indicates that the execution body within a duration statement can be abstracted separately.

Example Assume the pump motor must be set off within 5ms once an alarm of high methane level occurs. The program is as follows:

```
duration 5ms  in
      motor-status:=off
end
```

With this duration rule, the program will be abstracted to the follows:

$$\{motor\text{-}status\} : \Delta 5ms \wedge motor\text{-}status := off \ ; \ true \wedge motor\text{-}status := off \supset len \leq 5ms$$

And then be further abstracted to:

$$motor\text{-}status := off \wedge len \leq 5ms$$

8. Signal

$$\frac{\mathcal{A}_1 \succeq \Phi_1 \qquad \mathcal{A}_2 \succeq \Phi_2}{\mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2 \succeq \mathsf{frame}\,(\Phi_1) \cup \mathsf{frame}\,(\Phi_2) \cup \{s\} : (\Delta t \wedge \mathsf{stable}\,(\surd s) \ ; \ \Phi_1) \vee (\Delta t \wedge \neg\, \mathsf{stable}\,(\surd s) \ ; \ \Phi_2)}$$

The two execution bodies in a signal statement can be abstracted separately and then joined together with the formula defined above. This rule extracts a logic formula from the signal statement.

Example Assume there is the following fragment of a control system. If there is an overload alarm within 10ms, the red light will be set on, otherwise the green light will be set.

```
wait on overload for 10ms do
        red-light:=on
else
        green-light:=on
end
```

With this signal rule, the program will be abstracted to the follows:

$$\{overload, red\text{-}light, green\text{-}light\} :$$

$$(\Delta 10ms \wedge \mathsf{stable}\,(\surd overload)\ ;\ red\text{-}light = on)$$

$$\vee(\Delta 10ms \wedge \neg\,\mathsf{stable}\,(\surd overload)\ ;\ green\text{-}light = on)$$

9. Object Definition

As type specification, classes defined in COOL or ObTAM programs will disappear once they are abstracted to ITL specification. Only objects exist as formulae with frames in ITL.

Let $T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T'_l)[\mathcal{A}_j]\}$, then

$$\frac{\mathcal{A}_j \succeq \Psi_j}{x : T \succeq W_x : f}$$

where $W_x = \bigcup_{i \in I} x_i$

$$f = \bigwedge_{i \in I} f_{T_i}(x_i) \wedge (\bigvee_{j \in J} \mathsf{frame}\,(\Psi_j) \cup \{pout_{j_l}\} : \Psi_j)^*$$

This rule transforms the definition of an object in source code into a logic description. W_x is the data fields of the object, it forms the object's *observables*. f is the behaviour description of the object where $\mathsf{frame}\,(\Psi_j) \cup \{pout_{j_l}\} : \Psi_j$ is the description of method m_j.

Example Assume there are a set of sensors in a control system. A class *sensor* is defined to describe the general features and operations of all sensors. The definition in COOL is as follows:

```
class Sensor
   {
      String: id;
      Boolean: status;
      enable() { status:=enabled };
      disable() { status:=disabled }
   }
```

status represents whether the sensor is enabled or disabled, and each sensor has a unique identification recorded in *id*. Class *Sensor* has two operations: enable() to enable the sensor, and disable() to disable it. With this abstraction rule, an object *sensor* of class *Sensor* can be abstracted to the follows:

$$sensor \succeq \{id, status\} : string(id) \wedge boolean(status) \wedge (enable() \vee disable())^*$$

$$enable() \succeq \{status\} : status := enabled$$

$$disable() \succeq \{status\} : status := disabled$$

10. Object Hierarchy

$$\text{Let } T = \{x_i : T_i,\ m_j(In\ pin_{j_k} : T_k,\ Out\ pout_{j_l} : T_l)[\mathcal{A}_j]\}$$

$$T' = \{y_{i'} : T'_{i'},\ m'_{j'}(In\ pin_{j'_{k'}} : T'_{k'},\ Out\ pout_{j'_{l'}} : T'_{l'})[\mathcal{A}'_{j'}]\}, \quad \text{then}$$

$$\mathcal{A}_j \succeq \Psi_j,\ \mathcal{A}'_{j'} \succeq \Psi'_{j'}$$

$$x : T <_{sub} T' \succeq W_x : f$$

where $W_x = \bigcup_{i \in I} x_i \cup \bigcup_{i' \in I'} y_{i'}$ iff for all $x_i\ i \in I, y_{i'} \neq x_i$

$$f = \bigwedge_{i \in I} f_{T_i}(x_i) \wedge \bigwedge_{i' \in I'} f_{T_{i'}}(y_{i'}) \wedge (\bigvee_{j \in J} \Phi_j \vee \bigvee_{j' \in J'} \Phi'_{j'})^*$$

iff for all $x_i\ i \in I, y_{i'} \neq x_i$, and iff for all $\Phi_j\ j \in J, \Phi'_{j'} \neq \Phi_j$

$$\forall j \in J \bullet \Phi_j = \mathsf{frame}\,(\Psi_j) \cup \{pout_{j_l}\} : \Psi_j$$

$$\forall j' \in J' \bullet \Phi'_{j'} = \mathsf{frame}\,(\Psi'_{j'}) \cup \{pout_{j'_{l'}}\} : \Psi'_{j'}$$

The subclass relation $<_{sub}$ is transitive. This rule transforms the object hierarchy definition, including inheritance, into a logic formula. Assume that T is a subclass of T', for any object x of class T, it will inherit all the data fields and methods in T' if they are not redefined in T. On the other hand, all the data fields and methods in T' will be overridden with the counterparts in T if they are redefined in T.

Example Assume there is some temperature sensors in the control system described above. Besides the general data and operations introduced in class *Sensor*, the temperature sensors have a data field to indicate current temperature and one related operation to read out the current temperature. The *enable* operation is overwritten: the current temperature reading must be reset to zero after the sensor being enabled. We define a subclass derived from class *Sensor*, namely *Temp-sensor*. All data and operations of *sensor* are inherited by *Temp-sensor* except that *enable* is overridden. The definition in COOL is as follows:

```
Temp-sensor extends Sensor;
```

```
class Temp-sensor
        {
            Float: current-temp;

            enable() { status:=enabled; current-temp:=0 };
            temp-read(Out temp) { temp:=current-temp }
        }
```

With the object hierarchy rule, an object *t-sensor* of class *Temp-sensor* will be abstracted to:

$$t\text{-}sensor \succeq \{id, status, current\text{-}temp\} : string(id) \wedge boolean(status) \wedge$$
$$float(current\text{-}temp) \wedge (enable() \dot{\vee} disable() \vee temp\text{-}read())^*$$
$$enable() \succeq \{status, current\text{-}temp\} : status := enabled \wedge current\text{-}temp := 0$$
$$disable() \succeq \{status\} : status := disabled$$
$$temp\text{-}read(temp) \succeq \{temp\} : temp := current\text{-}temp$$

11. Method Invocation

$$\frac{\mathcal{A} \succeq \Phi}{x.m(e_i, y_j) \succeq \{y_j\} : \Phi(pin_i/e_i,\ pout_j/y_j)}$$

where $m(In\ pin_i : T_i,\ Out\ pout_j : T_j)[\mathcal{A}]$

A method invocation equals the execution of the method's abstracted agent with the input parameters passed in and the result of output parameters returned.

12. Field Reference

$$x.d \succeq d \in W_x$$

A data field of an object is a variable belonging to the frame of the object.

6.7 Further Abstraction Rules

Further abstraction rules aim to extract more concise and abstract specifications from the formulae obtained through applying the elementary abstraction rules. Logic composition and semantics weakening are the basis of further abstraction. During further abstraction domain knowledge may be applied by software engineers to give the software system a more concise and "professional" description.

There is not any "object combination" during further abstraction, i.e. objects will be abstracted but not combined.

As stated in section 4.2.1, abstraction is a process of generalisation, removing restrictions, eliminating detail and removing inessential information. Unlike transformation which keeps the semantics unchanged, abstraction endeavours in weakening the original semantics of system implementation. Identification of the parts to be abstracted away cannot be determined automatically within the system, therefore, user guidance is needed. Further abstraction rules cover the principles to identify some kinds of implementation details, however, not a complete set of them. To increase the automation of the supporting tool, a set of abstraction patterns based on relevant further abstraction rules are developed as means of acquiring observations identified by software engineers. These observations are necessary informations for automated further abstractions based on corresponding rules. These abstraction patterns are embodied as *abstraction pattern assertions* in the resulting tool RA, which will be introduced in detail in Chapter 7.

Assume $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}_i$, $\mathcal{B}_i$ are representations, and $\Phi, \Psi, \Phi_i, \Psi_i$ are formulae, then we have the following abstraction rules. Proof of soundness of these rules are given in appendix.

1. Transitive

$$\mathcal{A} \succeq \mathcal{B}$$
$$\mathcal{B} \succeq \mathcal{C}$$

$$\frac{}{\mathcal{A} \succeq \mathcal{C}}$$

This rule states that a system representation can be abstracted step by step, and the final result will be an abstraction of the original representation if it is guaranteed that each step is an abstraction.

2. Monotonic

$$\frac{\mathcal{A} \succeq \mathcal{B} \qquad \mathcal{CX} = \wedge \mid \vee \mid ; \mid \parallel \mid \Rightarrow}{\mathcal{CX}(\mathcal{A}) \succeq \mathcal{CX}(\mathcal{B})}$$

For the context of conjunction, disjunction, sequential composition, parallel composition and implication, all abstractions discussed in section 6.2 are monotonic in the sense of weakening, temporal, hiding and structural abstraction.

3. Sequence Folding

$$\frac{[\![\mathcal{A} ; \mathcal{B}]\!] \Rightarrow [\![\mathcal{A} \wedge \mathcal{B}]\!]}{\mathcal{A} ; \mathcal{B} \succeq \mathcal{A} \wedge \mathcal{B}}$$

If no contradiction is caused when substituting the sequential composition between two representations to conjunction composition, then the sequence can be folded through conjunction.

This rule can be applied when the execution order of a sequence is not crucial. In non-parallel systems, this is true under most situations except any operation provides parts of the pre-conditions of its successor within the sequence. However, in parallel systems, if the sequence relates with communication or shared resources, it can not be folded with conjunction.

4. Specification Combination

$$4.1\ (W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (W_1 \cup W_2) : \Phi_1 \wedge \Phi_2$$

$$4.2\ (W_1 : \Phi_1) \vee (W_2 : \Phi_2) \succeq (W_1 \cup W_2) : \Phi_1 \vee \Phi_2$$

This rule is used to combine specifications in ITL because there are often quite a number of specifications within one software system and some of them can be potentially combined for further abstractions. Two specifications with conjunction relation can be merged into one specification with their frames united and their description formulae conjunctively composed. Similarly, two specifications with disjunction relation can be merged into one specification with their frames united and their description formulae disjunctly composed.

5. State Test and Exception Handling

State tests and exception handling are often used in programs to assure smooth execution. Although they may be important in system implementation, these details do not involve the crucial functionality of the system. Therefore, in high-level specification, these details are unnecessary and should be abstracted away. The related abstraction pattern is called "state test and exception handling pattern", which consists of the following cases:

- State test and exception handling branch. The identified state test and exception handling parts are branches in conditional structures. In this case, the branches should be abstracted away.

- State test and exception handling loop. The identified state test and exception handling part is a loop structure. In this case, the loop should be abstracted away.

- State test and exception handling component. The identified state test and exception handling part is a procedure or function (component). In this case, the component should be abstracted away.

- State test and exception handling expression. An expression is identified as related with state test and exception handling. In this case, the expression together with the smallest representation unit (statement or ITL formula) in which the expression directly locates should be abstracted away.
- State test and exception handling variable. A variable is identified as related with state test and exception handling. In this case, all the smallest representation units (statements or ITL formulae) in which the variable directly locates should be abstracted away.

6. User Interface Format

Almost all computing systems have to pay some attention to the format of its interface with the user. There are three sorts of so called user interface format:

- Input format
- Output format
- Graphic User Interface (GUI)

For some systems, a rather big part is devoted to making a better user interface format. However, these format related parts are not involved in the function core of the system and could be left out in the high level specification. The related abstraction pattern is called "user interface format pattern", which consists of the following cases:

- User interface format branch. The identified user interface format parts are branches in conditional structures. In this case, the branches should be abstracted away.
- User interface format loop. The identified user interface format part is a loop structure. In this case, the loop should be abstracted away.
- User interface format branch component. The identified user interface format part is a procedure or function (component). In this case, the component

should be abstracted away.

- User interface format branch expression. An expression is identified as related with user interface format. In this case, the expression together with the smallest representation unit (statement or ITL formula) in which the expression directly locates should be abstracted away.
- User interface format branch variable. A variable is identified as related with User interface format. In this case, all the smallest representation units (statements or ITL formulae) in which the variable directly locates should be abstracted away.

7. Semantic Core

   The semantic core of a specification is the part which covers the specification's key contents. In this abstraction pattern, once the semantic core of a specification is identified, further abstraction will keep the core but omit other parts of the specification.

8. Concise Specification

   If a more concise specification is observed and it is a weakening of the original specification, the software engineer could insert it as observations. Then in further abstraction the original specification will be substituted with this observation.

9. Comment Revision

   Comments in source code often give great help to the understanding of the system. During reverse engineering, comments should be kept and revised to fit specifications at different abstraction levels. For higher level specification, comments should be revised into abstract ones.

10. Trivial Elements

If a part of the system's functionality are considered too "trivial" to be kept in high level specification, the elements related to this part of functionality is identified as "trivial elements", which should be abstracted away in further abstraction.

Trivial elements could be the following cases:

- Trivial branch. The identified trivial element is a branch in a conditional structure. In this case, the branch should be abstracted away.
- Trivial loop. The identified trivial element is a loop structure. In this case, the loop should be abstracted away.
- Trivial component. The identified trivial element is a procedure or function (component). In this case, the component should be abstracted away.
- Trivial expression. The identified trivial element is an expression. In this case, the expression together with the smallest representation unit (statement or ITL formula) in which the expression directly locates should be abstracted away.
- Trivial variable. The identified trivial element is a variable. In this case, all the smallest representation units (statements or ITL formulae) in which the variable directly locates should be abstracted away.

11. Domain Function

Domain functions give more scientific and concise descriptions of the functionality of target systems. If a representation is identified as an implementation of certain domain function, then it should be abstracted back to the domain function in further abstraction. This will make the specification more abstract and concise. For example, the specification $\{x, y\} : (x > 0 \wedge y = y * x \,;\; x = x - 1)^{x-1}$ implements $y = x!$. Therefore, it can be abstracted as $\{x, y\} : y = x!$.

12. Efficiency-Improving Details

The implementation is often cluttered with information/details to improve the efficiency of the system. These details are normally not function related, and could be abstracted away in high-level specification. For example, using of register variable is to improve the system's efficiency, all the related parts are classified as efficiency-improving details.

This abstraction pattern consists of the following cases:

- Efficiency-improving branch. The identified efficiency-improving part is a branch in a conditional structure. In this case, the branch should be abstracted away.
- Efficiency-improving loop. The identified efficiency-improving part is a loop structure. In this case, the loop should be abstracted away.
- Efficiency-improving component. The identified efficiency-improving part is a procedure or function (component). In this case, the component should be abstracted away.
- Efficiency-improving expression. An expression is identified as related with efficiency improving. In this case, the expression together with the smallest representation unit (statement or ITL formula) in which the expression directly locates should be abstracted away.
- Efficiency-improving variable. A variable is identified as related with efficiency improving. In this case, all the smallest representation units (statements or ITL formulae) in which the variable directly locates should be abstracted away.

The formal representation of Rule 5 to 12 is as follows:

$$\mathcal{A} \succeq \Phi$$

$$\Phi \Rightarrow \Psi$$

$$\mathcal{A} \succeq \mathsf{frame}\,(\Psi) : \Psi$$

When moving from Φ to Ψ, the identified contents are abstracted away or substituted with more concise representation. These contents could be state test and exception handling detail, or user interface format detail, or trivial elements, or obsolete comments, or efficiency-improving details, or too detailed domain function description.

13. Conjunction

$$\frac{\mathcal{A} \succeq \Phi \qquad \mathcal{A} \succeq \Psi}{\mathcal{A} \succeq \mathsf{frame}(\Phi) \cup \mathsf{frame}(\Psi) : \Phi \wedge \Psi}$$

If a representation is abstracted separately into two results, a more accurate abstraction may be obtained by making a conjunction of them.

14. Specification

$$(W : \Phi) \wedge \mathsf{stable}(s) = (W - s) : \Phi \text{ (if } s \text{ not in } \Phi)$$

This rule eliminates the redundant variables in a specification. If a variable is stable and does not occurs in the description formula, then it should be left out of the frame.

15. Sequential

15.1 $\mathbf{empty}\ ;\ \mathcal{A} = \mathcal{A} = \mathcal{A}\ ;\ \mathbf{empty}$

15.2 $\mathcal{A}\ ;\ (\mathcal{B}\ ;\ \mathcal{C}) = (\mathcal{A}\ ;\ \mathcal{B})\ ;\ \mathcal{C}$

15.3 $\mathcal{A}_1\ ;\ (\mathcal{A}_2 \vee \mathcal{A}_3)\ ;\ \mathcal{A}_4 = (\mathcal{A}_1\ ;\ \mathcal{A}_2\ ;\ \mathcal{A}_4) \vee (\mathcal{A}_1\ ;\ \mathcal{A}_3\ ;\ \mathcal{A}_4)$

These rules indicate that sequential composition operator has empty as a unit and is associative and distributive over nondeterministic choice.

16. Delay

$\mathbf{delay}_{d_1}; \mathbf{delay}_{d_2} = \mathbf{delay}_{d_1+d_2}$

$\mathbf{skip} = \mathbf{delay}_1$

The first rule indicates that two successive delay can be simplified to one delay with the times added. The second rule indicates that skip equals to delay of one time unit.

17. Parallel

17.1 $\mathcal{A} \parallel \mathcal{B} = \mathcal{B} \parallel \mathcal{A}$

17.2 $\mathcal{A} \parallel (\mathcal{B} \parallel \mathcal{C}) = (\mathcal{A} \parallel \mathcal{B}) \parallel \mathcal{C}$

17.3 $\mathcal{A} \parallel \mathit{true} = \mathcal{A}$

17.4 $\mathcal{A} \parallel (\mathcal{B} \vee \mathcal{C}) = (\mathcal{A} \parallel \mathcal{B}) \vee (\mathcal{A} \parallel \mathcal{C})$.

17.5 $\mathcal{A} \parallel \mathcal{B} \succeq \mathcal{A}' \parallel \mathcal{B}$, for any $\mathcal{B}$ if $\mathcal{A} \succeq \mathcal{A}'$

17.6 $(G \Rightarrow \Phi_1) \parallel (G' \Rightarrow \Phi_2) \succeq (G \wedge G') \Rightarrow \Phi_1 \wedge \Phi_2$

The first four rules indicate that parallel composition operator is symmetric, associative, and distributive over nondeterministic choice. The fifth rule indicates that if one of the representations in parallel is abstracted the parallel composition is also abstracted. The last rule means that two parallel implications can be abstracted to one implication with the premises and conclusions conjunctively composed.

18. Signal

18.1 $(\mathcal{A} \trianglerighteq_s^n \mathcal{B}) \parallel (\mathcal{C} \trianglerighteq_s^n \mathcal{D}) = (\mathcal{A} \parallel \mathcal{C}) \trianglerighteq_s^n (\mathcal{B} \parallel \mathcal{D})$

18.2 $\mathcal{A} \trianglerighteq_s^n (\mathcal{C} \trianglerighteq_s^0 \mathcal{B}) = \mathcal{A} \trianglerighteq_s^n \mathcal{B}$

These rules are for the simplification of signal-related formulae. Two parallel signal formulae can be rewritten in one signal formula. An signal formula can be deployed over time span or the opposite.

19. Non-deterministic choice

19.1 $\mathcal{P} \vee \mathcal{P} = \mathcal{P}$

19.2 $\mathcal{P} \vee \mathcal{Q} = \mathcal{Q} \vee \mathcal{P}$

19.3 $\mathcal{P} \vee (\mathcal{Q} \vee \mathcal{R}) = (\mathcal{P} \vee \mathcal{Q}) \vee \mathcal{R}$

19.4 $true \vee \mathcal{P} = true$

These rules indicate that non-deterministic choice is reflective, symmetric and associative.

20. Iteration

$\mu_{n+1}\,\mathcal{A} = \mathcal{A};\ \mu_n\,\mathcal{A} = \mu_n\,\mathcal{A};\ \mathcal{A}$

This rule indicates that an iteration may be deployed totally or partially.

6.8 Demonstrative Examples

A prototype system, named the Reengineering Assistant (*RA*), has been developed and example programs have been experimented with the system. In this section, two example programs are used to demonstrate the proposed RWSL-based abstraction approach, i.e., how to extract an ITL specification from source code through the developed abstraction rules. Before the start, we assume that we know nothing about these programs, and the only information source is source code programs. The first example is a sequential program and the second one is a real-time, interrupt handling program.

Example 1 Assume we have a sequential CSL program as follows:

```
proc factorial(In x: int, Out y: int)
    {
        int: k;
```

```
    k:=x; y:=1;
    while (k>1) do y:=y*k; k:=k-1 od;
    if (k<0) then y:= -100 fi;
}
```

We first extract a specification of the program through applying the elementary abstraction rules:

$$factorial \succeq \{k, y\} : k := x \ ; \ y := 1 \ ;$$
$$((k > 1) \wedge (y := y * k \ ; \ k := k - 1))^* \ ; \ ((k < 0) \wedge y := \text{-}100)$$

We then apply the further abstraction rules to make the specification more concise through logic composition:

$$factorial \succeq \{k, y\} : k := x \ ; \ y := 1 \ ;$$
$$(k > 1) \wedge (y := y * k \ ; \ k := k - 1)^{k-1} \ ; \ ((k < 0) \wedge y := \text{-}100)$$

$k < 0$ is state test and exception handling added as implementation details to assure smooth execution. In the next step, these details are eliminated by applying a further abstraction rule (semantics weakening) and we obtain:

$$factorial \succeq \{k, y\} : k := x \ ; \ y := 1 \ ; \ (k > 1) \wedge (y := y * k \ ; \ k := k - 1)^{k-1}$$

In the next step, we make the specification more concise and professional by applying domain knowledge to identify the domain function $y = k!$:

$$factorial \succeq \{k, y\} : \ k := x \ ; \ y = k!$$
$$factorial \succeq \{y\} : \ y = x!$$

We can finally understand that the original program is for calculation the factorial of an integer. The specification is an ITL formula and software engineer can gain a better understanding of the program through it.

Example 2 Assume we have a real-time, interrupt handling program as follows:

```
proc pump-control()
{
  alarm: Shunt;
  sw: Boolean;
  tm: Integer;

while true do
    wait on alarm for 1ms
    do
        delay 0ms
    else
        (tm, signal)←alarm;
        if signal=High-alarm
        then [1ms]sw:=on fi;
        if signal=Low-alarm
        then [1ms]sw:=off fi;
    od
  od
}
```

We first extract a specification of the program through applying elementary abstraction rules. The signal agent in TGCL $\mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2$ is written in a more program-like style in CSL: **wait on** s **for** t **do** $\mathcal{A}_1$ **else** $\mathcal{A}_2$ **od**.

$$\begin{aligned} \textit{pump-control} \succeq \{\textit{alarm}, \textit{signal}, \textit{tm}\} : (\textit{true} \wedge ((\Delta 1 \wedge \mathsf{stable}\,(\surd \textit{alarm})\ ;\ \textit{len} = 0) \vee \\ (\Delta 1 \wedge \neg\, \mathsf{stable}\,(\surd \textit{alarm})\ ;\ (\textit{tm} = \surd \textit{alarm} \wedge \textit{signal} = \mathsf{read}\,(\textit{alarm})\ ; \\ (\textit{signal} = \textit{High-alarm} \wedge \Delta 1 \wedge \textit{sw} := \textit{on} \wedge (\textit{sw} := \textit{on} \supset \textit{len} \leq 1))\ ; \\ (\textit{signal} = \textit{Low-alarm} \wedge \Delta 1 \wedge \textit{sw} := \textit{off} \wedge (\textit{sw} := \textit{off} \supset \textit{len} \leq 1)))))))^* \end{aligned}$$

We then apply further abstraction rules to make the specification more concise through logic composition:

$$\begin{aligned} \textit{pump-control} \succeq \{\textit{alarm}, \textit{signal}, \textit{tm}\} : (\textit{true} \wedge ((\Delta 1 \wedge \mathsf{stable}\,(\surd \textit{alarm})\ ;\ \textit{len} = 0) \vee \\ (\Delta 1 \wedge \neg\, \mathsf{stable}\,(\surd \textit{alarm})\ ;\ (\textit{tm} = \surd \textit{alarm} \wedge \textit{signal} = \mathsf{read}\,(\textit{alarm})\ ; \end{aligned}$$

$$(signal = \mathit{High\text{-}alarm} \wedge sw := \mathit{on} \wedge len \leq 1)\ ;$$

$$(signal = \mathit{Low\text{-}alarm} \wedge sw := \mathit{off} \wedge len \leq 1))))^*$$

Since timestamp *tm* is never used in the program, we delete it through a further abstraction rule. Since $len = 0$ means empty operation, we change it to *empty*. Because an alarm cannot be both a high level alarm and a low level alarm simultaneously, we change the sequential composition between the two condition statements into non-deterministic choice. Meanwhile, simplify the specification by taking away of *true* condition.

$$\begin{aligned} \mathit{pump\text{-}control} \succeq \{alarm, signal\} : ((&\Delta 1 \wedge \mathsf{stable}\,(\surd alarm)\ ;\ \mathit{empty}) \vee \\ &(\Delta 1 \wedge \neg\, \mathsf{stable}\,(\surd alarm)\ ;\ (signal = \mathsf{read}\,(alarm))\ ; \\ &(signal = \mathit{High\text{-}alarm} \wedge sw := \mathit{on} \wedge len \leq 1) \vee \\ &(signal = \mathit{Low\text{-}alarm} \wedge sw := \mathit{off} \wedge len \leq 1))))^* \end{aligned}$$

Through the final specification we can understand that the program keeps testing whether there is any interrupt (alarm) sent by water level sensors. If the alarm is a high level alarm which means the water level hitting the high safety limit, the pump must be switched on within 1 millisecond; similarly, if the signal is a low level alarm, the pump must be switched off within 1 millisecond. If there is not any alarm, no operation is performed. In fact, this program describes a simple water drainage pump control system.

Chapter 7

Reengineering Assistant: A Realisation

7.1 System Architecture

Reengineering Assistant (RA) is a semi-automatic tool which aims at helping software engineers in quite comprehensive processes of the reengineering of legacy systems. RA is a rule-based intelligent system. Automation is the goal of RA, however, human intervention is crucial in reverse engineering, i.e., full automation is impossible, RA adopts semi-automation to facilitate the process of reengineering. Figure 7.1 shows the general system architecture of RA.

The architecture reflects the working flow of RWSL in figure 4.5. The legacy source code is firstly translated into CSL or COOL, and then the CSL/COOL code is parsed and displayed in the browser interface. An internal LISP database of the code is meanwhile generated. The internal database is in a form of syntax tree, which is convenient for transformation and abstraction. Once the CSL/COOL is parsed and stored, software engineers could choose three different processes to reengineer the legacy system:

1. Program Transformer. CSL/COOL code is improved through the program transformer. New required extra functionalities can also be added. The new CSL/COOL code can then be translated into an equivalent programming language through a universal translator.

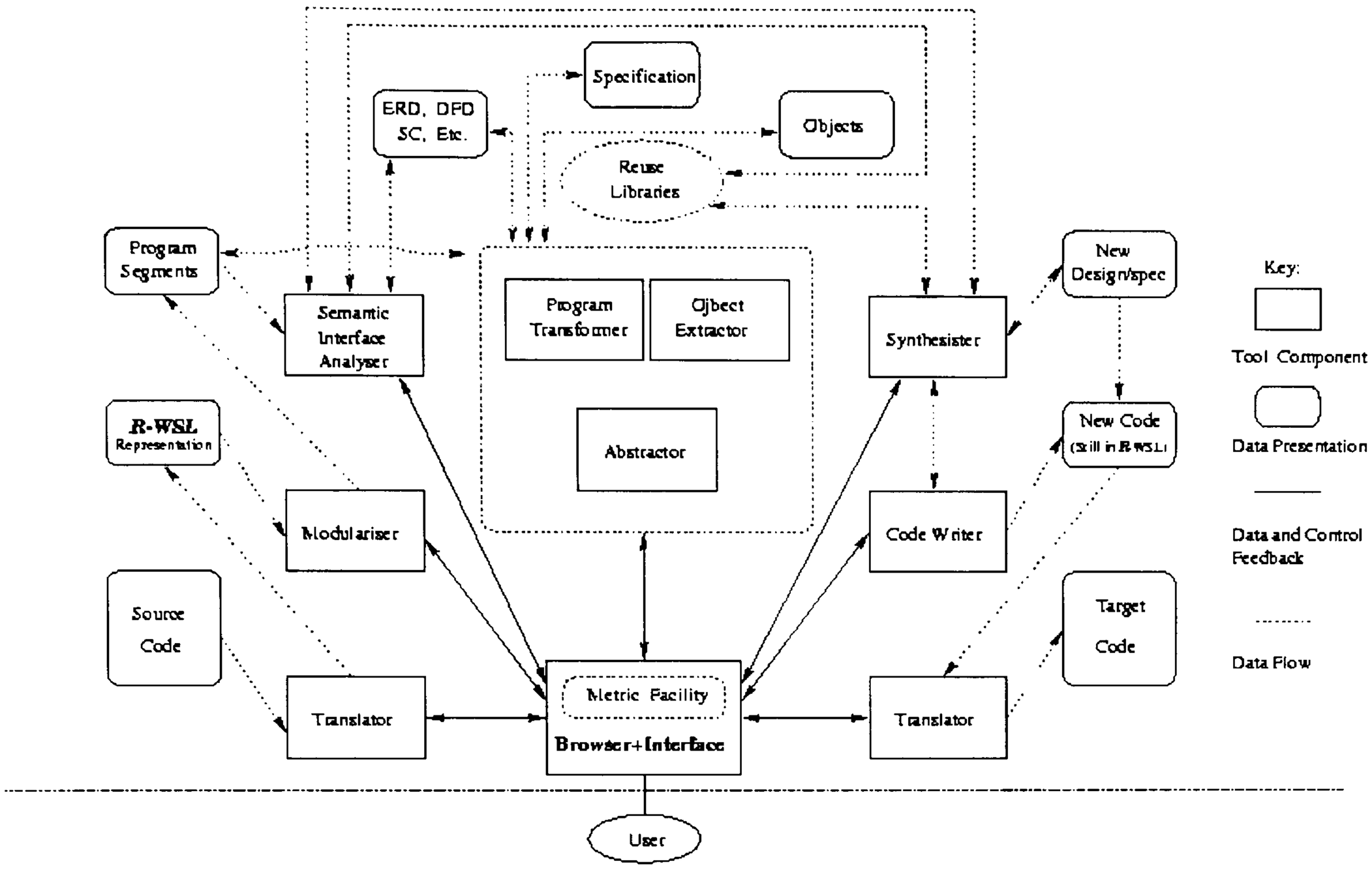

Figure 7.1: General System Architecture of Reengineering Assistant

2. Object Extractor. If the object-oriented paradigm is sought, object extraction is performed on CSL code to obtain an equivalent COOL code. Then the COOL code could be extended or improved or left unchanged. Subsequently, the new code can be transformed into an object-oriented language, such as ADA, JAVA and C++.

3. Abstractor. This is the main part on which this thesis concentrates. To seek a high level specification, the abstractor extracts it from CSL or COOL code with abstraction technology discussed in the thesis. The abstraction taxonomy and rules discussed in chapter 6 are implemented in the abstractor. The extracted specification could subsequently be used as a basis for re-specification, re-design and forward engineering through refinement.

In further development, RA could absorb reuse techniques by building up *Reuse Libraries* and *The Synthesiser*. Reuse libraries are used to store reuseable components,

which may form a repository. The synthesiser can build up new systems by integration of components in the reusable library. Graphic models may also be introduced in RA to help understand the legacy system, for example, Entity-Relation diagram (ER), Data Flow Diagram (DFD), and Structure Chart (SC).

The sole interface between software engineers and RA is the *Browser-Interface*. It has the following functions:

- To display the translated legacy source code in CSL/COOL;
- To accept process command from software engineers;
- To accept necessary information which must be acquired from software engineers;
- To display process results, including extracted specification, new object-oriented COOL program, and transformed source code;
- To display the metric result of processes.

A more detailed architecture of the reverse engineering part in RA, i.e., extraction of ITL specification from legacy source code, is shown in figure 7.2.

The lexical scanner and parser are used to check the syntax of the CSL/COOL code; any error will be reported to the software engineer through the browser-interface for correction. Correct programs will be stored in the CSL/COOL LISP Database in specially designed syntax tree structure. Meanwhile, the program displayer is started to display the program in the browser-interface in pretty print format, for example, with indentation and different fonts.

The program abstractor is the most important part, it is an inference machine. Various abstractions are classified into corresponding catalogues, and abstraction rules are implemented as inference rules. The extracted ITL specification is stored in the ITL LISP Database, in a syntax tree structure specially designed for logic formulae. During abstraction inference, the CSL/COOL LISP database and ITL LISP database provide

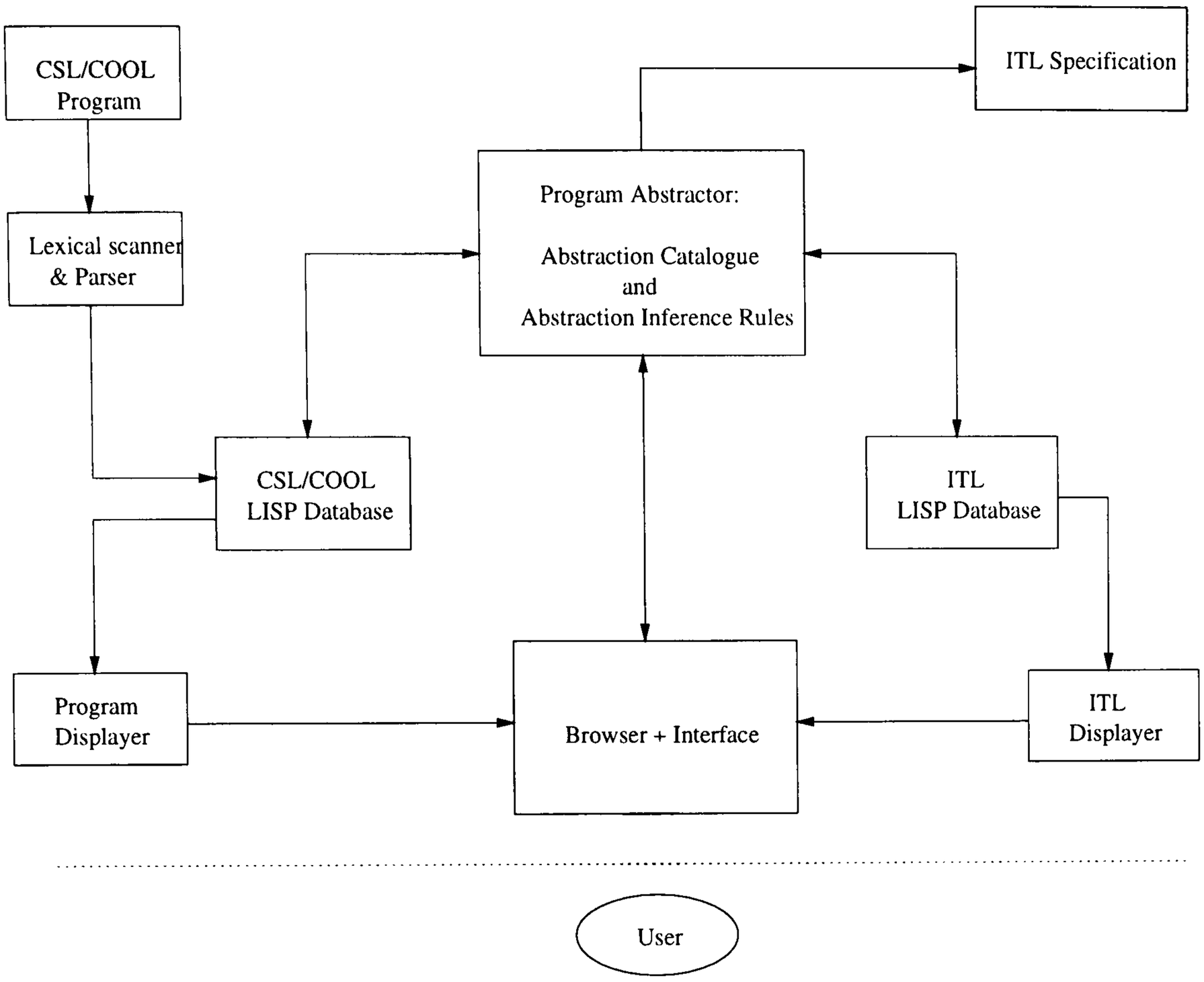

Figure 7.2: Architecture of the Reverse Engineering Part of Reengineering Assistant

the necessary data. In a knowledge system sense, they are the databases backing the inference machine. For the extracted ITL specification, further abstraction may also be applied to make it more concise, i.e., high level. During the abstraction process, the extracted specification is pretty printed in the browser-interface with the ITL displayer, and the process information is displayed in the LISP dialogue window.

CSL/COOL programs are at source code level, and ITL specifications are at the specification level. The program abstracter is an inference machine to cross various abstraction levels with interactions. In the next subsections of this chapter, we will discuss the reverse engineering parts of RA part by part.

7.2 Embedding CSL and COOL in LISP

7.2.1 Syntax Check

The syntax of CSL and COOL is defined formally in chapter 5. In RA, the syntax of CSL and COOL code needs to be checked to assure its correctness. The syntax is checked in the following situations:

- when a CSL/COOL program in text format is loaded into RA.
- when RA performed any transformation on the loaded CSL/COOL program and redisplays the new CSL/COOL program in the interface.

Lex and Yacc techniques are used to generate the syntax scanner and parser in C code automatically. Before this generation, the syntax of CSL/COOL need to be defined in corresponding Lex and Yacc format. Here are some examples.

To identify keywords from text program with the scanner generated by Lex, the keywords need to be defined in the following program segment as *input* of Lex:

```
class           { return class_; }
CLASS           { return class_; }
read            { return read_; }
READ            { return read_; }
delay           { return delay_; }
DELAY           { return delay_; }
parbegin        { return parbegin_; }
PARBEGIN        { return parbegin_; }
parend          { return parend_; }
PAREND          { return parend_; }
parallel        { return parallel_; }
PARALLEL        { return parallel_; }
... ...
```

To parse the syntax of *signal* statement, which is shown below, the syntax need to be described in a program segment as input to Yacc. The syntax of *signal* statement is as follows:

wait on s **for** t **do**

$\mathcal{A}$

else $\triangleq \mathcal{A} \trianglerighteq_s^t \mathcal{B}$

$\mathcal{B}$

end

And the corresponding Yacc input segment could be the following:

```
| wait_ on_                  { cat("(Signal "); }
  variable                   { tmp = pop(); cat (tmp); cat(" ");}
  for_
  expn                       { tmp = pop(); cat (tmp); }
  do_                        { cat(" ("); }
  stmnts
  else_                      { cat(") ("); }
  stmnts
  end_                       { cat(") )"); }
```

Here, we assume that the syntax of *variable*, *stmnts*(statements) and *expn*(expression) is already defined. The left column gives the syntax rules in sequence. The text in { } in the right column defines the actions to take once the syntax rules are matched. In the above example, the action is to construct the external LISP representation of *signal* statement.

7.2.2 CSL/COOL LISP Database

In RA, RWSL is internally represented as a syntax tree and is expressed, in a LISP style, as a series of nested lists. This representation, together with additional internal information, constructs the LISP Database of RWSL, including both code level CSL/COOL

and specification level ITL. To the user, RWSL is always presented by the browser-interface in a easy-to-read form, that is, in a Pascal/Java-style text for CSL/COOL, and in logic formulae for ITL.

In this section, we focus on the code level, CSL/COOL. Consider the duration statement as an example. In the text form, it could be as follows:

```
duration  t1+t2 in
    x:=a*b
end
```

The statement could be represented in a syntax tree shown in Figure 7.3. With proper pre-defined procedures, this tree structure can be easily traversed and changed. Since both ITL at specification level and CSL/COOL at code level are represented in syntax tree structures, abstraction and transformation can be implemented easily on this tree structure.

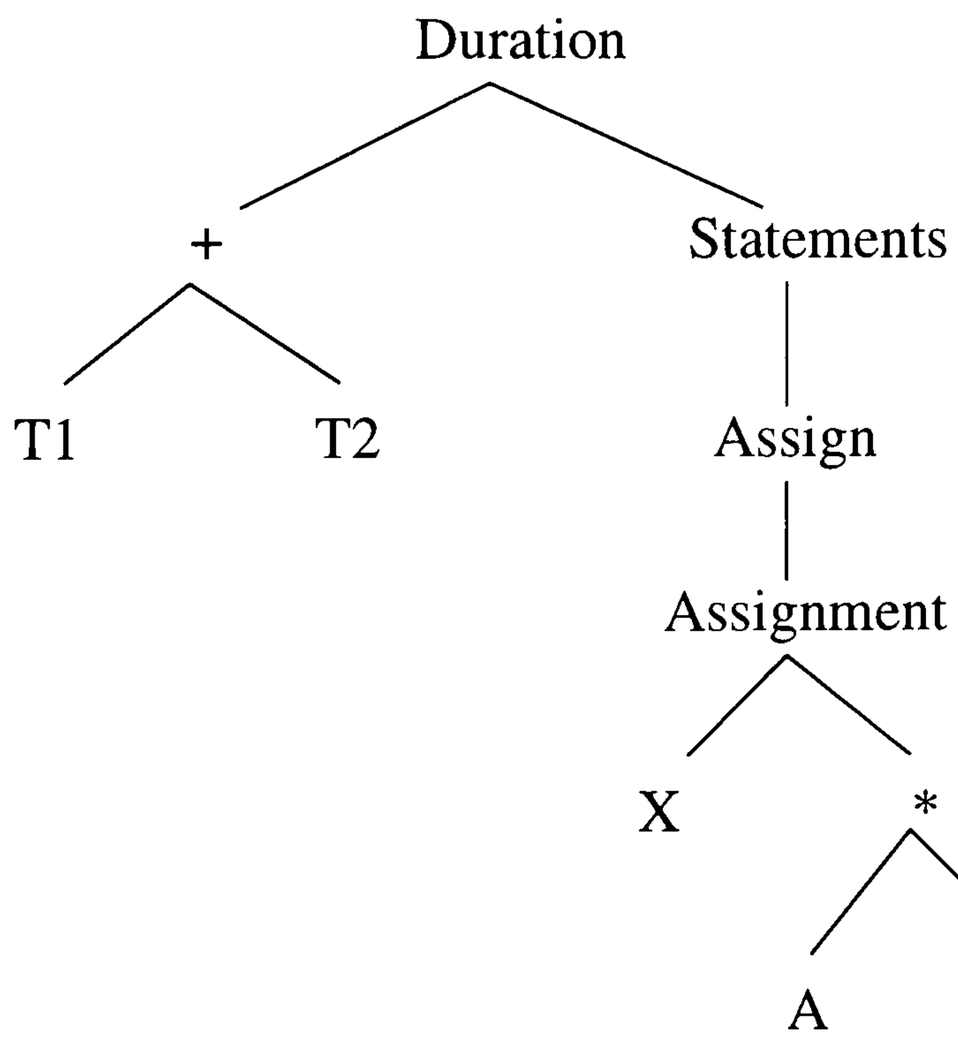

Figure 7.3: Syntax Tree Form of RWSL Duration Statement.

In CSL/COOL database, two forms of the above syntax tree are used. The first one, namely the *internal form*, stores at each node additional information, such as its

database table. The internal form is the main structure of CSL/COOL database, and is used to perform abstraction and transformation. The second form, namely the *external form*, is "LISP-like". It omits the extra information so that programs in this form can be executed via a number of macro and function definitions. The external form is more easy to understand and check, it is an abstract form of CSL/COOL database. All programs being reengineered in RA are stored using an abstract data type which implements the first form.

For the above example, its external LISP form is as follows:

(DURATION (+ T1 T2) ((ASSIGN (X (+ A B)))))

And its internal form appears as follows:

```
((__ NIL 152) (|| NIL) (DURATION STATEMENT)
 ((__ NIL 153) (|| NIL) (+ EXPRESSION)
  ((__ NIL 154) (|| NIL) (VARIABLE EXPRESSION T1))
  ((__ NIL 155) (|| NIL) (VARIABLE EXPRESSION T2)))
((__ NIL 156) (|| NIL) (STATEMENTS STATEMENTS)
 ((__ NIL 157) (|| NIL) (ASSIGN STATEMENT)
  ((__ NIL 158) (|| NIL) (ASSIGNMENT ASSIGNMENT)
   ((__ NIL 159) (|| NIL) (VARIABLE ASSD_VAR X))
   ((__ NIL 160) (|| NIL) (* EXPRESSION)
    ((__ NIL 161) (|| NIL) (VARIABLE EXPRESSION A))
    ((__ NIL 162) (|| NIL) (VARIABLE EXPRESSION B))))))
```

7.2.3 Pretty Print Display

Using X Window graphic functions, RA displays the stored CSL/COOL programs in the interface window in an indented format with various fonts. This gives the user a nice environment to view and reengineer the program. Figure 7.4 is a sample display.

```
                                STATEMENTS  STATEMENTS  NIL
comment: "Pump control";
proc pump_control(In Out
        ) {
        shunt : alarm;
        boolean : sw;
        integer : tm;
        while true do
                wait on alarm for 1 do
                        delay 0
                else read tm,signal from alarm;
                        if (signal=high_alarm) then duration in 1 sw:=1 end fi;
                        if (signal=low_alarm) then duration in 1 sw:=0 end fi end od }
```

Figure 7.4: A Sample Pretty Print Display of a CSL/COOL Program in the User Interface.

7.3 Embedding Interval Temporal Logic in LISP

7.3.1 Tree Structure and Stepwise Abstraction

As stated in the last subsection, ITL, as the specification part of RWSL, is also represented in a syntax tree structure. Adopting the same representation structure of both code and specification levels facilities abstraction greatly, because this ensures consistency between code and specification. Moreover, the syntax tree structure is easy for traversal, structural change and pattern matching. By using the tree structure for both code and specification levels, crossing levels of abstractions could be done on tree structures with three main tree operations: traversal, change in tree structure and tree structure pattern matching.

Although the syntax tree structure of a source code language can be obviously obtained from its formal syntax definition, representation of a logic in tree structure needs some study.

The basic elements in ITL are *terms*. A term is either a variable symbol or the application of a function symbol of n arguments to n terms. Terms are composed into *formulae* with operators. We classify ITL formulae into two categories: primitive formulae and composite formulae. A primitive formula has no sub-formula, while a composite

formula has sub-formulae composed with operators.

Stepwise abstraction often requires us to slice a large system into sub-systems, then to abstract these sub-systems individually, and finally to integrate the specifications of each sub-system into a whole system specification. This requires that a system can be represented with a combination of both specification segments and code pieces.

In RA, a *specification statement* is defined to act as the junction between code and specification. A RWSL representation consists of RWSL statements, including both CSL/COOL statements and specification statements. A specification statement is composed of two parts: a *formula* which is a segment of system specification in ITL, and a *frame* which includes all variables that may possibly change in the formula. In this design, specification and source code could be combined in the same system representation. And therefore *stepwise abstraction* becomes possible.

The ITL syntax is defined in syntax tree structure as follows:

RWSL representation ::= statements

Statements ::= CSL/COOL statements | specification statements

Specification statement ::= {variables} : formula

Formula ::= primitive formula | composite formula

Primitive formula ::= empty | stable (*variable*) | *more* | *finite* | *inf* | *Skip*
| ○*expression* | *expression* = *expression*
| *expression* < *expression* | *expression* > *expression*
| *P*(*expressions*) | *fin expression* | *variable* := *expression*
| *function description* | read (*variable*)

Composite formula ::= *formula* ∧ *formula* | *formula* ∨ *formula* | ¬*formula*
| *formula* ; *formula* | *formula**
| ∀ *variables* • *formula* | ∃ *variables* • *formula*
| ○*formula* | □*formula* | ◇*formula*
| *halt formula* | *keep formula* | *fin formula* | *fstar formula*

7.3.2 ITL LISP Database

RA implements ITL as its specification part. Similar to CSL/COOL, based on the syntax tree structure defined in the last section, ITL is internally represented as a syntax tree and is expressed, in a LISP style, as a series of nested lists. This representation, together with additional internal information, constructs the LISP Database of ITL.

In this section, we demonstrate the method to embed ITL specification in a LISP database with an example. Consider the following ITL formula:

$$\exists X, Y \bullet (X > Y * Y + 62) \wedge (\bigcirc X = Y + 100)$$

The syntax tree of the above formula is shown in Figure 7.5.

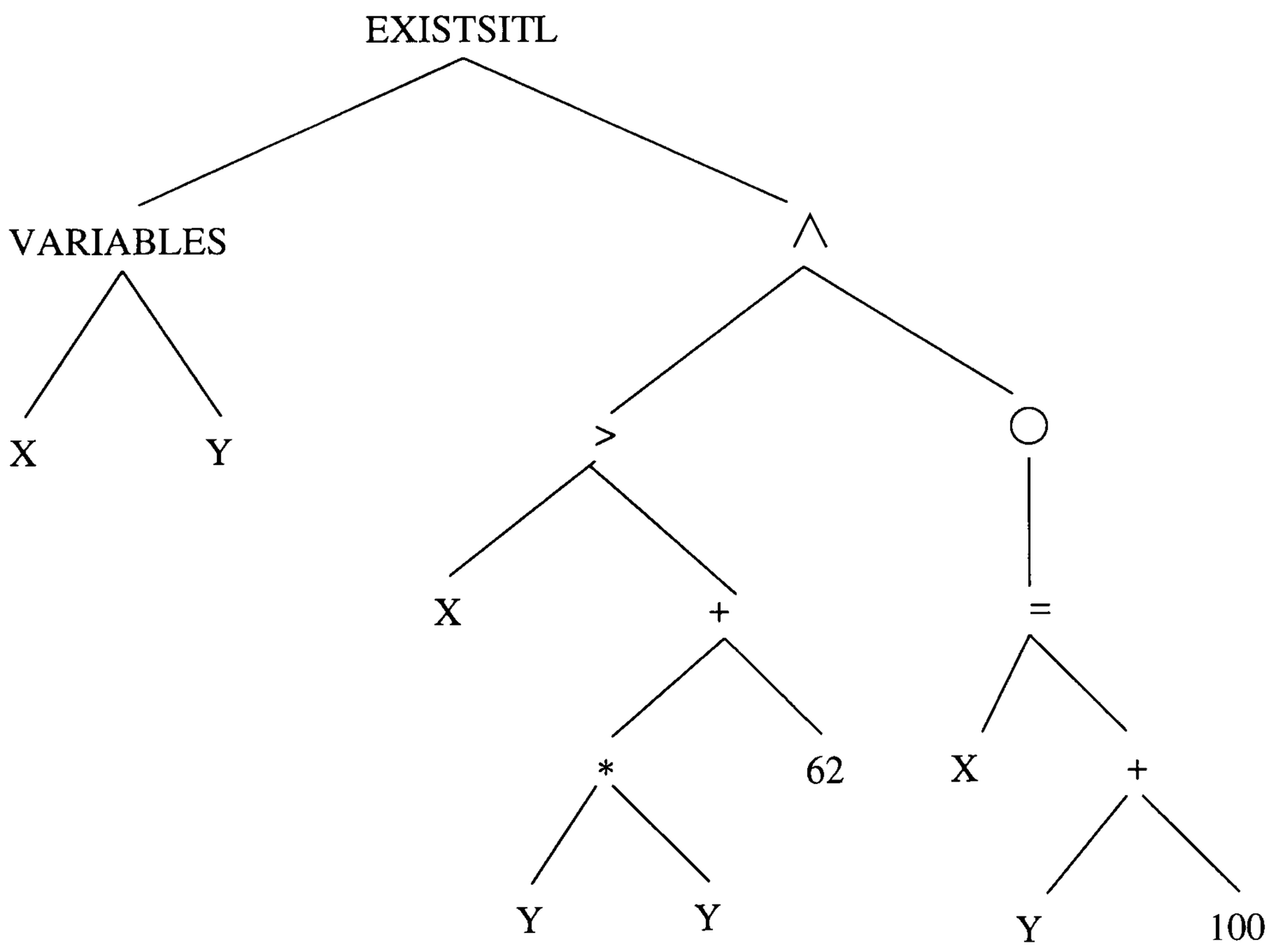

Figure 7.5: Syntax Tree Form of ITL Exists Formula.

In the ITL database, two forms of the above syntax tree are used. The first one, namely the *internal form*, stores at each node additional information, such as its database table. The internal form is the main structure of the ITL database, and is used to perform abstraction and transformation. The second form, namely the *external form*, is

"LISP-like". It omits the extra information so that programs in this form can be executed via a number of macro and function definitions. The external form is more easy to understand and check as it is an abstract form of the ITL database. All specifications extracted with RA are stored using an abstract data type which implements the first form.

For the above example, its external form is as follows:

```
(EXISTS (X Y)  (WEDGE (LARGEEXP X (+ (* Y Y) 62))
                      (NEXTF (EQUALEXP X (+ Y 100))) ) )
```

And its internal form appears as follows:

```
((__ NIL 6) (|| NIL) (EXISTSITL FORMULA)
 ((__ NIL 7) (|| NIL) (VARIABLES VARIABLES)
  ((__ NIL 8) (|| NIL) (VARIABLE VARIABLE X))
  ((__ NIL 9) (|| NIL) (VARIABLE VARIABLE Y)))
 ((__ NIL 10) (|| NIL) (WEDGE FORMULA)
  ((__ NIL 11) (|| NIL) (LARGEEXP FORMULA)
   ((__ NIL 12) (|| NIL) (VARIABLE EXPRESSION X))
   ((__ NIL 13) (|| NIL) (+ EXPRESSION)
    ((__ NIL 14) (|| NIL) (* EXPRESSION)
     ((__ NIL 15) (|| NIL) (VARIABLE EXPRESSION Y))
     ((__ NIL 16) (|| NIL) (VARIABLE EXPRESSION Y)))
    ((__ NIL 17) (|| NIL) (NUMBER EXPRESSION 62))))
  ((__ NIL 18) (|| NIL) (NEXTF FORMULA)
   ((__ NIL 19) (|| NIL) (EQUALEXP FORMULA)
    ((__ NIL 20) (|| NIL) (VARIABLE VARIABLE X))
    ((__ NIL 21) (|| NIL) (+ EXPRESSION)
     ((__ NIL 22) (|| NIL) (VARIABLE EXPRESSION Y))
     ((__ NIL 23) (|| NIL) (NUMBER EXPRESSION 100)))))))
```

7.3.3 Pretty Print Display

To facilitate further abstraction, re-design, re-specification and forward engineering, ITL Specifications stored in the ITL LISP database are displayed in the user interface window in an indented format with various fonts, which we call "pretty print". Compared with CSL/COOL, ITL has special temporal logic symbols, such as ○,□, ∀, ∃, etc. We decided to use these symbols in RA because they make the specification more concise and consistent to ITL. These symbols are represented with bitmaps and then loaded into the interface window at precise location. Figure 7.6 is a sample display.

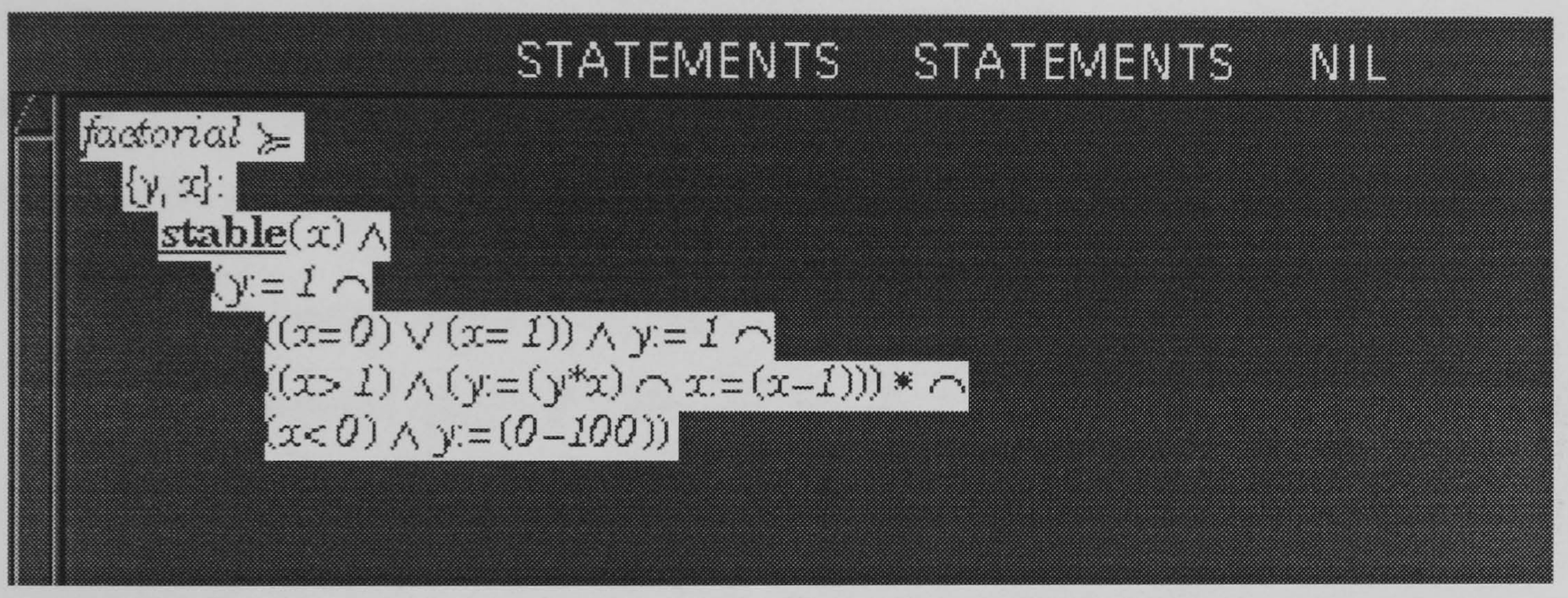

Figure 7.6: A Sample Pretty Print Display of ITL Specification in the User Interface.

7.4 Realisation of Elementary Abstraction Rules

7.4.1 Constructing the Catalogue

The Reengineering Assistant incorporates a large number of abstractions, which are classified and built in different categories. Most generally, abstractions in RA fall into two major groups: *elementary abstractions* and *further abstractions*. As stated in section 6.6, elementary abstraction rules aim at abstracting CSL/COOL statements to formulae in ITL, which is a specification. With further abstraction rules, logic composition

and semantics weakening will be applied to abstract these formulae to extract a more concise and abstract specification.

In this section, we focus on the implementation of elementary rules. Two relevant categories of abstractions are built in RA, namely, *primitive abstractions* and *compound abstractions*. The first category includes all the abstractions dealing with simple statements in CSL/COOL, and the second category consists of abstractions about composite statements.

The primitive abstraction category is divided into detailed abstractions depending on the statements dealt with, which are assignment abstraction, type declaration abstraction, delay abstraction, input abstraction and output abstraction. The composite abstraction category is also divided into detailed abstractions according to the statements dealt with. The detailed abstractions includes the following:

- General control structure abstractions, including sequential composition, conditional statement, iteration statements (while, for, etc.), procedure invocation, and external procedure invocation.

- Real-time structure abstractions, including parallel statement, signal statement, and duration statement.

- Definition abstractions, including procedure definition, function definition, class definition, method definition and hierarchy definition.

The user can select to work with any or all of these abstractions on the source code program.

7.4.2 Inference Process

Once an abstraction category is selected, RA will test the applicability of all the abstractions in this category. The valid abstractions will be returned and displayed as sub-menu of the category in the interface. To do this, each abstraction needs to have

its distinct "applicability condition" and have the condition coded. Here we define abstraction applicability condition as follows:

> An abstraction's **applicability condition** is the test which determines whether the particular abstraction can be legitimately applied (i.e., suitable for the program/system's situation and liable for making progress in abstraction) at the currently selected point in the program.

A pattern matching technique is used the in applicability test. Each abstraction is applicable to certain program/system situations, which are represented as a "situation pattern". If a situation pattern is identified as matching the current selected program, then the related abstraction is considered feasible. Pattern matching is carried out with LISP inference code and data from the LISP database of the current program.

If an abstraction is tested as applicable and then confirmed by the user, RA will perform it as the "action" part of the inference rule. As the first step, basic informations of the processed program segment are extracted from the CSL/COOL LISP database, such as name of variables, expressions, sub-statements, etc. Then, with these basic informations and inference action logic, RA constructs the new specification, which is in ITL LISP database format. Each abstraction has its own inference action logic, which is embedded in its LISP code.

Once the construction of the new specification finished, the ITL LISP database is updated with the new value. And the new RWSL program, including the generated specifications, is displayed in pretty print format.

During the whole abstraction process, the operation information, including error message if any error happens, is echoed in the LISP operation window so that the user could monitor the system's operation and make correct decisions.

Obviously, the LISP CSL/COOL database and ITL database play important roles during the whole abstraction process.

Examples Consider the delay statement as an example of primitive abstractions. Its applicability test code is as follows:

```
(Defun Do_What_Abstr (Type)
 (Cond
   ( (Eq Type 'PRIMITIVE-ABSTRACTION)
   (Eval (Expand_Trans_Patterns
     '(Let ((Ok_Trans Nil))
      (Cond
        ... ...
       (([_Check?_] Statement (Delay (~?~)) )
        (Setq Ok_Trans (Cons 'Delay_Statement Ok_Trans)))
        ... ...
      )

      (Funcall 'Print_Abstr Ok_Trans) ) )
   )
)
```

And its inference action code is as follows:

```
(Defun Abstr_Delay_Stat (&Optional Data)
 (Var ((Token Nil) (OP Nil) (Varlist Nil) (Result Nil))
  (Setq Token Data)
  (Setq OP
    '(Var ((Table ([_Match_] Statement
            (Delay (~>?~ Tm))  Empty)))

      ;; If this item is abstracted to a specification statement,
      ;;  then prepare its frame part:
      (Cond
       ((Eq Token 'Sepa)
        (Setq Varlist ([_Variables_] %Item%))
        (Setq Table ([_Put_] Frame (Generate_Frame Varlist) Table))
       )
      )

      (Setq Table ([_Put_] Con
                   '((__ NIL 0) (|| NIL) (VARIABLE VARIABLE LEN))
                   Table) )
(Setq Table ([_Put_] Formu
                   ([_Fill_In_] Formula
                                (EqualExp (~<?~ Con) (~<?~ Tm) )
                                Table)
                   Table) )

;; Abstract to a specification statement or a formula:
(Cond
```

```
      ((Eq Token 'Sepa)
       (@Change_To ([_Fill_In_] Statement
                    (Specification (~<?~ Frame) (~<?~ Formu)) Table)))
       ((Else)
        (Setq Result ([_Get_] Formu Table)) )
       )
   )
)
(Eval (Expand_Trans_Patterns OP))
(Return-from  Abstr_Delay_Stat Result)
 )
)
```

For the compound abstractions, we use the "while iteration" statement as an example. The applicability test code is as follows:

```
 ( (Eq Type 'COMPOSITE-ABSTRACTION)
   (Eval (Expand_Trans_Patterns
     '(Let ((Ok_Trans Nil) (Sts Nil))
        (Cond
         (([_Check?_] Statement (While (~?~) (~*~)))
          (Setq Ok_Trans (Cons 'While_Statement Ok_Trans)))
             ... ...
        )
      )
```

And the inference action code is as follows:

```
(Defun Abstr_While_Stat (&Optional Data)
 (Var ((Token Nil) (OP Nil) (Varlist Nil) (Loop_Times Nil)
       (Buf Nil) (Fm Nil) (Result Nil))
  (Setq Token Data)
  (Setq OP
    '(Var ((Table ([_Match_] Statement
             (While (~>?~ Cd) (~>*~ Sts))  Empty))
           (Head (Car %Item%)) )

      ;; Store the original value of current while statement
      (Setq Buf %Item%)

      ;; If this item is abstracted to a specification statement,
      ;;  then prepare its frame part:
      (Cond
        ((Eq Token 'Sepa)
         (Setq Varlist ([_Variables_] %Item%))
         (Setq Table ([_Put_] Frame (Generate_Frame Varlist) Table))
        )
```

```
)

;; Transform the condition in CSL/COOL to a formula in ITL.
(@Down)
(Setq Table ([_Put_] CF (Cond_To_Formu) Table) )
(@Up)

;; Recover the current while statement to the original value.
(@Change_To Buf)

;; Process the sub-statements one by one.
(Setq Loop_Times 'First)
(@When 0 ((And ([_G_Type?_] Statement) (Not (Eq (car %Item%) Head)))
 (Setq Fm (Abstr_Stats) )
 (Cond ( (not (Equalp Fm 'Ignore))
  (Setq Table ([_Put_] Formu1 Fm Table) )
  (Cond ((Eq Loop_Times 'First)
         (Setq Table ([_Put_] Formu2 ([_Get_] Formu1 Table) Table))
         (Setq Loop_Times 'Second))
        ((Eq Loop_Times 'Second)
         (Setq Table ([_Put_] Formu2 (List ([_Get_] Formu2 Table)
                                           ([_Get_] Formu1 Table)) Table))
        (Setq Loop_Times 'Many) )
        ((Eq Loop_Times 'Many)
         (Setq Table ([_Put_] Formu2 (Append ([_Get_] Formu2 Table)
                                (List ([_Get_] Formu1 Table))) Table)))
  ) )
 ) )
)

(Cond ((Eq Loop_Times 'Many)
       (Setq Table ([_Put_] Formu3
                            ([_Fill_In_] Formula
                                  (Chop ((~<*~ Formu2)))
                                  Table)
                    Table) )
      )
      ((Else)
       (Setq Table ([_Put_] Formu3
                            ([_Get_] Formu2 Table)
                            Table) )
      )
)
)
(Setq Table ([_Put_] Formu
                     ([_Fill_In_] Formula
                       (Iteration (Wedge (~<?~ CF) (~<?~ Formu3)) )
                       Table)
                     Table) )

;; Abstract to a specification statement or a formula:
(Cond
  ( (Eq Token 'Sepa)
    (@Change_To ([_Fill_In_] Statement
```

```
                    (Specification (~<?~ Frame) (~<?~ Formu)) Table)))
      ( (Else)
        (Setq Result ([_Get_] Formu Table)))
    )

  )
)
(Eval (Expand_Trans_Patterns OP))
(Return-from Abstr_While_Stat Result)
  )
)
```

7.5 Realisation of Further Abstraction Rules

7.5.1 Abstraction Patterns

Abstraction patterns are introduced as a means of acquiring observations of the legacy system identified by software engineers. Abstraction patterns are classified into various groups according to abstraction situations, which have been discussed in section 6.7. These observations are then embedded in RWSL representation as *abstraction pattern assertions*.

RA adopts the following two-step process of using abstraction patterns.

1. Identifying Abstraction Patterns. This step lets the user express his observations about the current abstraction situation. Based on the diversity of abstraction situations and relevant further abstraction rules, RA includes the following abstraction patterns:

    - State Test and Exception Handling
    - User Interface Format
    - Semantic Core
    - Concise Specification
    - Trivial Elements
    - Domain Function

- Efficiency-Improving Details

Each abstraction pattern category may be classified further into sub abstraction patterns, for example, "State Test and Exception Handling" includes several sub patterns, namely, state test and exception handling branch, state test and exception handling loop, state test and exception handling component, state test and exception handling expression, state test and exception handling variable. And "Domain Function" includes regular domain function and irregular domain function as its sub abstraction patterns.

2. Commitment of Abstraction Patterns. This step performs further abstractions according to the identified abstraction patterns. This process is carried out automatically, no user intervention is needed.

7.5.2 Constructing the Catalogue

Each instance of abstraction rules is implemented as a distinct abstraction. Further abstractions help in extracting more high-level specifications from the preliminary specifications obtained directly from the elementary abstractions. Based on the further abstraction rules discussed in section 6.7, further abstractions in RA are classified into the following categories:

- State Test and Exception Handling. This category abstracts away the identified state test and exception handling details.

- User Interface Format. This category abstracts away the identified user interface format details.

- Semantic Core. This category abstracts the specification to the identified semantic core.

- Concise Specification. This category abstracts the specification to the identified concise specification.

- Trivial Elements. This category abstracts away the identified trivial elements details.

- Domain Function. This category abstracts the specification by introducing the domain function.

- Efficiency-Improving Details. This category abstracts away the identified efficiency-improving details.

- Sequence Folding. This category changes sequential composition to conjunctive relations.

- Specification Combination. This category combines two separate ITL specifications into one specification. It includes two sub categories, namely, conjunctive combination and disjunctive combination.

- Comment Revision. This category revises comments to keep it consistent with new specifications.

7.5.3 Inference Process

Once an further abstraction category is selected, similar to elementary abstractions, RA will test the applicability of all the sub further abstractions in this category. The valid abstractions will be returned and displayed as sub-menu of the category in the interface. To do this, each abstraction needs to have its distinct "applicability condition" and have the condition coded. For example, to apply the sequence folding abstraction, there must be at least one sequential composition in the current system representation.

If an abstraction is tested as applicable and then confirmed by the user, RA will perform it as the "action" part of the inference rule. Different from elementary abstractions, in further abstractions RA will extract necessary information about user observations from abstraction pattern assertions, or under extreme situations, popup dialogue

windows to acquire this kind of information. Basic information, such as system structure, variable names, etc., will be extracted by pattern matching from the RWSL system representation automatically.

For example, when using domain function abstraction, although the domain function's position and the category of abstraction pattern could be decided by RA atomically, the details of the domain function need to be acquired as user observation from the corresponding abstraction pattern assertion, which was embedded in RWSL when the pattern was identified.

In the next step, with the automatically extracted basic informations, the user observations and built-in inference action logic, RA constructs the new specification, which is in ITL LISP database format. Each abstraction has its own inference action logic, which is embedded in its LISP code.

Once the construction of the new specification finished, the ITL LISP database is updated with the new value. And the new RWSL program, including the generated specifications, is displayed in pretty print format.

During the whole abstraction process, the operation information, including error message if any error happens, is echoed in the LISP operation window so that the user could monitor the system's operation and make correct decisions.

The ITL database plays important roles in further abstraction processes, but the CSL/COOL LISP database is rarely used.

Examples Consider the introduction of abstraction pattern "Regular Domain Function". RA will first check the feasibility of introducing a domain function at the current representation position, where the item should be a formula, a specification or an abstracted component in ITL. If it is feasible, then acquire the domain function's name and parameters through interaction with the software engineer. Then, construct the internal LISP form of the abstraction pattern. In the last step, insert the abstraction pattern at the selected position, that is, the first place before the current specification statement.

The corresponding LISP code is as follows. Please note that the interaction part in

C is not listed.

```
(Defun Abstr_Regular_Domain_Function (Data)
  (Var ((NPlist Data) (OP Nil) (Pname Nil) (Contents Nil) (Psn1 Nil)
        (Psn2 Nil) (AbsPattern Nil) (NTop True) (Feasible Nil)
        (ExtPn Nil) (IntPn Nil) (ExtPl Nil) (IntPl Nil)
        (V Nil) (IV) (Int_Var) (Dfunction Nil))
  (Setq OP
    '(Var ((Table Nil))

     ;; Check the feasibility of introducing the Domain Function
     ;; abstraction pattern
      (Cond ( ([_G_Type?_] Formula)
              (Setq Feasible True)
            )
            ( ([_Check?_] Statement (Abstracted_Agent (~*~)) )
              (Setq Feasible True)
            )
            ( ([_Check?_] Statement (Specification (~*~)) )
              (Setq Feasible True)
            )
            ((Else)
             (Showln "Wrong situation, impossible to introduce a
                      Domain Function abstraction pattern assertion.
                      The current object should be a variable, an
                      expression, a condition branch, a loop or
                      a component. ")
               (return-from Abstr_Domain_Function Nil)
            )
    )

  ;; prepare the abstraction pattern's name in internal format:
    (Setq Pname (List 'NAME 'NAME 'Domain_Function))
    (Setq Pname (List '(__ NIL 0) '(|| NIL) Pname))

  ;; prepare the abstraction pattern's contents in internal format:
    (Setq Contents %Item%)

  ;; prepare the function name in internal format:
    (Setq V (Car NPlist))
    (Setq ExtPn (List 'NAME 'NAME V))
    (Setq IntPn (List '(__ NIL 0) '(|| NIL) ExtPn))
    (Setq Table ([_Put_] Pname IntPn Table))

  ;; prepare the parameter list in internal format:
    (Setq ExtPl (Cdr NPlist))
    (Dolist
      (V ExtPl)
      (Setq IV (List 'VARIABLE 'VARIABLE V))
      (Setq Int_Var (List '(__ NIL 0) '(|| NIL) IV))
      (Setq IntPl (Append IntPl (List Int_Var) ))
    )
```

```
  (Setq Table ([_Put_] Plist IntPl Table))


;; prepare the domain function, save it in Dfunction:
  (Setq Dfunction ([_Fill_In_] Formula
                    (FuncITL (~<?~ Pname) ((~<*~ Plist)) ) Table) )

;; insert the abstraction pattern assertion just ahead of the
;; beginning of the current specification or abstracted agent
;; statement:
   (Setq Psn1 %Posn%)
   (Loop
     (Setq Psn2 %Posn%)
     (Cond ( (Or ([_Check?_] Statement (Abstracted_Agent (~*~)) )
                 ([_Check?_] Statement (Specification (~*~)) ) )
               (Setq Feasible True)
               (return 'NTop))
           ((And (Else) (Not (Eq NTop Nil)))
            (Setq NTop (@Up))
           )
           ((And (Else) (Eq NTop Nil))
            (Setq Feasible Nil)
            (Showln "Wrong position, impossible to introduce an
                     abstraction pattern assertion.")
            (return 'NTop)
           )
       )
     )

     (Cond (Feasible
           (Setq AbsPattern (List '(__ NIL 0) '(|| NIL) '(ABSTRPATTERN
                                  STATEMENT) Pname Dfunction Contents))
           (Showln AbsPattern)
           (@Ins_Before AbsPattern)
           (@Goto Psn1)
           ) )


 ) )

 (Eval (Expand_Trans_Patterns OP))
)
```

Chapter 8

Case Studies

8.1 Introduction

Case studies have been experimented with the proposed approach and resulting prototype. Various legacy systems are considered, from sequential non-time system to real-time systems with parallelism and communication.

The lexical scanner system aims at testing the approach and tool's ability in dealing with sequential non-time system. The robot control system is a multiple-process application. The purpose of the task farming system is to demonstrate how the approach and tool deal with concurrency/parallel and communication. At the last section, a mine drainage system is used to demonstrate the real-time ability of the approach.

8.2 Lexical Scanner

8.2.1 Background

This case study demonstrates the application of the proposed approach to a common sequential non-timed system. The legacy system is a lexical scanner implemented in PASCAL [140]. Before processing it, we assume that the software engineer does not know the system's structure and function details at all. What available to him/her is

only the system's source code, which is given in the appendix.

8.2.2 Extracting the Specification

The PASCAL source code is first translated into RWSL, that is, CSL. The resultant code is given in the appendix.

We choose part of the lexical system for detailed discussion, namely, procedure *initialise* and *scanreal*. The extracted specification of the whole system is given in the appendix.

Module: initialise

The translated CSL code of procedure *initialise* is as follows:

```
proc initialise(Out linebuffer: linebufrec)
{
    int: i;

    linebuffer.echo := true;
    linebuffer.lineerror := false;
    linebuffer.linecount := 0;

    for i := 0 to maxcharsperline do
        linebuffer.line[i] := '';
        linebuffer.errorline[i] := errnone
    od;

    linebuffer.errorset := [];
    linebuffer.fileerror := false;
    linebuffer.endoffile := false;
    linebuffer.endofline := true;
    linebuffer.pnum := 0
};
```

There are two 0-leveled data items: *linebuffer* and *token*. *initialise* is a component at level 1. *i* is the only local data item of procedure *initialise*; therefore, one goal of the abstraction process is to hide *i* because it is implementation detail.

We start by isolating the effect of i:

```
proc initialise(Out linebuffer: linebufrec) ||
  {                                         ||
     int: i;                                ||
                                            ||
     linebuffer.echo:= true;                ||
     linebuffer.lineerror:= false;          ||
     linebuffer.linecount:= 0;              ||
     for i:= 0 to maxcharsperline do        || {i} : i := 0;
                                            || (line[i] :=' ' ∧
        linebuffer.line[i] := ' ';          ||  errorline[i] := errnone;
        linebuffer.errorline[i] := errnone  ||  i := i + 1
     od;                                    ||  )^maxcharperline
     linebuffer.errorset:= [];              ||
     linebuffer.fileerror:= false;          ||
     linebuffer.endoffile:= false;          ||
     linebuffer.endofline:= true;           ||
     linebuffer.pnum := 0                   ||
  }                                         ||
```

Here we abstract the possible sequential composition inside the for-loop away by replacing it by $\wedge$ (parallel composition).

We then record its effect by defining an auxiliary procedure/predicate *initline* at level 1 and eliminate all references to i in the program:

```
proc initialise(var linebuffer: linebufrec); ||
  {                                           ||
     int: i;                                  ||
                                              ||
     linebuffer.echo:= true;                  ||
     linebuffer.lineerror:= false;            ||
     linebuffer.linecount:= 0;                ||
     initline(Out linebuffer.line,;           || initline(line, errorline) ≙
        linebuffer.errorline)                 || {i} : i := 0; (line[i] :=' ' ∧ errorline[i] := erronone;
                                              ||                i := i + 1)^maxcharperline
     linebuffer.errorset:= [];                ||
     linebuffer.fileerror:= false;            ||
     linebuffer.endoffile:= false;            ||
     linebuffer.endofline:= true;             ||
     linebuffer.pnum := 0                     ||
```

```
  end                                                    ||
end;                                                     ||
```

Let ϕ represent an empty set, the final specification is as follows:

$$\mathbf{initline(line, errorline)} \triangleq \{i\} : i := 0;\ (line[i] :=' \ ' \wedge errorline[i] := erronone\ ;\ i := i + 1)^{maxcharperline}$$

$$\begin{aligned}\mathbf{initialise(linebuffer)} \succeq \{linebuffer\} :\ & linebuffer.echo := true \wedge linebuffer.linerror := false \wedge linebuffer.linecount := 0; \\ & initline(linebuffer.line, linebuffer.errorline); \\ & linebuffer.errorset := \phi \wedge linebuffer.fileerror := false \wedge linebuffer.endoffile := false \wedge \\ & linebuffer.endofline := true \wedge linebuffer.pnum := 0\end{aligned}$$

Module: scanreal

The translated CSL code of procedure *initialise* is as follows:

```
comment: " scan a real number with/without exponent";

proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        int: expo;
        real: fac;
        int: i;
        boolean: negexp;
        int: nexpo;
        real: r;
        int: scale;
        real: x;

        if debug then !p writeln('scanning real number') fi;
        token.class:= realconstant;

comment:"do integer part, overflow assumed impossible";
x:= 0.0;
expo:= 0;
for i:= linebuffer.pint to linebuffer.charptr-1 do
   x=x*10.0+ord(linebuffer.line[i])-ord('0')
od;

nexpo := linebuffer.charptr-linebuffer.pint;
scale := 0;
if linebuffer.ch ='.' then
```

```
getnextchar(Out linebuffer);
linebuffer.pfrac := linebuffer.charptr;
if numeric(linebuffer.ch) then
    while (numeric(linebuffer.ch)) do
        scale:=scale-1;
        x:=x*10.0+ord(ch)-ord('0');
        getnextchar(Out linebuffer)
    od
  else puterror(In errnodigit, linebuffer)
  fi;
  comment: "check if we must find first nonzero digit";
  if nexpo=0 then
      i:= linebuffer.pfrac;
      while linebuffer.line[i]= '0' do i:=i+1 od;
      nexpo := linebuffer.pfrac-i
  fi
fi; comment "fractional ch='.'";

comment: "do we have an exponent?";
if ch='e' then
  negexp := false;
  getnextchar(Out linebuffer);
  if ch='-' then
      negexp:=true;
      getnextchar(Out linebuffer)
else if ch='+' then getnextchar(Out linebuffer) fi
fi;

comment "build exponent";
if numeric(linebuffer.ch) then
    while numeric(linebuffer.ch) do
        expo:= expo*10+ ord(ch)-ord('0');
        getnextchar(linebuffer)
    od;

    comment: "adjust scale and nexpo";
    if negexp then
        scale:= scale-expo;
        nexpo:= scale-expo
    else
        scale:= scale+expo;
        nexpo:= scale+expo
    fi
  else puterror(In errexpochar, linebuffer)
  fi; comment "process numeric"
fi; comment "exponent"
```

```
comment: "compute 10**scale using right to left binary method";
if abs(nexpo)≤maxexponent then
   if scale<>0 then
        r:= 1.0;
        negexp:= scale<0;
        scale:= abs(scale);
        fac:= 10.0;
        while scale<>0 do
            if odd(scale) then r:=r*fac;
            fac:=sqr(fac);
            scale:= scale div 2
        od;
        if negexp then realvalue:= x/r
        else realvalue:= x*r
        fi
     else realvalue:=x fi
   else puterror(In errexposize, linebuffer)
   fi
};
```

scanreal takes *linebuffer* and *token* as global variables (level 0), and *expo*, *fac*, *i*, *negexp*, *nexpo*, *r*, *scale* and *x* as local variables which we will try to hide. Since the procedure is somewhat long, stepwise abstraction is used. We will first divide it into several sections and deal with them separately, and then combine the results into one complete specification. The section division is normally equal to program blocks because a block is normally a functional unit in a structured program. As an advantage of a wide spectrum language, specification and code could appear together in the system representation.

The variable declaration part and debug mode test part are abstracted to specification first:

```
comment: " scan a real number with/without exponent";

proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        ∃ expo, fac, i, negexp, nexpo, r, scale, x • int(expo) ∧ real(fac) ∧ int(i) ∧
              boolean(negexp) ∧ int(nexpo) ∧ real(r) ∧ int(scale) ∧ real(x) ∧

        debug ∧ writeln('scanning real number') ∧
```

token.class := *realconstant*;

```
comment:"do integer part, overflow assumed impossible";
x:= 0.0;
expo:= 0;
for i:= linebuffer.pint to linebuffer.charptr-1 do
   x=x*10.0+ord(linebuffer.line[i])-ord('0')
od;

nexpo := linebuffer.charptr-linebuffer.pint;
scale := 0;
if linebuffer.ch ='.' then
   getnextchar(Out linebuffer);
   linebuffer.pfrac := linebuffer.charptr;
   if numeric(linebuffer.ch) then
        while (numeric(linebuffer.ch)) do
            scale:=scale-1;
            x:=x*10.0+ord(ch)-ord('0');
            getnextchar(Out linebuffer)
        od
   else puterror(In errnodigit, linebuffer)
   fi;
   comment: "check if we must find first nonzero digit";
   if nexpo=0 then
        i:= linebuffer.pfrac;
        while linebuffer.line[i]= '0' do i:=i+1 od;
        nexpo := linebuffer.pfrac-i
   fi
fi; comment "fractional ch='.'";

comment: "do we have an exponent?";
if ch='e' then
   negexp := false;
   getnextchar(Out linebuffer);
   if ch='-' then
        negexp:=true;
        getnextchar(Out linebuffer)
else if ch='+' then getnextchar(Out linebuffer) fi
fi;

comment "build exponent";
if numeric(linebuffer.ch) then
     while numeric(linebuffer.ch) do
          expo:= expo*10+ ord(ch)-ord('0');
          getnextchar(linebuffer)
     od;
```

```
comment: "adjust scale and nexpo";
if negexp then
    scale:= scale-expo;
    nexpo:= scale-expo
else
    scale:= scale+expo;
    nexpo:= scale+expo
fi
  else puterror(In errexpochar, linebuffer)
  fi; comment "process numeric"
fi; comment "exponent"

comment: "compute 10**scale using right to left binary method";
if abs(nexpo)≤maxexponent then
  if scale<>0 then
      r:= 1.0;
      negexp:= scale<0;
      scale:= abs(scale);
      fac:= 10.0;
      while scale<>0 do
          if odd(scale) then r:=r*fac;
          fac:=sqr(fac);
          scale:= scale div 2
          od;
          if negexp then realvalue:= x/r
          else realvalue:= x*r
          fi
        else realvalue:=x fi
    else puterror(In errexposize, linebuffer)
    fi
};
```

In this case, we abstract variable type information into a specific function *var-declare* and only present the function in the high level specification, leaving the variable type details in a lower level specification for possible retrieval when needed. Obviously, debug message is purely for implementation. Therefore, these parts are abstracted away.

The result is as follows:

```
comment: " scan a real number with/without exponent";
```

$$\begin{aligned} &\textit{var-declare}(\textit{expo}, \textit{fac}, i, \textit{negexp}, \textit{nexpo}, r, \textit{scale}, x) = \\ &\qquad \exists\, \textit{expo}, \textit{fac}, i, \textit{negexp}, \textit{nexpo}, r, \textit{scale}, x \cdot \textit{int}(\textit{expo}) \wedge \textit{real}(\textit{fac}) \wedge \textit{int}(i) \\ &\qquad \wedge \textit{boolean}(\textit{negexp}) \wedge \textit{int}(\textit{nexpo}) \wedge \textit{real}(r) \wedge \textit{int}(\textit{scale}) \wedge \textit{real}(x) \end{aligned}$$

```
proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        var-declare(expo,fac,i,negexp,nexpo,r,scale,x)∧
        token.class := realconstant;

        comment:"do integer part, overflow assumed impossible";
        x:= 0.0;
        expo:= 0;
        for i:= linebuffer.pint to linebuffer.charptr-1 do
          x=x*10.0+ord(linebuffer.line[i]-ord('0')
        od;

        nexpo := linebuffer.charptr-linebuffer.pint;
        scale := 0;
        if linebuffer.ch ='.' then
          getnextchar(Out linebuffer);
          linebuffer.pfrac := linebuffer.charptr;
          if numeric(linebuffer.ch) then
              while (numeric(linebuffer.ch)) do
                  scale:=scale-1;
                  x:=x*10.0+ord(ch)-ord('0');
                  getnextchar(Out linebuffer)
              od
          else puterror(In errnodigit, linebuffer)
          fi;
          comment: "check if we must find first nonzero digit";
          if nexpo=0 then
              i:= linebuffer.pfrac;
              while linebuffer.line[i]= '0' do i:=i+1 od;
              nexpo := linebuffer.pfrac-i
          fi
        fi; comment "fractional ch='.'";

        comment: "do we have an exponent?";
        if ch='e' then
          negexp := false;
          getnextchar(Out linebuffer);
          if ch='-' then
              negexp:=true;
              getnextchar(Out linebuffer)
          else if ch='+' then getnextchar(Out linebuffer) fi
          fi;

          comment "build exponent";
          if numeric(linebuffer.ch) then
              while numeric(linebuffer.ch) do
```

```
    expo:= expo*10+ ord(ch)-ord('0');
    getnextchar(linebuffer)
od;

comment: "adjust scale and nexpo";
if negexp then
    scale:= scale-expo;
    nexpo:= scale-expo
    else
        scale:= scale+expo;
        nexpo:= scale+expo
    fi
  else puterror(In errexpochar, linebuffer)
  fi; comment "process numeric"
fi; comment "exponent"

comment: "compute 10**scale using right to left binary method";
if abs(nexpo)≤maxexponent then
  if scale<>0 then
      r:= 1.0;
      negexp:= scale<0;
      scale:= abs(scale);
      fac:= 10.0;
          while scale<>0 do
              if odd(scale) then r:=r*fac;
              fac:=sqr(fac);
              scale:= scale div 2
          od;
          if negexp then realvalue:= x/r
          else realvalue:= x*r
          fi
    else realvalue:=x fi
  else puterror(In errexposize, linebuffer)
  fi
};
```

Then we process the next block and find nothing to abstract away at this moment only change possible chop operators to logic conjunctions:

```
comment: " scan a real number with/without exponent";

proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        var-declare(expo,fac,i,negexp,nexpo,r,scale,x)∧
        token.class := realconstant;
```

```
comment:"do integer part, overflow assumed impossible";
x := 0.0 ∧ expo := 0 ∧ i := linebuffer.pint;
(x := x × 10.0 + ord(linebuffer.line[i]) − ord('0') ∧ i := i + 1)^(linebuffer.charptr−linebuffer.pint−1);

nexpo := linebuffer.charptr-linebuffer.pint;
scale := 0;
if linebuffer.ch ='.' then
getnextchar(Out linebuffer);
linebuffer.pfrac := linebuffer.charptr;
if numeric(linebuffer.ch) then
     while (numeric(linebuffer.ch)) do
          scale:=scale-1;
          x:=x*10.0+ord(ch)-ord('0');
          getnextchar(Out linebuffer)
     od
else puterror(In errnodigit, linebuffer)
fi;
comment: "check if we must find first nonzero digit";
if nexpo=0 then
     i:= linebuffer.pfrac;
     while linebuffer.line[i]= '0' do i:=i+1 od;
     nexpo := linebuffer.pfrac-i
   fi
fi; comment "fractional ch='.'";

comment: "do we have an exponent?";
if ch='e' then
   negexp := false;
   getnextchar(Out linebuffer);
   if ch='-' then
        negexp:=true;
        getnextchar(Out linebuffer)
   else if ch='+' then getnextchar(Out linebuffer) fi
   fi;

   comment "build exponent";
if numeric(linebuffer.ch) then
     while numeric(linebuffer.ch) do
          expo:= expo*10+ ord(ch)-ord('0');
          getnextchar(linebuffer)
     od;

     comment: "adjust scale and nexpo";
     if negexp then
          scale:= scale-expo;
          nexpo:= scale-expo
```

```
        else
            scale:= scale+expo;
            nexpo:= scale+expo
        fi
    else puterror(In errexpochar, linebuffer)
    fi; comment "process numeric"
fi; comment "exponent"

comment: "compute 10**scale using right to left binary method";
if abs(nexpo)≤maxexponent then
    if scale<>0 then
        r:= 1.0;
        negexp:= scale<0;
        scale:= abs(scale);
        fac:= 10.0;
        while scale<>0 do
            if odd(scale) then r:=r*fac;
            fac:=sqr(fac);
            scale:= scale div 2
        od;
        if negexp then realvalue:= x/r
        else realvalue:= x*r
        fi
        else realvalue:=x fi
    else puterror(In errexposize, linebuffer)
    fi
};
```

Then we apply domain knowledge, knowing the section processes "integer" part of the float number, so define it as a domain function to block its details from upper level. To save space, meanwhile we process next block, changing statements into formulae:

comment: " scan a real number with/without exponent";

$$dealInteger(linebuffer, x, expo) \triangleq \{i\} : \ x := 0.0 \wedge expo := 0 \wedge i := linebuffer.pint;$$
$$(x := x \times 10.0 + ord(linebuffer.line[i]) - ord('0') \wedge i := i + 1)^{linebuffer.charptr - linebuffer.pint - 1}$$

```
proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        var-declare(expo,fac,negexp,nexpo,r,scale,x)∧
        token.class := realconstant;

        comment:"do integer part, overflow assumed impossible";
        dealInteger(linebuffer,x,expo);
```

$nexpo := linebuffer.charptr - linebuffer.pint;\ scale := 0;$
$linebuffer.ch =' .' \wedge (getnextchar(linebuffer);\ linebuffer.pfrac := linebuffer.charptr;$
$(numeric(linebuffer.ch) = true \wedge$
$(numeric(linebuffer.ch) \wedge (scale := scale - 1;\ x := x \times 10.0 + ord(ch) - ord('0');$
$getnextchar(linebuffer)))^*$
$\vee(numefic(ch) = false \wedge puterror(errnodigit, linebuffer)));$
$nexpo = 0 \wedge (i := linebuffer.frac;\ (linebuffer.line[i] =' 0' \wedge i := i + 1)^*;$
$nexpo := linebuffer.pfrac - i));$

```
comment: ”do we have an exponent?”;
if ch='e' then
   negexp := false;
   getnextchar(Out linebuffer);
   if ch='-' then
        negexp:=true;
        getnextchar(Out linebuffer)
   else if ch='+' then getnextchar(Out linebuffer) fi
   fi;

   comment ”build exponent”;
   if numeric(linebuffer.ch) then
        while numeric(linebuffer.ch) do
            expo:= expo*10+ ord(ch)-ord('0');
            getnextchar(linebuffer)
        od;

        comment: ”adjust scale and nexpo”;
        if negexp then
            scale:= scale-expo;
            nexpo:= scale-expo
        else
            scale:= scale+expo;
            nexpo:= scale+expo
        fi
   else puterror(In errexpochar, linebuffer)
   fi; comment ”process numeric”
fi; comment ”exponent”

comment: ”compute 10**scale using right to left binary method”;
if abs(nexpo)≤maxexponent then
   if scale<>0 then
        r:= 1.0;
        negexp:= scale<0;
        scale:= abs(scale);
        fac:= 10.0;
        while scale<>0 do
```

```
        if odd(scale) then r:=r*fac;
        fac:=sqr(fac);
        scale:= scale div 2
    od;
    if negexp then realvalue:= x/r
    else realvalue:= x*r
    fi
else realvalue:=x fi
    else puterror(In errexposize, linebuffer)
    fi
};
```

Using state test and error handling pattern, we abstract away *puterror* branch; with domain knowledge, we know this block deals with "fractional" part of the float number, similarly, we block its details by defining a domain function:

comment: " scan a real number with/without exponent";

$dealFraction(linebuffer, nexpo, scale, x) \triangleq nexpo := linebuffer.charptr - linebuffer.pint \wedge scale := 0;$
$linebuffer.ch =' .' \wedge (getnextchar(linebuffer) \wedge linebuffer.pfrac := linebuffer.charptr;$
$(numeric(linebuffer.ch) = true \wedge$
$(numeric(linebuffer.ch) \wedge (scale := scale - 1 \wedge x := x \times 10.0 + ord(ch) - ord('0') \wedge$
$getnextchar(linebuffer)))^*;$
$nexpo = 0 \wedge (i := linebuffer.frac;\ (linebuffer.line[i] =' 0' \wedge i := i + 1)^*;$
$nexpo := linebuffer.pfrac - i));$

```
proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        var-declare(expo,fac, i, negexp, nexpo, r, scale, x)∧
        token.class := realconstant;

        comment:"do integer part, overflow assumed impossible";
        dealInteger(linebuffer, x, expo);

        dealFraction(linebuffer, nexpo, scale, x);

        comment: "do we have an exponent?";
        if ch='e' then
          negexp := false;
          getnextchar(Out linebuffer);
          if ch='-' then
              negexp:=true;
              getnextchar(Out linebuffer)
          else if ch='+' then getnextchar(Out linebuffer) fi
```

```
    fi;

    comment "build exponent";
    if numeric(linebuffer.ch) then
        while numeric(ch) do
            expo:= expo*10+ ord(ch)-ord('0');
            getnextchar(linebuffer)
        od;
        comment: "adjust scale and nexpo";
        if negexp then
            scale:= scale-expo;
            nexpo:= scale-expo
        else
            scale:= scale+expo;
            nexpo:= scale+expo
        fi
    else puterror(In errexpochar, linebuffer)
    fi; comment "process numeric"
fi; comment "exponent"

comment: "compute 10**scale using right to left binary method";
if abs(nexpo)≤maxexponent then
    if scale<>0 then
        r:= 1.0;
        negexp:= scale<0;
        scale:= abs(scale);
        fac:= 10.0;
        while scale<>0 do
            if odd(scale) then r:=r*fac;
            fac:=sqr(fac);
            scale:= scale div 2
        od;
        if negexp then realvalue:= x/r
        else realvalue:= x*r
        fi
    else realvalue:=x fi
        else puterror(In errexposize, linebuffer)
        fi
};
```

The last part of *scanreal* is divided into two sections, which are abstracted into two separate procedures *dealExponent* and *CalcuReal*. Local variables *i* is hidden in procedure *dealInteger*, *expo* in *dealExponent*, and *r*, *fac* in *calcuReal*, because they are only used in the corresponding procedure. And the relevant variable type information

in *var-declare* should also be distributed to the procedures where the variables are used.

The result is as follows:

comment: " scan a real number with/without exponent";

$dealExponent(linebuffer, nexpo, negexp, scale) \triangleq$
$\{expo\} : linebuffer.ch =' e' \wedge (negexp := false \wedge getnextchar(linebuffer);$
$(linebuffer.ch =' -' \wedge (negexp := true \wedge getnextchar(linebuffer)))$
$\vee(linebuffer.ch =' +' \wedge getnextchar(linebuffer));$
$numeric(linebuffer.ch) \wedge (numeric(linebuffer.ch) \wedge (expo := expo \times 10 + ord(ch) - ord('0');$
$getnextchar(linebuffer)))^*;$
$(negexp \wedge (scale := scale - expo \wedge nexpo := scale - expo))$
$\vee(\neg negexp \wedge (scale := scale + expo \wedge nexpo := scale - expo)));$

$calcuReal(token, scale, nexpo, negexp, x) \triangleq \{r, fac\} : abs(nexpo) <= maxexponent \wedge (scale <> 0 \wedge$
$(r := 1.0 \wedge negexp := scale < 0 \wedge scale := abs(scale) \wedge fac := 10.0;$
$(scale <> 0 \wedge (odd(scale) \wedge r := r * fac;\ fac := sqr(fac);\ scale := scale/2))^*;$
$(negexp \wedge realvalue := x/r) \vee (\neg negexp \wedge realvalue := x \times r))$
$\vee(scale = 0 \wedge realvalue = x)) \vee (abs(nexpo) > maxexponent \wedge puterror(errexposize, linebuffer))$

```
proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
        var-declare(negexp, nexpo, scale, x)∧
        token.class := realconstant;

        comment:"do integer part, overflow assumed impossible";
        dealInteger(linebuffer, x, expo);

        dealFraction(linebuffer, nexpo, scale, x);

        comment: "do we have an exponent?";
        dealExponent(linebuffer, nexpo, scale);

        comment: "compute 10**scale using right to left binary method";
        calcuReal(token, scale, x)
    };
```

In the last step, we abstract main procedure *scanreal* into specification:

comment: " scan a real number with/without exponent";

$scanreal(linebuffer, token) \triangleq \{negexp, nexpo, scale, x\} :$

$var\text{-}declare(negexp, nexpo, scale, x) \wedge token.class := realconstant;$
$dealInteger(linebuffer, x, expo)\ ;\ dealFraction(linebuffer, nexpo, scale, x);$
$dealExponent(linebuffer, nexpo, scale)\ ;\ calcuReal(token, scale, x)$

From the final specification, it is easy to see that the function of *scanreal* is to scan a real number with or without exponent and fraction. *scanreal* processes the integer part first, then the fractional part (if there is any), and then the exponent part (if there is any). In the last step, *scanreal* puts the three part together to form a real number. The details of relevant procedures are not present in the high level specification, as these procedures are treated as domain functions. In case that these details are needed, the software engineer can retrieve them at the lower level specification.

8.2.3 Summary

The lexical scanner is a typical sequential non-time software. The point of this case study is to decompose large procedures (monolithic) into reasonable sections and abstract them separately. This would improve the understandability of the system and the clarity of the extracted specification efficiently.

8.3 Robot Control System

8.3.1 Background

This case study is a multiple-process application [37]. The tele-operated robot is a tracked device which was originally developed for military use. It is driven by two motors, left and right. Both of these motors can move forwards and backwards. The robot is steered by moving one motor faster than the other.

From a control point of view, commands are issued to the motors via an operator joystick which issues integer values in the range 0...127 for forward motion (127 max. speed) and 0...-128 for reverse motion. It is possible to drive only one motor at a time, in such a case the robot will turn. The speed of the motors is directly proportional to

the value written to them.

The robot is equipped with 8 infra red sensors. These return an integer value in the range 0...255 depending on whether an obstacle is present or not. 0 indicates no obstacle, 255 indicates obstacle very near. The robot is operated normally with a threshold of around 100, above which the robot takes notice of the sensor readings, i.e., an obstacles of interest. At this point reactive control takes over from the manual control by moving the robot away from the obstacle until the 100 threshold is not set. The sensor positions are as follows: N, NE, E, SE, S, SW, W and NW, covering the body of the robot and shown in Figure 8.1.

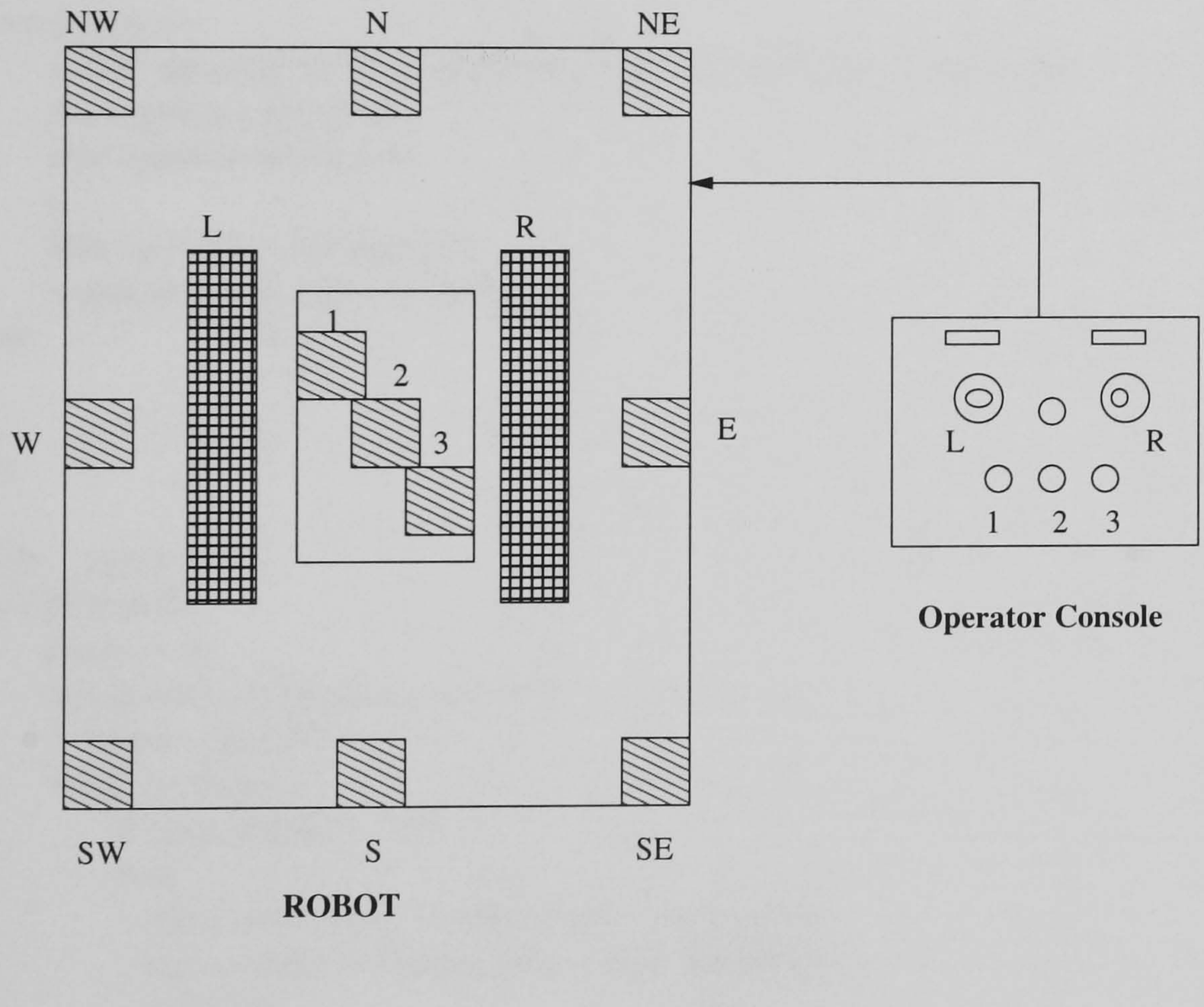

Figure 8.1: The Robot Control System

8.3.2 Extracting the Specification

CSL program translated from C code The robot control system was implemented in C. In order to extract a system specification of the robot control system, we first translate its source code into CSL as the basis for abstraction.

```
proc move(In left-op:int, right-op:int)
  {
        send-left-motor(left-op);
        send-right-motor(right-op);
        }

proc motor-control()
  {
        while true do
            if(ir-active=1)
            then move(left-ir-cmd, right-ir-cmd)
            else if(operator-active=1)
                  then move(left-op-cmd, right-op-cmd)
                  fi
            fi
        od
  }

proc operator()
  {
        while true do
            if((left-op-cmd<>lleft-op-cmd) & (right-op-cmd<>lright-op-cmd))
            then operator-active := 1
            else operator-active := 0
            fi;
            lleft-op-cmd := left-op-cmd;
            lright-op-cmd:= right-op-cmd;
        od
  }

proc ir()
  {
        int: i, count;
        while true do
            count := 0;
            left-ir-cmd := 0; right-ir-cmd := 0;
            ir-active := 0; i:=0;
            while (i<8) do
                  if (ir-counts(i) > 100)
                  then
                    left-ir-cmd := left-ir-cmd+motor-values[i][0];
                    right-ir-cmd := right-ir-cmd+motor-values[i][1];
                    count++
                  fi;
                  i:=i+1
            od
            if (count>0) then ir-active := 1 fi;
        od
  }

proc main()
  {
        int: left-ir-cmd, right-ir-cmd;
```

```
int: left-op-cmd, right-op-cmd;
int: lleft-op-cmd, lright-op-cmd;
int: ir-active, operator-active;
int: motor-values[8][2];

left-ir-cmd:=0; right-ir-cmd:=0;
left-op-cmd:=0; right-op-cmd:=0;
lleft-op-cmd:=0; lright-op-cmd:=0;
ir-active:=0; operator-active:=0;
motor-values[8][2]={{-20,-20}, {-20, 0}, {-20, 20},
        {0, 20}, {20,20}, {20, 0}, {20, -20}, {0, -20} };

parbegin
    motor-control()|| ir()|| operator()
parend;
}
```

System Specification in ITL At the first step, elementary abstraction rules are applied to each procedure in the program. Because the procedures are small enough, no section decomposition will be conducted. The result is as follows:

$$move(\textit{left-op}, \textit{right-op}) \succeq \textit{send-left-motor}(\textit{left-op}) \wedge \textit{send-right-motor}(\textit{right-op})$$

$$\begin{aligned}\textit{motor-control}() \succeq\ & ((\textit{ir-active} = 1 \wedge move(\textit{left-ir-cmd}, \textit{right-ir-cmd})) \\ & \vee(\textit{ir-active} \neq 1 \wedge \textit{operater-active} = 1 \wedge move(\textit{left-op-cmd}, \textit{right-op-cmd})))^*\end{aligned}$$

$$\begin{aligned}\textit{operator}() \succeq\ & \{\textit{left-op-cmd}, \textit{lleft-op-cmd}, \textit{right-op-cmd}, \textit{lright-op-cmd}, \textit{operator-active}\} : \\ & ((\textit{left-op-cmd} \neq \textit{lleft-op-cmd} \wedge \textit{right-op-cmd} \neq \textit{lright-op-cmd} \wedge \textit{operator-active} := 1) \\ & \vee(\textit{left-op-cmd} = \textit{lleft-op-cmd} \vee \textit{right-op-cmd} = \textit{lright-op-cmd} \wedge \textit{operator-active} := 0); \\ & (\textit{lleft-op-cmd} := \textit{left-op-cmd})\ ;\ (\textit{lright-op-cmd} := \textit{right-op-cmd}))^*\end{aligned}$$

$$\begin{aligned}\textit{ir}() \succeq\ & \{\textit{ir-active}, \textit{left-ir-cmd}, \textit{right-ir-cmd}, i, \textit{count}\} : \\ & (\textit{count} := 0\ ;\ \textit{left-ir-cmd} := 0\ ;\ \textit{right-ir-cmd} := 0\ ;\ \textit{ir-active} := 0\ ;\ i := 0; \\ & (i < 8 \wedge (\textit{ir-counts}(i) > 100 \wedge \textit{left-ir-cmd} := \textit{left-ir-cmd} + \textit{motor-value}[i][0]; \\ & \textit{right-ir-cmd} := \textit{right-ir-cmd} + \textit{motor-value}[i][1]\ ;\ \textit{count} := \textit{count} + 1)\ ;\ i := i + 1)^*; \\ & (\textit{count} > 0 \wedge \textit{ir-active} := 1))^*\end{aligned}$$

$$\begin{aligned}\textit{main}() \succeq\ & \{\textit{left-ir-cmd}, \textit{right-ir-cmd}, \textit{left-op-cmd}, \textit{right-op-cmd}, \textit{lleft-op-cmd}, \\ & \textit{lright-op-cmd}, \textit{ir-active}, \textit{operator-active}, \textit{motor-values}[8][2]\} : \\ & \textit{left-ir-cmd} := 0\ ;\ \textit{right-ir-cmd} := 0\ ;\ \textit{left-op-cmd} := 0\ ;\ \textit{right-op-cmd} := 0; \\ & \textit{lleft-op-cmd} := 0\ ;\ \textit{lright-op-cmd} := 0\ ;\ \textit{ir-active} := 0\ ;\ \textit{operator-active} := 0; \\ & \textit{motor-values}[8][2] := \{\{-20, -20\}, \{-20, 0\}, \{-20, 20\}, \{0, 20\}, \{20, 20\}, \{20, 0\}, \\ & \qquad \{20, -20\}, \{0, -20\}\}; \\ & \textit{motor-control}() \parallel \textit{ir}() \parallel \textit{operator}()\end{aligned}$$

Then we begin to do further abstraction to each procedure. For procedure *operator* sequence folding rule is applied to change chop into logic conjunction:

$$\begin{aligned}&operator() \succeq \{left\text{-}op\text{-}cmd, lleft\text{-}op\text{-}cmd, right\text{-}op\text{-}cmd, lright\text{-}op\text{-}cmd, operator\text{-}active\} :\\ &\quad ((left\text{-}op\text{-}cmd \neq lleft\text{-}op\text{-}cmd \wedge right\text{-}op\text{-}cmd \neq lright\text{-}op\text{-}cmd \wedge operator\text{-}active := 1)\\ &\quad \vee(left\text{-}op\text{-}cmd = lleft\text{-}op\text{-}cmd \vee right\text{-}op\text{-}cmd = lright\text{-}op\text{-}cmd \wedge operator\text{-}active := 0)\wedge\\ &\quad (lleft\text{-}op\text{-}cmd := left\text{-}op\text{-}cmd) \wedge (lright\text{-}op\text{-}cmd := right\text{-}op\text{-}cmd))^*\end{aligned}$$

Then use logic reasoning to make it more concise:

$$\begin{aligned}&operator() \succeq \{left\text{-}op\text{-}cmd, lleft\text{-}op\text{-}cmd, right\text{-}op\text{-}cmd, lright\text{-}op\text{-}cmd, operator\text{-}active\} :\\ &\quad (operator\text{-}active := (left\text{-}op\text{-}cmd \neq lleft\text{-}op\text{-}cmd \wedge right\text{-}op\text{-}cmd \neq lright\text{-}op\text{-}cmd)\wedge\\ &\quad (lleft\text{-}op\text{-}cmd := left\text{-}op\text{-}cmd) \wedge (lright\text{-}op\text{-}cmd := right\text{-}op\text{-}cmd))^*\end{aligned}$$

Procedure $ir()$ has two local variables, i and *count*. Since the loop time is fixed to 8, we change the chop-star into a concrete number 8. *count* could be left out by rewriting the specification in a more compact style since it is merely a boolean test. Sequence folding is done whenever possible.

$$\begin{aligned}&ir() \succeq \{ir\text{-}active, left\text{-}ir\text{-}cmd, right\text{-}ir\text{-}cmd, i, count\} :\\ &\quad (count := 0 \wedge left\text{-}ir\text{-}cmd := 0 \wedge right\text{-}ir\text{-}cmd := 0 \wedge ir\text{-}active := 0 \wedge i := 0;\\ &\quad ((ir\text{-}counts(i) > 100 \wedge left\text{-}ir\text{-}cmd := left\text{-}ir\text{-}cmd + motor\text{-}value[i][0]\wedge\\ &\quad right\text{-}ir\text{-}cmd := right\text{-}ir\text{-}cmd + motor\text{-}value[i][1] \wedge count := count + 1) \wedge i := i + 1)^8;\\ &\quad (count > 0 \wedge ir\text{-}active := 1))^*\end{aligned}$$

$$\begin{aligned}&ir() \succeq \{ir\text{-}active, left\text{-}ir\text{-}cmd, right\text{-}ir\text{-}cmd, i, count\} :\\ &\quad i \in [0,7] \wedge (ir\text{-}active = \bigvee_i (ir\text{-}counts(i) > 100)\wedge\\ &\quad left\text{-}ir\text{-}cmd = \sum_i ((ir\text{-}counts(i) > 100) * motor\text{-}values[i][0])\wedge\\ &\quad right\text{-}ir\text{-}cmd = \sum_i ((ir\text{-}counts(i) > 100) * motor\text{-}values[i][1]))^*\end{aligned}$$

For procedure $main()$, the initialisation part could be left out as trivial details, therefore we get a quite concise specification:

$$main() \succeq motor\text{-}control() \parallel ir() \parallel operator()$$

Putting together, the final specification is as follows:

$$move(\textit{left-op}, \textit{right-op}) \succeq \textit{send-left-motor}(\textit{left-op}) \wedge \textit{send-right-motor}(\textit{right-op})$$

$$\begin{aligned}\textit{motor-control}() \succeq\ & ((\textit{ir-active} = 1 \wedge move(\textit{left-ir-cmd}, \textit{right-ir-cmd})) \\ & \vee(\textit{ir-active} \neq 1 \wedge \textit{operater-active} = 1 \wedge move(\textit{left-op-cmd}, \textit{right-op-cmd})))^*\end{aligned}$$

$$\begin{aligned}\textit{operator}() \succeq\ & \{\textit{left-op-cmd}, \textit{lleft-op-cmd}, \textit{right-op-cmd}, \textit{lright-op-cmd}, \textit{operator-active}\} : \\ & (\textit{operator-active} := (\textit{left-op-cmd} \neq \textit{lleft-op-cmd} \wedge \textit{right-op-cmd} \neq \textit{lright-op-cmd}) \wedge \\ & (\textit{lleft-op-cmd} := \textit{left-op-cmd}) \wedge (\textit{lright-op-cmd} := \textit{right-op-cmd}))^*\end{aligned}$$

$$\begin{aligned}\textit{ir}() \triangleq\ & \{\textit{ir-active}, \textit{left-ir-cmd}, \textit{right-ir-cmd}, i, \textit{count}\} : \\ & i \in [0,7] \wedge (\textit{ir-active} = \bigvee_i (\textit{ir-counts}(i) > 100) \wedge \\ & \textit{left-ir-cmd} = \sum_i ((\textit{ir-counts}(i) > 100) * \textit{motor-values}[i][0]) \wedge \\ & \textit{right-ir-cmd} = \sum_i ((\textit{ir-counts}(i) > 100) * \textit{motor-values}[i][1]))^*\end{aligned}$$

$$\textit{main}() \succeq \textit{motor-control}() \parallel \textit{ir}() \parallel \textit{operator}()$$

8.3.3 Summary

The purpose of this case study is to test whether the proposed approach is capable of multiple concurrent processes without communication.

From *main*, it is clear that the robot system is composed of three concurrent processes, namely, *motor-control*, *ir* and *operator*. More details of the three processes are given in their own specification. Since the final specification is quite concise, it is easy to see the following points:

- From *motor-control*: if the control mode is infra-red, then move the robot through parameters *left-ir-cmd* and *right-ir-cmd*; if the control mode is operator, then move the robot through parameters *left-op-cmd* and *right-op-cmd*.

- From *operator*: if there is new operator command then set operator control mode active, and change the former command to current value.

- From *ir*: check the eight sensor, if any of them detects a nearby obstacle, then move the robot away from it.

The final specification will help software engineers understand the robot system.

8.4 Task Farming System

8.4.1 Background

This example is about a message-passing task farm. Its purpose is to demonstrate how the proposed approach deals with concurrency/parallel and communication [161]. In this task farm, every worker communicates directly with the source in order to get jobs and forward results. All workers run their dispatched tasks in parallel. Since the source has no way of telling when a worker has finished its job, and needs another, workers must send requests for more work to the source. These requests must be tagged in some way to identify the sender, so that the source knows where to send its reply.

8.4.2 Extracting the Specification

Translated CSL code The task farm is implemented in FORTRAN-KCSP. As preliminary process, we translated the FORTRAN-KCSP implementation into CSL.

```
struct message{
   integer: id;
   string: mbody;
   integer: sender;
   integer: receiver
   };

shunt: array connect[100];

proc root()
   {
       integer: id, i, j, t, numw;
       string: taskid;
       message: msg1, msg2;
       integer: array buffer[100];
```

```
comment:"setup";
id:=!xf getprocessid();
!xp mpsynch();
numw:=!xf mp-grp-size(workerGrp);
for i:=1 to 100 step 1 do
      buffer[i]:=-1
od;

comment:"assign tasks";
for i:=1 to numw step 1 do
      read t,msg1 from connect[i];
      if (msg1.id <> buffer[i]) and (msg1.receiver=id)
      then buffer[i]:= msg1.id;
          if msg1.mbody="finished"
          then !xp gettask(taskid);
                if taskid<0
                then msg2.mbody:="idle"
                else msg2.mbody:=!xf strcat("do",taskid)
                fi
          fi;
          if msg1.mbody="faulty"
          then msg2.mbody:="terminate"
          fi;
          msg2.id:=!xf gen-msg-id();
          msg2.sender:=id;
          msg2.receiver:=i;
          write msg2 to connect[i]
      fi
   od;

   comment:"tide up";
   msg1.id:=0; msg1.mbody:="
};

proc worker()
{
   integer:id, buffer;
   string: taskid, taskstate;
   message: msg1, msg2;
   integer: array buffer[100];

comment:"setup";
id:=!xf getprocessid();
!xp mpsynch();
buffer:=-1;

comment:"deal task";
read t,msg1 from connect[id];
if (msg1.id<>buffer) and (msg1.receiver=id)
then buffer:=msg1.id;
      if msg1.mbody="terminate"
      then !xp terminate(taskid)
```

```
        fi;
        if msg1.mbody="do xxxxx"
        then taskid:="xxxxx";
            !xp execute(taskid)
        fi;
        if msg1.mbody="idle"
        then skip fi;
    fi;

        comment:"sending message to root";
        taskstate:=!xf gettaskstate(taskid);
        if taskstate="finished"
        then msg2.mbody:="finished" fi;
        if taskstate="faulty"
        then msg2.mbody:="faulty" fi;
        if taskstate="running"
        then skip fi;
        msg2.id:=!xf gen-msg-id();
        msg2.sender:=id;
        msg2.receiver:=0;
        write msg2 to connect[id]
    };

proc main()
    {
        int: i;
        !xp createprocess(root());
        for i:=1 to W step 1 do
            !xp createprocess(worker())
        od;
    };
```

There are three procedures in the program, *root*, *worker* and *main*. We abstract the *main* procedure. The new system representation is as follows:

```
procroot()
    {
        comment:"setup";
        id:=!xf getprocessid();
        !xp mpsynch();
        numw:=!xf mp-grp-size(workerGrp);
        for i:=1 to 100 step 1 do
            buffer[i]:=-1
        od;

        comment:"assign tasks";
        for i:=1 to numw step 1 do
            read t,msg1 from connect[i];
            if (msg1.id <> buffer[i]) and (msg1.receiver=id)
```

```
then buffer[i]:= msg1.id;
  if msg1.mbody="finished"
  then !xp gettask(taskid);
       if taskid<0
       then msg2.mbody:="idle"
       else msg2.mbody:=!xf strcat("do",taskid)
       fi
  fi;
  if msg1.mbody="faulty"
       then msg2.mbody:="terminate"
       fi;
       msg2.id:=!xf gen-msg-id();
       msg2.sender:=id;
       msg2.receiver:=i;
       write msg2 to connect[i]
    fi
  od;

  comment:"tide up";
  msg1.id:=0; msg1.mbody:="
};

proc worker()
  {
  comment:"setup";
id:=!xf getprocessid();
!xp mpsynch();
buffer:=-1;

comment:"deal task";
read t,msg1 from connect[id];
if (msg1.id<>buffer) and (msg1.receiver=id)
then buffer:=msg1.id;
  if msg1.mbody="terminate"
  then !xp terminate(taskid)
  fi;
  if msg1.mbody="do xxxxx"
  then taskid:="xxxxx";
    !xp execute(taskid)
  fi;
  if msg1.mbody="idle"
  then skip fi;
fi;

comment:"sending message to root";
taskstate:=!xf gettaskstate(taskid);
if taskstate="finished"
then msg2.mbody:="finished" fi;
if taskstate="faulty"
then msg2.mbody:="faulty" fi;
if taskstate="running"
then skip fi;
msg2.id:=!xf gen-msg-id();
```

```
    msg2.sender:=id;
    msg2.receiver:=0;
    write msg2 to connect[id]
  };
```

$main() \triangleq \{i\} : createprocess(root())\ ;\ i := 1\ ;\ (createprocess(worker()))^{W}$

Since i is no longer used in the specification of *main*, it is omitted. *root* is divided into three sections, which are defined as domain functions in ITL. The first result looks like the following:

$RTsetup(id, numw, buffer) \triangleq \{i\} : id := getprocessid()\ ;\ mpsynch()\ ;\ numw := mp\text{-}grp\text{-}size(\text{“}worker\text{''});$
$i := 1\ ;\ (buffer[i] := -1\ ;\ i := i + 1)^{100}$

$assignTasks(numw, msg1, msg2, connect, id, buffer) \triangleq \{i, t, taskid\} : i := 1;$
$(t = \surd connect[i] \wedge msg1 = read(connect[i]);$
$msg1.id \neq buffer[i] \wedge msg1.receiver = id \wedge$
$(buffer[i] = msg1.id\ ;\ msg1.mbody = \text{`}finished\text{'} \wedge (gettask(taskid);$
$(taskid < 0 \wedge msg2.mbody := \text{`}idle\text{'}) \vee (taskid \geq 0 \wedge msg2.mbody := strcat(\text{`}do\text{'}, taskid);$
$msg1.mbody = \text{`}faulty\text{'} \wedge msg2.mbody := \text{`}terminate\text{'};$
$msg2.id = gen\text{-}msg\text{-}id()\ ;\ msg2.sender := id\ ;\ msg2.receiver := i;$
$skip \wedge connect[i] := (\surd connect[i] + 1, msg2)\ ;\ i := i + 1)))^{numw}$

$tideUp(msg1) \triangleq msg1.id := 0 \wedge msg1.mbody :=''$

```
procroot()
  {
    comment:"setup";
    RTsetup(id, numw);

    comment:"assign tasks";
    assignTasks(numw, msg1, msg2, connect, id, buffer);

    comment:"tide up";
    tideUp(msg1)
  };

proc worker()
  {
    comment:"setup";
    id:=!xf getprocessid();
    !xp mpsynch();
    buffer:=-1;

    comment:"deal task";
    read t,msg1 from connect[id];
    if (msg1.id<>buffer) and (msg1.receiver=id)
```

```
then buffer:=msg1.id;
   if msg1.mbody="terminate"
   then !xp terminate(taskid)
   fi;
   if msg1.mbody="do xxxxx"
   then taskid:="xxxxx";
      !xp execute(taskid)
   fi;
   if msg1.mbody="idle"
      then skip fi;
   fi;

   comment:"sending message to root";
   taskstate:=!xf gettaskstate(taskid);
   if taskstate="finished"
   then msg2.mbody:="finished" fi;
   if taskstate="faulty"
   then msg2.mbody:="faulty" fi;
   if taskstate="running"
   then skip fi;
   msg2.id:=!xf gen-msg-id();
   msg2.sender:=id;
   msg2.receiver:=0;
   write msg2 to connect[id]
};
```

$main() \triangleq createprocess(root())\ ;\ (createprocess(worker()))^{W}$

Since the timestamp t is never used in *assignTask*, we abstract it away. Meanwhile we conduct sequence folding, and abstract the frame of *root*.

$$RTsetup(id, numw, buffer) \triangleq \{i\} : id := getprocessid()\ ;\ mpsynch() \wedge numw := mp\text{-}grp\text{-}size(\text{“}worker''\text{)} \wedge$$
$$i := 1\ ;\ (buffer[i] := -1 \wedge i := i + 1)^{100}$$

$$assignTasks(numw, msg1, msg2, connect, id, buffer) \triangleq \{i, taskid\} : i := 1;$$
$$(msg1 = read(connect[i]);$$
$$msg1.id \neq buffer[i] \wedge msg1.receiver = id \wedge$$
$$(buffer[i] = msg1.id \wedge msg1.mbody = \text{‘}finished'' \wedge (gettask(taskid);$$
$$(taskid < 0 \wedge msg2.mbody := \text{‘}idle'') \vee (taskid \geq 0 \wedge msg2.mbody := strcat(\text{‘}do'', taskid);$$
$$msg1.mbody = \text{‘}faulty'' \wedge msg2.mbody := \text{‘}terminate'';$$
$$msg2.id = gen\text{-}msg\text{-}id()\ ;\ msg2.sender := id\ ;\ msg2.receiver := i;$$
$$connect[i] := (\sqrt{}connect[i] + 1, msg2)\ ;\ i := i + 1)))^{numw}$$

$$tideUp(msg1) \triangleq msg1.id := 0 \wedge msg1.mbody := ''$$

$$root() \triangleq \{numw, msg1, msg2, connect, id, buffer\} : RTsetup(id, numw);$$
$$assignTasks(numw, msg1, msg2, connect, id, buffer)\ ;\ tideUp(msg1)$$

```
proc worker()
  {
     comment:"setup";
     id:=!xf getprocessid();
     !xp mpsynch();
     buffer:=-1;

     comment:"deal task";
     read t,msg1 from connect[id];
if (msg1.id<>buffer) and (msg1.receiver=id)
then buffer:=msg1.id;
     if msg1.mbody="terminate"
     then !xp terminate(taskid)
     fi;
     if msg1.mbody="do xxxxx"
     then taskid:="xxxxx";
        !xp execute(taskid)
     fi;
     if msg1.mbody="idle"
     then skip fi;
fi;

comment:"sending message to root";
taskstate:=!xf gettaskstate(taskid);
if taskstate="finished"
then msg2.mbody:="finished" fi;
     if taskstate="faulty"
     then msg2.mbody:="faulty" fi;
     if taskstate="running"
     then skip fi;
     msg2.id:=!xf gen-msg-id();
     msg2.sender:=id;
     msg2.receiver:=0;
     write msg2 to connect[id]
  };
```

$main() \triangleq createprocess(root())\ ;\ (createprocess(worker()))^{w}$

To get a clear overview of the system, the details of procedure *RTsetup*, *assignTasks* and *tideUp* are then retained at the low level specification, not present in the high level specification.

The *worker* procedure is dealt with in a similar way. Three new procedures are introduced to describe the three blocks in *worker*, and these procedures are abstracted separately. The results are as follows:

$root() \triangleq \{numw, msg1, msg2, connect, id, buffer\} : RTsetup(id, numw);$

$assignTasks(numw, msg1, msg2, connect, id, buffer)\ ;\ tideUp(msg1)$

$WKsetup(id, buffer) \triangleq id := getprocessid()\ ;\ mpsynch()\ ;\ buffer := -1$

$dealTask(id, msg1, connect, taskid) \triangleq (t = \surd connect[id] \wedge msg1 = read(connect[id]));$
$(msg1.id \neq buffer \wedge msg1.receiver = id \wedge (buffer := msg1.id\ ;\ msg1.mbody = terminate \wedge terminate(taskid);$
$msg1.mbody = do\ xxxxx \wedge (taskid := xxxxx\ ;\ execute(taskid))\ ;\ msg1.mbody = idle \wedge skip))$

$sendMessage(taskid, taskstate, id, msg2) \triangleq taskstate := gettaskstate(taskid);$
$taskstate = finished \wedge msg2.mbody := finishedfi\ ;\ taskstate = faulty \wedge msg2.mbody := faulty;$
$taskstate = running \wedge skip\ ;\ msg2.id := gen\text{-}msg\text{-}id() \wedge msg2.sender := id \wedge msg2.receiver := 0;$
$skip \wedge connect[id] := (\surd connect[id] + 1, msg2)$

$worker() \triangleq \{id, buffer, taskid, taskstate, msg1, msg2, buffer\} : WKsetup(id, buffer);$
$dealTask(id, msg1, connect, taskid)\ ;\ sendMessage(taskid, taskstate, id, msg2)$

$main() \triangleq createprocess(root())\ ;\ (createprocess(worker()))^{w}$

Abstract away the unused timestamp, fold possible sequences, a more concise result looks like as follows:

$root() \triangleq \{numw, msg1, msg2, connect, id, buffer\} : RTsetup(id, numw);$
$assignTasks(numw, msg1, msg2, connect, id, buffer)\ ;\ tideUp(msg1)$

$WKsetup(id, buffer) \triangleq id := getprocessid()\ ;\ mpsynch()\ ;\ buffer := -1$

$dealTask(id, msg1, connect, taskid) \triangleq msg1 = read(connect[id]);$
$(msg1.id \neq buffer \wedge msg1.receiver = id \wedge (buffer := msg1.id\ ;\ msg1.mbody = terminate \wedge terminate(taskid);$
$msg1.mbody = do\ xxxxx \wedge (taskid := xxxxx\ ;\ execute(taskid))\ ;\ msg1.mbody = idle \wedge skip))$

$sendMessage(taskid, taskstate, id, msg2) \triangleq taskstate := gettaskstate(taskid);$
$taskstate = finished \wedge msg2.mbody := finishedfi\ ;\ taskstate = faulty \wedge msg2.mbody := faulty;$
$taskstate = running \wedge skip\ ;\ msg2.id := gen\text{-}msg\text{-}id() \wedge msg2.sender := id \wedge msg2.receiver := 0;$
$skip \wedge connect[id] := (\surd connect[id] + 1, msg2)$

$worker() \triangleq \{id, buffer, taskid, taskstate, msg1, msg2, buffer\} : WKsetup(id, buffer);$
$dealTask(id, msg1, connect, taskid)\ ;\ sendMessage(taskid, taskstate, id, msg2)$

$main() \triangleq createprocess(root())\ ;\ (createprocess(worker()))^{w}$

Finally, we block the details of procedure *WKsetup*, *dealTask* and *sendMessage* from the high level specification, treating them as domain functions. The final high level specification is as follows:

$$root() \triangleq \{numw, msg1, msg2, connect, id, buffer\} : RTsetup(id, numw);$$
$$assignTasks(numw, msg1, msg2, connect, id, buffer)\ ;\ tideUp(msg1)$$

$$worker() \triangleq \{id, buffer, taskid, taskstate, msg1, msg2, buffer\} : WKsetup(id, buffer);$$
$$dealTask(id, msg1, connect, taskid)\ ;\ sendMessage(taskid, taskstate, id, msg2)$$

$$main() \triangleq createprocess(root())\ ;\ (createprocess(worker()))^{W}$$

8.4.3 Summary

The extracted specification gives software engineers a clear impression of the task farm system. It describes the system in an hierarchical order, and only keeps the basic descriptions. From *main*, it is easy to know that the system consists of one *root* process and *M worker* processes. The *root* process first sets up itself, then assigns tasks to every worker who send a 'finishes' signal to the *root*. After the assignment, *root* resets the message body. The key contents of each step are given in the corresponding procedures. For each *worker* process, it first sets up itself, then processes the assigned task and sends corresponding messages to the *root*.

The *root* process and all *worker* process run in parallel. The communication between them are implemented through 'shunts', namely, *connect*[100].

This case study demonstrates how the proposed approach deals with concurrent/parallel programs with communication.

8.5 Mine Drainage System

8.5.1 Background

This case study is based on one which commonly appears in the literature. It concerns the software necessary to manage a simplified pump control system for a mining environment [34]. It is a good demonstration of the real-time aspect of the proposed approach.

The system is used to pump mine water, which collects in a sump at the bottom of

the shaft, to the surface. The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion. A simple schematic diagram of the system is given in Figure 8.2.

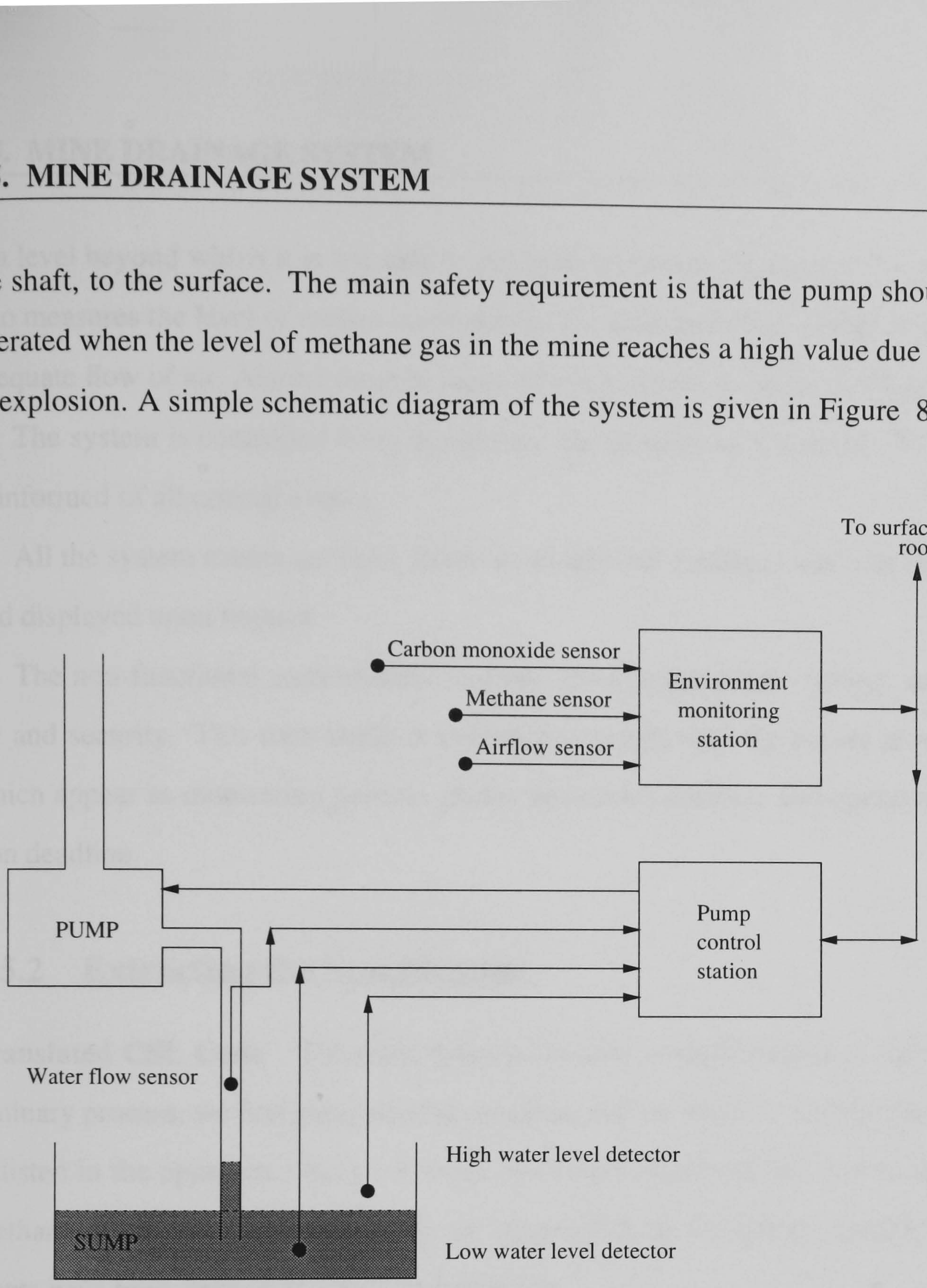

Figure 8.2: A Mine Drainage Control System

The functional specification of the system is divided into four components: the pump operation, the environment monitoring, the operator interaction, and system monitoring.

The required behaviour of the pump is that it monitors the water levels in the sump. When the water reaches a high level, the pump is turned on and the sump is drained until the water reaches the low level. At this point, the pump is turned off. A flow of water in the pipe can be detected if required. The pump should be allowed to operate only if methane level in the mine is below a critical level.

The environment must be monitored to detect the level of methane in the air; there

is a level beyond which it is not safe to cut coal or operate the pump. The monitoring also measures the level of carbon monoxide in the mine and detects whether there is an adequate flow of air. Alarms must be signaled if gas levels or air-flow become critical.

The system is controlled from the surface via an operator's console. The operator is informed of all critical events.

All the system events are to be stored in an archival database, and may be retrieved and displayed upon request.

The non-functional requirements includes three components: timing, dependability and security. This case study is mainly concerned with the timing requirements, which appear as monitoring periods, pump shut-down deadline and operator information deadline.

8.5.2 Extracting the Specification

Translated CSL Code The mine drainage system is implemented in ADA. As preliminary process, we first translated this implementation into CSL. The complete result is listed in the appendix. Here we focus on two selected modules: pump control and methane detection. For conciseness, we assume that all the global variables and constants have been defined in the main procedure.

Pump Module The CSL code is as follows:

```
proc motor-unsafe()
  {
     if motor-status=On
     then
        sw:=Off;
        motor-status:=Off;
        motor-log(In "motor-stopped")
     fi;
     motor-condition:=Disabled;
     motor-log(In "motor-unsafe")
  };

proc motor-safe()
  {
```

```
    if motor-status=Off
    then
        sw:=On;
        motor-status:=On;
        motor-log(In "motor-started")
    fi;
    motor-condition:=Enabled;
    motor-log(In "motor-safe")
};

proc set-pump(In pump-status: Boolean;)
  {
    if pump-status=On
    then if motor-status=Off
          then if motor-condition=Disabled
                 then err-msg(In "pump-not-safe")
                 fi;
                 if ch4-status=Motor-safe
                 then motor-status:=On;
                   sw:=On;
                   motor-log(In "motor-started")
                 else err-msg(In "pump-not-safe")
                 fi
          fi
    else if motor-status=On
          then motor-status:=Off;
                 if motor-condition=Enabled
                 then
                   sw:=Off;
                   motor-log(In "motor-stopped")
                 fi
          fi
    fi
};
```

As the first step, we abstract the three procedures separately into ITL specification:

$$motor\text{-}unsafe() \triangleq motor\text{-}status = On \wedge (sw := Off\ ;\ motor\text{-}status := Off\ ;\ motor\text{-}log('motor\text{-}stopped'));$$
$$motor\text{-}condition := Disabled\ ;\ motor\text{-}log('motor\text{-}unsafe')$$

$$motor\text{-}safe() \triangleq motor\text{-}status = Off \wedge (sw := On\ ;\ motor\text{-}status := On\ ;\ motor\text{-}log('motor\text{-}started'));$$
$$motor\text{-}condition := Enabled\ ;\ motor\text{-}log('motor\text{-}safe')$$

$$set\text{-}pump(pump\text{-}status) \triangleq$$
$$(pump\text{-}status = On \wedge$$
$$(motor\text{-}status = Off \wedge$$
$$(motor\text{-}condition = Disabled \wedge err\text{-}msg('pump\text{-}not\text{-}safe'));$$
$$(ch4\text{-}status = Motor\text{-}safe \wedge (motor\text{-}status := On\ ;\ sw := On\ ;\ motor\text{-}log('motor\text{-}started')))$$
$$\vee(ch4\text{-}status = Motor\text{-}unsafe \wedge err\text{-}msg('pump\text{-}not\text{-}safe'))))$$

$$\vee(\textit{pump-status} = \textit{Off} \wedge$$
$$\textit{motor-status} = \textit{On} \wedge (\textit{motor-status} := \textit{Off};$$
$$\textit{motor-condition} = \textit{Enabled} \wedge (\textit{sw} := \textit{Off}\ ;\ \textit{motor-log}('\textit{motor-stopped}'))))$$

In the above specification, there are several things that need to be simplified. Firstly, some "chop" operators could be replaced by logic conjunction, and therefore resulting in further logic composition. Secondly, there are quite a lot exception test and handling details in the specification. In high level specification, this kind of descriptions could be considered as implementation details and therefore be abstracted away. A more abstracted specification is given as follows:

$$\textit{motor-unsafe}() \triangleq \textit{motor-status} = \textit{On} \wedge (\textit{sw} := \textit{Off}\ ;\ \textit{motor-status} := \textit{Off} \wedge \textit{motor-log}('\textit{motor-stopped}'));$$
$$\textit{motor-condition} := \textit{Disabled} \wedge \textit{motor-log}('\textit{motor-unsafe}')$$

$$\textit{motor-safe}() \triangleq \textit{motor-status} = \textit{Off} \wedge (\textit{sw} := \textit{On}\ ;\ \textit{motor-status} := \textit{On} \wedge \textit{motor-log}('\textit{motor-started}'));$$
$$\textit{motor-condition} := \textit{Enabled} \wedge \textit{motor-log}('\textit{motor-safe}')$$

$$\textit{set-pump}(\textit{pump-status}) \triangleq$$
$$(\textit{pump-status} = \textit{On} \wedge$$
$$(\textit{motor-status} = \textit{Off} \wedge$$
$$(\textit{ch4-status} = \textit{Motor-safe} \wedge (\textit{motor-status} := \textit{On}\ ;\ \textit{sw} := \textit{On}\ ;\ \textit{motor-log}('\textit{motor-started}')))))$$
$$\vee(\textit{pump-status} = \textit{Off} \wedge$$
$$\textit{motor-status} = \textit{On} \wedge (\textit{motor-status} := \textit{Off};$$
$$\textit{motor-condition} = \textit{Enabled} \wedge (\textit{sw} := \textit{Off} \wedge \textit{motor-log}('\textit{motor-stopped}'))))$$

More concisely, the specification is as follows:

$$\textit{motor-unsafe}() \triangleq \textit{motor-status} = \textit{On} \wedge (\textit{sw} := \textit{Off}\ ;\ \textit{motor-status} := \textit{Off} \wedge \textit{motor-log}('\textit{motor-stopped}'));$$
$$\textit{motor-condition} := \textit{Disabled} \wedge \textit{motor-log}('\textit{motor-unsafe}')$$

$$\textit{motor-safe}() \triangleq \textit{motor-status} = \textit{Off} \wedge (\textit{sw} := \textit{On}\ ;\ \textit{motor-status} := \textit{On} \wedge \textit{motor-log}('\textit{motor-started}'));$$
$$\textit{motor-condition} := \textit{Enabled} \wedge \textit{motor-log}('\textit{motor-safe}')$$

$$\textit{set-pump}(\textit{pump-status}) \triangleq$$
$$(\textit{pump-status} = \textit{On} \wedge \textit{motor-status} = \textit{Off} \wedge \textit{ch4-status} = \textit{Motor-safe} \wedge$$
$$(\textit{motor-status} := \textit{On}\ ;\ \textit{sw} := \textit{On} \wedge \textit{motor-log}('\textit{motor-started}')))$$
$$\vee(\textit{pump-status} = \textit{Off} \wedge \textit{motor-status} = \textit{On} \wedge$$
$$(\textit{motor-status} := \textit{Off}\ ;\ \textit{motor-condition} = \textit{Enabled} \wedge (\textit{sw} := \textit{Off} \wedge \textit{motor-log}('\textit{motor-stopped}'))))$$

The log function is not directly related with system performance, therefore could be abstracted away:

$$\textit{motor-unsafe}() \triangleq \textit{motor-status} = \textit{On} \wedge (\textit{sw} := \textit{Off}\ ;\ \textit{motor-status} := \textit{Off})\ ;\ \textit{motor-condition} := \textit{Disabled}$$

$$\textit{motor-safe}() \triangleq \textit{motor-status} = \textit{Off} \wedge (\textit{sw} := \textit{On}\ ;\ \textit{motor-status} := \textit{On})\ ;\ \textit{motor-condition} := \textit{Enabled}$$

$$\begin{aligned}
&\textit{set-pump}(\textit{pump-status}) \triangleq \\
&\quad (\textit{pump-status} = \textit{On} \wedge \textit{motor-status} = \textit{Off} \wedge \textit{ch4-status} = \textit{Motor-safe} \wedge \\
&\qquad (\textit{motor-status} := \textit{On}\ ;\ \textit{sw} := \textit{On})) \\
&\quad \vee (\textit{pump-status} = \textit{Off} \wedge \textit{motor-status} = \textit{On} \wedge \\
&\qquad (\textit{motor-status} := \textit{Off}\ ;\ \textit{motor-condition} = \textit{Enabled} \wedge \textit{sw} := \textit{Off}))
\end{aligned}$$

Methane Model The CSL code is as follows:

```
proc init()
  {
    comment:"enable device";
    ch4-sensor-status:=Enabled;
    ch4-status:=Motor-unsafe
  };

proc ch4-process()
  {
    read tm, ch4-level from ch4-sensor;
    if ch4-level≥ch4-Max
    then if ch4-status=motor-safe
         then motor-unsafe();
              operator-console-alarm(In "High-methane");
              ch4-status:=motor-unsafe
         fi
    else if (ch4-level<ch4-Max−jitterrange)
         then motor-safe();
              ch4-status:=motor-safe
         fi
    fi;
    ch4-log(In ch4-level)
  };

proc ch4-period()
  {
    init();
    while true do
         duration in 30 ch4-process() end;
         delay (80−30)
    od
  };
```

As the first step, we abstract the three procedures separately into ITL specification:

$$init() \triangleq ch4\text{-}sensor\text{-}status := Enabled\ ;\ ch4\text{-}status := Motor\text{-}unsafe$$

$$\begin{aligned}
&ch4\text{-}process() \triangleq tm = \surd ch4\text{-}sensor \wedge ch4\text{-}level = read(ch4\text{-}sensor);\\
&\quad (ch4\text{-}level \geq ch4\text{-}Max) \wedge\\
&\qquad ch4\text{-}status = motor\text{-}safe \wedge\\
&\qquad\quad (motor\text{-}unsafe()\ ;\ operator\text{-}console\text{-}alarm('High\text{-}methane')\ ;\ ch4\text{-}status := motor\text{-}unsafe)\\
&\quad \vee (ch4\text{-}level < ch4\text{-}Max) \wedge (ch4\text{-}level < ch4\text{-}Max - jitterrange) \wedge (motor\text{-}safe()\ ;\ ch4\text{-}status := motor\text{-}safe);\\
&\quad ch4\text{-}log(ch4\text{-}level)
\end{aligned}$$

$$ch4\text{-}period() \triangleq init()\ ;\ (ch4\text{-}process() \wedge len \leq 30ms\ ;\ len = 50ms)^*$$

Similarly, we replace possible "chop" operators with logic conjunction, leave out the unused timestamp *tm*. The final result will look like as below:

$$init() \triangleq ch4\text{-}sensor\text{-}status := Enabled \wedge ch4\text{-}status := Motor\text{-}unsafe$$

$$\begin{aligned}
&ch4\text{-}process() \triangleq ch4\text{-}level = read(ch4\text{-}sensor);\\
&\quad (ch4\text{-}level \geq ch4\text{-}Max) \wedge ch4\text{-}status = motor\text{-}safe \wedge\\
&\qquad (motor\text{-}unsafe() \wedge operator\text{-}console\text{-}alarm('High\text{-}methane') \wedge ch4\text{-}status := motor\text{-}unsafe)\\
&\quad \vee (ch4\text{-}level < ch4\text{-}Max) \wedge (ch4\text{-}level < ch4\text{-}Max - jitterrange) \wedge (motor\text{-}safe() \wedge ch4\text{-}status := motor\text{-}safe);\\
&\quad ch4\text{-}log(ch4\text{-}level)
\end{aligned}$$

$$ch4\text{-}period() \triangleq init()\ ;\ (ch4\text{-}process() \wedge len \leq 30ms\ ;\ len = 30ms)^*$$

8.5.3 Summary

The purpose of the mine drainage case study is to demonstrate that the proposed approach has the ability to tackle systems with critical time requirement. This is achieved through the following points:

1. RWSL has the power to represent time critical systems from specification level to source code level.
2. The abstraction rules are specially designed to deal with time feature.
3. ITL is powerful for real time system specification.

Chapter 9

Conclusion

9.1 Criteria for Success and Analysis

9.1.1 The approach

In Chapter 1, a set of criteria are proposed to judge the success of the approach described in this thesis. In this section, detailed analysis of our approach are presented based on these criteria.

- *For a heavily modified legacy system which has never been developed in a well structured or object-oriented method, how viable is it to extract a specification from its source code with the proposed abstraction technology?*

  The approach can extract specifications of various legacy systems from their source code no matter whether they had been modified or were well structured. The abstraction definitions and rules are language independent and are based on most popular structures of legacy systems, i.e., both system structure and statement structure. The abstraction rules cover all the basic statements in legacy statement. Other statements, if uncovered, can be treated as variations of the basic statements, and relevant abstraction rules can be easily derived. However, the more structured a legacy system is, the easier the extraction process may be. As

discussed in section 4.2.1, to make specification extraction more efficient, various existing re-structuring techniques can be used to decompose and structure a legacy system if it is very monolithic or unstructured [20, 78, 100].

- *Is the extracted specification consistent to the original design? Is it reliable to perform redesign or respecification on the base of the extracted specification?*

  The answer is positive. This is guaranteed by the soundness of abstraction rules, which is proven in ITL. Since every movement in specification extraction is based on certain abstraction rules, the extracted specification must be consistent to the original design.

- *Is the extracted specification unambiguous and easy to understand?*

  The answer is also positive. Using ITL guarantees the unambiguity of the extracted specification. Actually, this is one of the main reasons that we adopt ITL. The extracted specification is also easy to understand, because ITL is well structured and has a first order predicate logic nature, which is quite popular in formal computing.

- *What kind of legacy systems can the approach deal with? Besides sequential non-time systems, can it tackle more complex and emergent-in-need but rarely addressed systems, such parallel and time critical systems?*

  Besides sequential non-time systems, which are addressed by most reverse engineering research, the approach takes time-critical systems with parallelism as its specific application domain. RWSL is designed to have real-time and parallel feature: from ITL at specification level to CSL/COOL at code level, relevant elements such as delay, duration, signal, parallel and communication, are developed. Correspondingly, abstraction rules cover how to deal with these elements.

- *Crossing levels of abstraction involves both semantics change/selection and transformation in representation. How does the proposed approach solve this prob-*

lem? Is the taxonomy of abstraction comprehensive enough and are the abstraction rules reliable?

Specification extraction involves crossing levels of abstraction. Abstraction is the crucial technique to reverse engineering. Without tackling abstractions properly, any design or specification recovery methodology can not succeed. To achieve correct and practical abstraction, our approach answered two fundamental questions. By formally defining a taxonomy of abstraction, we answered the first question "what abstraction is". Monotonicity and relations between these abstractions are discussed and healthiness obligations are developed as axioms to guarantee correct and sensible abstraction during reverse engineering. Once abstractions are identified, the next question is "how to perform these abstractions". Relevant abstraction rules are developed to solve this problem.

The taxonomy of abstraction covers abstractions in both normal sequential non-time and real-time parallel systems, and therefore are comprehensive enough to deal with abstraction problems in these domains. The developed abstraction rules are formally defined and proved sound, and therefore are considered reliable. The case studies conducted also show positive evidence to this conclusion.

- *Is the approach feasible for realisation? For example, is it possible to build a practical tool to demonstrate the approach?*

  Quite a lot attention was paid to the practical part of the approach during development. RWSL, abstraction taxonomy and rules are not only theoretically correct, but also workable for real legacy systems. The examples and case studies show that the approach is a "practical" one, i.e., feasible for practice. A resulting tool named Reengineering Assistant has been built.

- *Is the approach capable for industrial-scaled systems?*

  The approach is capable for industrial-scaled systems and efficient enough for real practice. The approach adopts systematic stepwise abstraction, which slices

a large system into manageable sub-systems, then deals with these sub-systems separately and finally integrates the results into one full view of the system. The approach also supports automation, and a semi-automatic tool has been built, which much improves the efficiency of reengineering process, together with the problem size.

- *Is there any special prerequisite for using the approach?*

  To use the approach, the software engineer has to have a sound understanding and skills of formal methods, i.e., ITL. Otherwise, the resulting specification may be somewhat difficult to understand. Without sound skills in ITL, the resulting specification may also be not concise enough.

9.1.2 The Tool

In this section, we assess the developed tool with a set of criteria. More comprehensive data may be collected from diverse users.

- *Ease of Use.*

  One measure of a tool's effectiveness is the ease with which the user can operate it. No matter how functional or complete a tool is, if the user spends most time thinking about how to use the tool or making the tool work, then the tool is hindering and not helping to complete the task. To justify using a tool, the tool's benefits must offset its cost with the time spent using it.

  RA ranks a high score of this criterion due to the following features:

  – *Intelligence*

    RA helps the user by performing its functions intelligently. This intelligence embodies in the strong automatic inference mechanism to perform its functions, which enables RA accomplish user-selected functions without user intervention. All the elementary abstractions can be done automatically, and all further abstractions can be done automatically provided

that correct user observations of current situation have been identified. In addition, RA could anticipate user decision and interaction by providing possible operations' prompt and information prompt.

– *Predictability*

Unpredicted responses from a tool usually result in unhappy users and unwanted output. RA was designed to avoid this shortcoming. Menu and command names in RA suggest the function well and users are provided with good explanations of the execution results. If an unpredicted result/response does occur, the user could use the "undo" menu to track back to any previous point he wants. No matter how drastic the result of a particular command be, this backtracking is always possible.

– *Error Handling*

RA considers possible error cases comprehensively. RA is tolerant to many user errors, it checks for the errors, corrects the errors whenever possible, and gives relevant prompt information.

– *System Interface*

RA provides a friendly user interface. RWSL representation of target systems are displayed nicely with the pretty print module. Operation feedback is displayed in the LISP operation window. And the menus and buttons are well formatted.

However, currently RA is only available under UNIX environment. To increase its popularity, a PC version will be developed.

- *Tool Leverage*

Leverage is the extent to which small actions by the user create large effects. The leverage of any interactive tool is a function of its command/menu set. The method RA uses to increase this leverage is to increase its intelligence and there-

fore to integrate closely related functions into one menu as many as possible. RA has a quite high tool leverage due to its high intelligence.

- *Performance*

  The performance of a tool can greatly affect the ease with which it is used and can ultimately determine the success of a tool within an organisation. A tool must be able to function efficiently and be responsive to the user. Poor tool performance can create costs that negate many of the benefits realised from tool use. A tool that performs inefficiently may result in missed schedules or frustrated users who are sceptical about whether the tool really helps them.

  RA responses to all the possible user choices correctly and within tolerable time. The examples and case studies show that RA's performance is satisfiable.

- *Robustness*

  The robustness of a tool is a combination of such factors as: the reliability of the tool, the performance of the tool under failure conditions, the criticality of the consequences of tool failures, the consistency of the tool operations, and the way in which a tool is integrated into the environment.

  RA is robust, because it has the following features:

  - *Consistency*

    The operations of RA are consistent with each other, they all contribute to the sole goal, that is, specification extraction.

  - *Evolution*

    In all but the most unusual cases, due to the component-based nature and good system interface, RA could evolve over time to accommodate changing requirements, changes to the environment, correcting detected flaws, and performance enhancements.

  - *Fault Tolerance*

RA considers possible error cases comprehensively. RA is tolerant to many user errors, it checks for the errors, corrects the errors whenever possible, and gives relevant prompt information.

Since RA has an operation history, the user can always backtrack to any previous point once an unrecoverable error happened.

- *Functionality*

  The functionality of a tool is not only driven by the task that the tool is designed to perform but also by the methods used to accomplish the task. Many tools support methodologies. The accuracy and efficiency with which the tool does this can directly affect the understandability and performance of the tool, as well as determine the quality and usefulness of tool outputs. In addition, a tool that generates incorrect outputs can lead to frustrated users and extra time expenditures needed to "fix" tool outputs. These additional costs may weigh heavily against a tool's benefits.

RA answered these questions successfully by providing the following features:

  - *Methodology Support*

    A methodology is a systematic approach to solving a problem. The proposed approach is a methodology because it prescribes a set of steps and work products as well as rules to guide the production and analysis of reverse engineering process. Automated support for a methodology can aid its use and effectiveness. The design and functionality of RA are based exactly on the proposed approach, and RA provides coherent support to the proposed approach.

  - *Correctness*

    To be useful, RA operates correctly and produces correct outputs.

9.2 Conclusion

9.2.1 Lessons Learnt

Through developing the approach, we learnt the following lessons:

Definition of abstraction levels - RWSL provides a spectrum of abstractions of the reengineered system, from concrete code to formal specification. These abstractions are integrated and cooperated in a uniform manner. All the layers in RWSL have formal syntax and semantics, which give the target system unambiguous descriptions at various abstraction levels.

Development of formal abstraction rules - In our approach, reverse engineering is carried out by extracting system descriptions at a higher abstraction level from those at lower abstraction levels. Based on RWSL, a set of the abstraction rules were developed. All the abstraction rules are defined formally with ITL. This assures precise and rigorous semantics of the rules, and provides us with confidence in the obtained specification.

Application in real-time domain - At present, existing reverse engineering technology is limited to merely sequential and non-time systems no matter it adopts formal techniques or ad hoc techniques [110]. Our approach is based on a wide spectrum language, which is designed to bear an ability to describe time critical features of the target system in a wide span.

Object-Orientation - The proposed approach relates to object-orientation in two aspects. Firstly, the approach aims to transform procedural legacy systems into object-oriented systems at the code level. A set of *object extraction rules* [108] are developed. For a well-structured procedural program, i.e., there is no unnecessary coupling and relevance between data and procedures, these rules can transform the procedural program

in TGCL into a reasonable and satisfactory object-oriented program in ObTAM. Secondly, our approach supports reverse engineering of object-oriented systems, i.e., using abstraction rules, an object-oriented program (in ObTAM form) can be abstracted into a logic specification.

Abstractness Measurement - We believe that corresponding metric measures should be developed in conjunction with the development of any reverse engineering approach and therefore a metric facility is developed by S. Zhou for the abstraction approach [171, 169]. Metrics on abstractness of software are useful to a software reverse engineer who is trying to derive software designs or specifications from existing code, because abstractness measures can help to guide the engineer to reverse engineer code more effectively in selecting abstraction rules (to help develop heuristics on what the final abstraction form the measured program should be in), to measure the progress made in optimising the program code and to measure the resulting quality of the program being abstracted. The following metrics were defined based on the data and control structures of RWSL programs in the prototype of our reengineering tool [116, 169, 172]: Abstractness based on McCabe's Cyclomatic Complexity Measure (ABST-MCCM), Abstractness based on Lines Of Code (ABST-LOC), Abstractness based on Control-Flow and Data-Flow Complexity (ABST-CFDF), Abstractness based on Loop Complexity (ABST-LOOP), Abstractness in Vocabulary (ABST-VOC), and Abstractness in Statement (ABST-STAT).

9.2.2 Our Approach and Existing Work

Existing research closely related to our work (formal and informal) on software abstraction for reverse engineering have been studied when our approach was developed. Here only the most related projects are briefly discussed.

- Transformation-based Maintenance Model (TMM) is a method proposed in [9] for recovering abstractions and design decisions that were made during imple-

mentation. The abstractions and design decisions of software must be recovered first before the software is re-implemented.

- [62] proposed "A Concept Recognition-Based Program Transformation System", whose characteristic is its use of concept recognition, the understanding and abstraction of high-level programming and domain entities in programs, as the basis for transformations. Four understanding levels are defined: the text level, the syntactic level, the semantic level, and the concept level. The program transformation system depends on its program understanding capabilities up to the concept level. The key component is a concept library which contains the knowledge about programming and application domain concepts, and concept recognition is done by pattern matching.

- REFORM project developed a tool named the Maintainer's Assistant to assist the human maintainer, handling assembler and Z in an easy to use way [22, 165, 167]. One of the most important successes of Maintainer's Assistant is that it is based on a wide spectrum language whose syntax and semantics are formally defined. Maintainer's Assistant focused on transformation rather than abstraction. It involved little in how to use multi-levelled abstractions and relevant abstraction rules to reach a good system reengineering, especially reverse engineering. The Wide Spectrum Language in Maintainer's Assistant is sequential and non-timed, which limits its application domains.

- Research in University of California at San Diego [91] based their approach to reverse engineering on abstraction, and identified three kinds of abstractions: problem domain, structural, and logical. However, this work is not formalised and did not have multiple abstraction levels with an integrated formal semantics. These limit the accuracy and power of their approach.

- PRISME is a reverse engineering tool based on functional abstraction [16], but the abstraction in PRISME is function-based instead of semantics-based. It does

not engage a mature formal method to specify the target system, and PRISME extracts 'signatures' as pieces of outline description of the system, but not complete specification. Moreover, the notations in PRISME lack of integrated semantic foundation.

- AUTOSPEC project [42, 44, 73] involves a two-phase approach to reverse engineering that integrate a process for abstracting formal specifications from program code with a technique for identifying candidate objects in program code. A prototype tool was built. This project only deals with the first abstraction step of reverse engineering, i.e., it extracts low level specifications, in the form of predicate logic as a notation of the source code. Therefore, *AUTOSPEC* only considers the initial step in the whole process of reversing source code into a system specification.

To summarise, although many aspects of reverse engineering have been researched, using formal abstraction rules to extract formal specifications from code is rarely addressed. The above listed studies solved some closely-related problems, such as transformation and part of informal abstraction. However, none of them engages in extracting semantics-consistent formal specifications from source code through abstraction. Formal abstraction rules for reverse engineering seemed not to be developed. Most of these approaches have been advocated for reverse engineering, but few have been evaluated in practice on large-scale code. Abstraction levels are not clearly defined. Abstraction to be used in a "real-time" system has been rarely addressed. Techniques for coping with crossing levels of abstraction covering all abstraction levels need more research. Where genuine crossing of levels of abstraction occurs, this is done manually.

9.2.3 Conclusion

The features of our abstraction approach (including the tool developed in the project - the Reengineering Assistant) are as follows:

- use of ITL to define RWSL, allowing both non-real-time and real-time programs to be represented and manipulated;

- a small, traceable kernel language, i.e., ITL plus TGCL and ObTAM, allowing very precise and thorough formal semantics to be given to RWSL;

- transformation for all kinds of programs;

- object-extraction rules to enable transferring legacy procedural programs to object-oriented programs;

- abstraction rules for crossing levels of abstraction in a stepwise manner;

- abstraction patterns as a means of describing current abstraction situations and acquiring expert observations of the target system, and then applying these observations in further abstraction;

- dealing with various languages via simple translation followed by automatic restructuring and simplification;

- an interactive, semi-automatic tool, rather than attempting complete automation, thereby making good use of human expert knowledge about the software and its domain;

- mechanical checking of the correctness conditions on transformation, object extraction and abstraction, appearing in the tool menus;

- using the prototype and manual case studies to demonstrate how the experienced user solves a problem, and then implementing these methods and heuristics;

- rapid prototyping development, with the system organised as a collection of abstract machines with formally defined interfaces.

To conclude this thesis: a reengineering approach with an emphasis on reverse engineering using program abstraction is proposed. A supporting tool based on the ap-

proach is developed to speed and to scale up practical reengineering. A formal framework based on ITL semantics was developed and it is implemented in a wide spectrum language, RWSL. We have formalised program abstraction within a reengineering environment. The abstraction problem has been addressed by software engineering researchers for some years but dedicated approaches used in a reengineering environment with both concurrency and real-time features have been non-existing.

The specification produced is then understood and used as basis of enhanced specification for forward engineering the system. Before proceeding, the specification may be changed and/or extended with extra non-functional requirement(s) that we require (e.g. reliability, dependability, limited resources, etc.).

Through the discussion in this thesis, it can be concluded that program abstraction is a powerful means for reverse engineering and a systematic approach of reengineering such as the one proposed in this thesis will help reengineering.

9.3 Future Directions

Based on the discussions in former sections, we concluded that the approach has novel ideas and is successful in reverse engineering. The resulting tool scales up the approach and is consistent with the approach. In this section, we explore some possible extensions of the present work.

An ITL specification is rigorous and structured. It provides software engineers with a good basis for respecification, redesign and further forward engineering. A suitable graphic model could be developed and integrated with the formal ITL model to give the target system more intuitive description. This graphic model should also be structured, and may be hierarchic. It should focus on the overview structure of the target system, and not include many system details. A mechanism should be developed to keep the consistency between the ITL model and the graphic model. Any changes in ITL model should be reflected in the graphic model automatically.

Software reuse is an important technique in software development. Component-

based software reengineering covers the study of extracting reusable components from legacy source code and reusing them in further design and forward engineering. As stated before, the Reengineering Assistant plans to include the reuse part in a broad sense. It could be a good and useful research issue how to apply the abstraction-based reverse engineering technique discussed in this thesis to the extraction of reusable components, especially their specifications and documents. The connection or integration between the current reverse engineering part and the reuse part should be addressed properly in future study, including both the theoretical approach and practical tool.

The approach and tool aim at dealing with sequential non-timed systems and real-time systems with parallelism. Although domain features and domain knowledge are considered carefully during the development, more profound study of specific domain knowledge could help improve the automation of the tool further. This is because real-time systems are diversified and complicated, and different sub branches have distinct characteristics.

References

1. ABD-EL-HAFIZ, S. K., AND BASILI, V. R. A knowledge-based approach to the analysis of loops. *IEEE Transactions on Software Engineering* 22, 5 (May 1996).

2. ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company, 1985.

3. ABRIAL, J. R., SCHUMAN, S. A., AND MEYER, B. *Specification Language Z.* Massachusetts Computer Associates Inc., Boston, 1979.

4. ALUR, R., AND DILL, D. Automata for modeling real-time systems. *In M.S. Paterson editor, ICALP 90: Automata, Languages and Programming, Lecture Notes in Computer Science* (1990), 322–335.

5. ANGER, F. D., RODRIGUEZ, R. V., AND YOUND, M. Combining static and dynamic analysis of concurrent programs. In *Proceedings of the International Conference on Software Maintenance* (Sept. 1995), IEEE Computer Society Press, pp. 98–99.

6. ANSI. *Standard 729.* IEEE Standard Glossary of Software Engineering Terminology, 1983.

7. ANTONINI, P., BENEDUSI, P., CANTONE, G., AND CIMITILE, A. Maintenance and reverse engineering: Low-level design documents production and improvement. In *IEEE Conference on Software Maintenance-1987* (Austin, Texas, 1987), pp. 13–24.

8. ANTONIOL, G., FIUTEM, R., MERLO, E., AND TONELLA, P. Application and user interface migration from basic to visual c++. In *International Conference on Software Maintenance* (Nice, Oct. 1995), pp. 76–85.

9. ARANGO, G., BAXTER, I., FREEMAN, P., AND PIDGEON, C. TMM: Software maintenance by transformation. *IEEE Software* (May 1986), 27–39.

10. ARNOLD, R. *Software Re-engineering*. IEEE Computer Society Press, ISBN 0-8186-3271-2, 1992.

11. ARNOLD, R. S., AND BOHNER, S. A. Impact analysis–towards a framework for comparison. In *Proceedings of the International Conference on Software Maintenance* (Sept. 1993), IEEE Computer Society Press, pp. 292–301.

12. ARTHUR, L. J. *Software Evolution: The Software Maintenance Chanllenge*. John Wiley & Sons, 1988.

13. AWAD, M., KUUSELA, J., AND ZIEGLER, J. Requirements specification and software architecture. *Embedded Systems Programming Magazine* (Oct. 1996).

14. BACHMAN, R. *A CASE for Reverse Engineering*. Cahners Publishing Company, July 1988. reprinted from DATAMATION.

15. BAETEN, J. C. M., AND BERGSTRA, J. A. Real time process algebra. *Formal Aspects of Computing 3* (Feb. 1991), 142–188.

16. BALMAS, F. Prisme: Formalizing programming strategies as a way to understand programs. In *Eighth International Conference on Software Engineering and Knowledge Engineering* (Lake Tahoe, Nevada, June 1996), IEEE Computer Society.

17. BALMAS, F. Toward a framework for conceptual and formal outlines of programs. In *Fourth Working Conference on Reverse Engineering* (Amsterdam, The Netherlands, October 1997), IEEE Computer Society, pp. 226 – 235.

18. BEHFOROOZ, A., AND HUDSON, F. J. *Software Engineering Fundamentals*. Oxford University Press, 1996.

19. BENNETT, K. H. The software maintenance of large software systems: Management method and tools. Technical report, Durham University, 1989.

20. BENNETT, K. H. Software maintenance for the year 2000. In *The Sixth European Workshop on Software Maintenance* (The Centre for Software Maintenance, Durham University, England, 1992).

21. BENNETT, K. H. An overview of maintenance and reverse engineering. In *The REDO Compendium*. John Wiley Sons, Inc., Chichester, 1993.

22. BENNETT, K. H., BULL, T., AND YANG, H. A transformation system for maintenance — turning theory into practice. In *IEEE Conference on Software Maintenance-1992* (Orlando, Florida, Nov. 1992).

23. BENNETT, K. H., CORNELIUS, B. J., MUNRO, M., AND ROBSON, D. J. Software maintenance. In *Software Engineer's Reference Book*. Butterworth Heinemann, 1991. pp. 20/1-20/18.

24. BENNETT, K. H., DENIER, J., AND ESTUBLIER, J. Environments for software maintenance. Technical report, Durham University, 1989.

25. BENVENISTE, A., AND HARTER, P. K. Proving real-time properties of programs with temporal logics. In *Proceedings of ACM SIGOPS 8th annual ACM symposium on Operating systems Principles* (Dec. 1981), pp. 1–11.

26. BERGSTRA, J. A., AND KLOP, J. W. Process algebra for synchronous communication. *Information and Control 60* (Jan. 1984), 109–137.

27. BERTHOMIEU, B., AND DIAZ, M. Modeling and verification of time dependent systems using timed petri nets. *IEEE Transactions on Software Engineering 17* (Mar. 1991), 259–273.

28. BERZINS, V. *Software Merging and slicing*. IEEE Computer Society Press, ISBN 0-8186-6792-3, 1995.

29. BILLINGTON, J., WHEELER, G. R., AND WILBUR-HAM, M. C. PROTEAN: a high-level petri net tool for the specification and verification of communication protocol. *IEEE Transactions on Software Engineering 14* (Mar. 1988), 301–316.

30. BJØRNER, D., AND JONES, C. B. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

31. BROWN, A. Specifications and reverse engineering. *Software Maintenance: Research and Practice 5*, 3 (1993).

32. BULL, T. An introduction to the WSL program transformer. In *IEEE Conference on Software Maintenance-1990* (San Diego, California, 1990).

33. BULL, T. *Software Maintenance by Program Transformations in A Wide Spectrum Language*. Ph.D. thesis, Durham University, 1994.

34. BURNS, A., AND WELLINGS, A. $HRT-hood^{TM}$: A structured design method for hard real-time ada system. *Real-time Safety Critical Systems Series 3* (1995).

35. BUSS, E., AND ET AL. Investigating reverse engineering technologies for the cas program understanding project. *IBM Systems Journal 33*, 3 (1994), 477–500.

36. CANFORA, G., CIMITILE, A., AND CARLINI, U. D. A logic based approach to reverse engineering tools production. In *Proceedings of the International Conference on Software Maintenance* (1991), IEEE Computer Society Press, pp. 83–91.

37. CAU, A., CZARNECKI, C., AND ZEDAN, H. Designing a provably correct robot control system using a 'lean' formal method. In *LNCS 1486: the 5th International Symposium, Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'98* (Lyngby, Denmark, Sept. 1998).

38. CAU, A., AND ZEDAN, H. Refining interval temporal logic specifications. In *the 4th AMAST Workshop on Real-Time systems, Concurrent, and Distributed Software (ARTS'97)* (Mallorca, Spain, May 1997).

39. CAU, A., ZEDAN, H., COLEMAN, N., AND MOSZKOWSKI, B. Using ITL and tempura for large scale specification and simulation. In *Proceedings of 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE* (Braga, Portugal, 1996), pp. 493–500.

40. CHAOCHEN, Z., HOARE, C., AND RAVN, A. P. A calculus of durations. *Information Processing Letters 40* (05 1991), 269–276.

41. CHAOCHEN, Z., RAVN, A. P., AND HANSEN, M. R. An extended duration calculus for hybrid systems. *Hybrid Systems, R.L. Grossman, A. Nerode, A. P. Ravn. H. Rischel(Eds.)* (1993), 36–59.

42. CHENG, B. H. C. Applying formal methods in automated software development. *Journal of Computer and Software Engineering 2* (02 1994), 137–164.

43. CHENG, B. H. C., AND GANNOD, G. C. Abstraction of formal specifications from program code. In *Proceedings for the 3rd International Conference on Tools for Artificial Intelligence* (1991), pp. 125–128.

44. CHENG, B. H. C., AND JENG, J. J. Reusing analogous components. *IEEE Transactions on Knowledge and Data Engineering* (Nov. 1994).

45. CHIKOFSKY, E. J., AND CROSS, II, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE Software* (Jan. 1990), 13–17.

46. CHLLAS, B. F. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

47. CHU, W. C., AND YANG, H. A formal method for software maintenance. In *IEEE International Conference on Software Maintenance (ICSM'96)* (Monterey, CA, Nov. 1996).

48. CIMITILE, A., AND CARLINI, U. D. Reverse engineering: Algorithms for program graph production. *Software Practice and Experience 21* (May 1991), 519–537.

49. CLARKE, E., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite concurrent systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (1983), pp. 117–126.

50. CLAYBROOK, B. G. A specification method for specifying data and procedural abstraction. *IEEE Transactions on Software Engineering SE-8*, 5 (Sept. 1982).

51. CLAYTON, R., AND RUGABER, S. The representation problem in reverse engineering. In *Proceedings of the First Working Conference on Reverse Engineering* (Maryland, May 1993).

52. COLBROOK, A., AND SMYTHE, C. The retrospective introduction of abstraction into software. In *IEEE Conference on Software Maintenance-1989* (Miami, Florida, 1989).

53. COLBROOK, A., SMYTHE, C., AND DARLISON, A. Data abstraction in a software re-engineering reference model. In *IEEE Conference on Software Maintenance-1990* (San Diego, California, 1990).

54. CROSS, J. H. Improving comprehensibility of ada with control structure diagrams. In *Proceedings of Software Technology Conference* (Salt Lake City, Apr. 1994).

55. CROSS, J. H., CHIKOFSKY, E. J., AND MAY, JR., C. H. Reverse engineering. *Advances in Computers 35* (1992).

56. CROSS, J. H., AND HENDRIX, T. D. Using generalized markup and sgml for reverse engineering graphical representations of software. In *Proceedings of Working Conference on Reverse Engineering* (Toronto, July 1995).

57. DIETRICH, S. W., AND CALLISS, F. W. The application of deductive databases to inter-module code analysis. In *Proceedings of the International Conference on Software Maintenance* (1991), IEEE Computer Society Press, pp. 120–128.

58. DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

59. DIJKSTRA, E. W., AND SCHOLTEN, C. S. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

60. EDWARDS, H. M., AND MUNRO, M. White paper on edcs: The evolutionary design of complex software program. Technical report, DARPA EDCS Project Group, USA, 1996.

61. EIXELSBERGER, W., WARHOLM, L., KLOSCH, R., AND GALL, H. Software architecture recovery of embedded software. In *International Conference on Software Engineering (ICSE'97)* (Boston, USA, May 1997).

62. ENGBERTS, A., KOZACZYNSKI, W., AND NING, J. Concept recognition-based program transformation. In *IEEE Conference on Software Maintenance-1991* (Sorrento, Italy, 1991), pp. 73–82.

63. ETESSAMI, F., AND HURA, G. Rule-based design methodology for solving control problems. *IEEE Transactions on Software Engineering 17* (Mar. 1991), 274–282.

64. FENCOTT, C. *Formal Methods for Concurrency.* International Thomson Publishing Company, ISBN 1-85032-173-6, 1996.

65. FIUTEM, R., MERLO, E., ANTONIOL, G., AND TONELLA, P. Understanding the architecture of software systems. In *4th Workshop on Program Comprehension* (Berlin, Mar. 1996), IEEE Computer Society Press, pp. 187–196.

66. FRAKES, W. B., AND POLE, P. T. An empirical study of representation method for reusable software components. *IEEE Transactions on Software Engineering SE-20*, 8 (Aug. 1994), 617–630.

67. FRASER, M. D., KUMER, K., AND VAISHNAVI, V. K. Informal and formal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering SE-17*, 5 (May 1991).

68. GABRIELIAN, A., AND FRANKLIN, M. State-based specification of complex real-time systems. In *Proceedings of the 9th Real-Time Systems Symposium* (Dec. 1988), pp. 2–11.

69. GALL, H., KLÖSCH, R., AND MITTERMEIR, R. Object recovery from procedural systems for changing the architecture of applications. In *IEEE, ACM Third International Conference for Systems Integration (ICSI '94)* (Sao Paulo City, Brazil, Aug. 1994).

70. GALL, H., KLÖSCH, R., AND MITTERMEIR, R. Architectural transformation of legacy systems. In *ICSE-17 Workshop on Program Transformation for Software Evolution* (Seattle, USA, Apr. 1995).

71. GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* (Aug. 1991), 751–761.

72. GANNOD, C., AND CHENG, B. H. C. A two-phase approach to reverse engineering using formal methods. In *Proc. of Formal Methods in Programming and Their Applications Conference* (June 1993), Springer-Verlag.

73. GANNOD, C., AND CHENG, B. H. C. Strongest postcondition semantics as a basis for reverse engineering. In *Proc. of IEEE Working Conference on Reverse Engineering,* (Toronto, Ontario, July 1995), pp. 188–197.

74. GANNOD, G. C., AND CHENG, B. H. Facilitating the maintenance of safety-critical systems using formal methods. *The International Journal of Software Engineering and Knowledge Engineering 4* (Feb. 1994).

75. GOGUEN, J., AND TARDO, J. An introduction to OBJ: A language for writing and testing software specifications,. *In Marvin Zelkowitz editor, Specification of Reliable Software* (1979), 170–189. Reprinted by Addison Wesley in 1985, 'Specification Techniques', p391-420.

76. GOGUEN, J. A., AND TARDO, J. J. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In *Software Specification Techniques.* Addison-Wesley Publishing Company, 1986.

77. GRISWOLD, W. G., CHEN, M. I., BOWDIDGE, R. W., CABANISS, J. L., NGUYEN, V. B., AND MORGENTHALER, J. D. Tool support for planning the restructuring of data abstractions in large systems. Technical Report CS97-559, Software Engineering Laboratory, University of California at San Diego, 1997.

78. GRISWOLD, W. G., AND NOTKIN, D. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering SE-21*, 4 (Apr. 1995), 275–287.

79. GUTTAG, J., AND HORNING, J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

80. HANSSON, H. A. Time and probability in formal design of distributed systems. *Real-Time Safety Critical Systems Series 2* (1994).

81. HAUSLER, P. A., PLESZKOCH, M. G., LINGER, R. C., AND HEVNER, A. R. Using function abstraction to understand program behaviour. *IEEE Software 7*, 1 (January 1990), 55–63.

82. HOARE, C. Communicating sequential processes. *Communication of ACM 21* (08 1978), 666–677.

83. HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of ACM 12* (Oct. 1969), 576–580.

84. HOARE, C. A. R. Notes on data structuring. In *Structured Programming*. Academic Press, Inc., London, 1972.

85. HOARE, C. A. R. Proof of A structured program: The sieve of eratosthenes. *Computer 14*, 4 (1972).

86. HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

87. HOLTZBLATT, L. J., PIAZZA, R. L., REUBENSTEIN, H. B., ROBERTS, S. N., AND HARRIS, D. R. Design recovery for distributed systems. *IEEE Transactions on Software Engineering 23*, 7 (July 1997).

88. HOOMAN, J. Specification and compositional verification of real-time systems. *PhD Thesis* (1991).

89. HOOMAN, J., AND DE ROEVER, W. P. Design and verification in real-time distributed computing: an introduction to compositional methods. In *Proceedings of the 9th International Symposium on Protocol Specification, Testing and Verification* (North Holland, 1989).

90. HOOMAN, J., RAMESH, S., AND DE ROEVER, W. A compositional semantics for statecharts. *Technical Report* (1989).

91. HOWDEN, W. E., AND PAK, S. Problem domain, strutural and logical abstractions in reverse engineering. In *Proceedings of the International Conference on Software Maintenance 1992* (Nov. 1992), IEEE Computer Society Press, pp. 214–224.

92. IEEE. *IEEE Standard Collection: Software Engineering*. IEEE Inc., New York, 1997.

93. ISO. Information systems processing—open systems interconnection—LOTOS. *Technical Report* (1987).

94. JAHANIAN, F., AND MOK, A. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering 12* (09 1986).

95. JOINER, J. K., TSAI, W. T., CHEN, X. P., AND SUBRAMANIAN, S. Data-centered program understanding. In *Proceedings of the International Conference on Software Maintenance* (Sept. 1994), IEEE Computer Society Press, pp. 272–281.

96. JONES, C. B. *Software Development: A Rigorous Approach.* Prentice-Hall International, 1980.

97. JONES, C. B. *Systematic Software Development Using VDM.* Prentice-Hall International, Inc., London, 1986.

98. KARLSSON, E.-A. *Software Reuse–A Holistic Approach.* John Wiley Ltd, ISBN 0471 95489 6, 1995.

99. KINLOCH, D. A., AND MUNRO, M. Understanding C programs using the combined C graph representation. In *Proceedings of the International Conference on Software Maintenance* (Sept. 1994), IEEE Computer Society Press, pp. 172–180.

100. KOREL, B. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering SE-23*, 1 (July 1984), 352–257.

101. KWIATKOWSKI, J., PUCHALSKI, I., AND YANG, H. Pre-processing cobol programs for reverse engineering in a software maintenance tool. In *1st US Colloquium on Object Technology and System Re-engineering* (Oxford, England, Apr. 1998).

102. LANO, K. *The B Language and Method: A Guide to Practical Formal Development.* Springer-Verlag, ISBN 3-540-76033-4, 1996.

103. LANO, K. C., AND HAUGHTON, H. P. Formal development in B. *Information and Software Technology 37* (June 1995), 303–316.

104. LAYZELL, P. J., FREEMAN, M. J., AND BENEDUSI, P. Improving reverse engineering through the use of multiple knowledge sources. *Software Maintenance: Research and Practice 7* (1995), 279–299.

105. LEVESON, N. G., AND STOLZY, J. L. Safety analysis using petri nets. *IEEE Transactions on Software Engineering 13* (Mar. 1987), 386–397.

106. LIU, X. Abstraction rules for system reverse engineering. Tech. rep., STRL, Department of Computer Science, De Montfort University, England, November 1997.

107. LIU, X. A design framework for system re-engineering. Tech. rep., STRL, Department of Computer Science, De Montfort University, England, July 1997.

108. LIU, X. Object extraction rules for system reverse engineering. Tech. rep., STRL, Department of Computer Science, De Montfort University, England, July 1997.

109. LIU, X., CHEN, Z., YANG, H., ZEDAN, H., AND CHU, W. A design framework for system re-engineering. In *the Proceedings of Joint Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)* (Hong Kong, December 1997), IEEE Computer Society.

110. LIU, X., YANG, H., AND ZEDAN, H. Formal methods for the re-engineering of computing systems. In *the Proceedings of The 21st IEEE International Conference on Computer Software and Application (COMPSAC'97)* (Washington, D.C., August 1997), IEEE Computer Society, pp. 409–414.

111. LIU, X., YANG, H., AND ZEDAN, H. Improving maintenance through development experiences. In *Workshop on Empirical Studies in Software Maintenance (WESS98)* (Metropolitan, Washington D.C., USA, November 1998), IEEE Computer Society.

112. LIVADAS, P. E., AND ROY, P. K. Program dependence analysis. In *Proceedings of the International Conference on Software Maintenance* (Nov. 1992), IEEE Computer Society Press, pp. 356–365.

113. LOGRIPPO, L., MELANCHUK, T., AND WORS, R. J. D. The algebraic specification language LOTOS: an industrial experience. *ACM SIGSOFT Software Engineering Notes 15* (04 1990), 59–66.

114. MAHONEY, B. P., AND HAYES, I. J. A case study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering 18* (09 1992), 817–825.

115. MANNA, Z., AND PNUELI, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, ISBN 0-387-97664-7, 1996.

116. MCCABE, T. J. A complexity measure. *IEEE Transaction on Software Engineering SE-2*, 4 (Dec. 1976), 308–320.

117. MEHMET, A., AND WANLI, M. An overview of temporal logic programming. In *First International Conference, ICTL'94, Lecture Notes in AI* (1994), vol. 827, Springer-Verlag, pp. 445–481.

118. MERLIN, P. M., AND SEGALL, A. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications* (Sept. 1976), 1036–1043.

119. MILI, A., MILI, R., AND MITTERMEIR, R. Storing and retrieving software components: A refinement-based system. In *Proceedings of the 16th International Conference on Software Engineering* (May 1994), IEEE Computer Society Press, pp. 91–102.

120. MILLER, H. A., ORGUN, M. A., TILLEY, S. R., AND UHL, J. S. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice 5* (Dec. 1993), 181–204.

121. MILNER, R. A calculus of communicating systems. *LNCS 90* (1980).

122. MILNER, R. Some directions in concurrency theory(panel statement). In *Proceedings of the international Conference on the fifth Generation Computer Systems* (1988).

123. MILNER, R. *Communication and Concurrency.* Prentice Hall, Hertfordshire, 1989.

124. MOSZKOWSKI, B. *A Temporal Logic for Multilevel Reasoning about Hardware.* IEEE Computer Society, Feb. 1985.

125. MOSZKOWSKI, B. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge UK, 1986.

126. MYLOPOULOS, J., STANLEY, M., WONG, K., AND ET AL. Towards an integrated toolset for program understanding. In *CAS Conference 1994 Proceedings (CASCON 1994)* (1994), pp. 19–31.

127. NARAT, V. Using a relational database for software maintenance: A case study. In *Proceedings of the International Conference on Software Maintenance* (1993), IEEE Computer Society Press, pp. 244–245.

128. NARAYANA, K. T., AND AABY, A. A. Specification of real-time systems in real-time temporal interval logic. In *Proceedings of Real-Time Systems Symposium* (Dec. 1988), IEEE Computer Society, pp. 86–95.

129. OSTROFF, J. S. Temporal logic for real-time systems. *Advanced Software Development Series* (1989).

130. OSTROFF, J. S. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems 1* (Apr. 1990), 170–183.

131. OSTROFF, J. S., AND WONHAM, W. M. A temporal logic approach to real-time control. In *Proceedings of the 24th IEEE Conference on Decision and Control* (Florida, Dec. 1985), pp. 656–657.

132. PAUL, S., AND PRAKASH, A. Querying source code using an algebraic query language. In *Proceedings of the International Conference on Software Maintenance* (1994), IEEE Computer Society Press, pp. 127–136.

133. PETERSON, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

134. RAMCHANDANI, C. Analysis of asynchronous concurrent systems by timed petri nets. *Technical Report* (Feb. 1974).

135. RAZOUK, R. R., AND PHELPS, C. V. Performance analysis of timed petri nets. In *Proceedings of 4th International Workshop on Protocol Verification and Testing* (June 1984).

136. REED, G. M., AND ROSCOE, A. W. Timed CSP: Theory and practice. In *REX Workshop—Real-Time : Theory and Practice* (1992), LNCS Springer-Verlag.

137. REISIG, W. *Petri Nets: an Introduction*. Springer-Verlag, Berlin, 1985.

138. RESCHER, N., AND URQUHART, A. Temporal logic. *Library of Exact Philosophy* (1971).

139. RUGABER, S., ORNBURN, S. B., AND JR. RICHARD J. LEBLANC. Recognizing design decisions in programs. *IEEE Software* (Jan. 1990), 46–54.

140. SCHNEIDER, G. M. *Advanced Programming and Problem Solving with PASCAL.* John Wiley & Sons INC, 1987.

141. SCHNEIDEWIND, N. F., AND EBERT, C. Preserve or redesign legacy systems. *IEEE Software 15*, 4 (1998).

142. SCHOLEFIELD, D. A refinement calculus for real-time systems. *PhD thesis* (1992).

143. SCHOLEFIELD, D., AND ZEDAN, H. TAM: A temporal agent model for distributed real-time systems. In *EUROMICRO'90 Workshop* (North Holland, Aug. 1990).

144. SCHOLEFIELD, D., AND ZEDAN, H. TAM: A formal framework for the development of distributed real-time systems. In *Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems* (Nijmegen, Netherland, Jan. 1992).

145. SCHOLEFIELD, D., ZEDAN, H., AND HE, J. A specification-oriented semantics for the refinement of real-time systems. *Theoretical Computer Science 130* (Aug. 1994).

146. SERE, K., AND WALDÉN, M. Reverse engineering distributed algorithms. *Software Maintenance: Research and Practice 8* (1996), 117–144.

147. SITARAMAN, M., WEIDE, B. W., AND OGDEN, W. F. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering 23*, 3 (March 1997).

148. SNEED, H. M. Planning the reengineering of legacy systems. *IEEE Software 12*, 1 (January 1995).

149. SNEED, H. M., AND JANDRASICS, G. Inverse transformation of software from code to specification. In *IEEE Conference on Software Maintenance-1988* (Phoenix, Arizona, 1988).

150. SOLOWAY, E., AND EHRLICH, K. Empirical studies of programming knowledge. *IEEE Transaction on Software Engineering SE-10*, 5 (Sept. 1984), 595–609.

151. SOWMYA, A., AND RAMESH, S. Extending statecharts with temporal logic. *IEEE Transaction on Software Engineering 24*, 3 (Mar. 1998), 216–227.

152. SPIVEY, J. M. *Understanding Z*. Cambridge University Press, 1988.

153. SWANSON, E. B. The dimension of maintenance. In *Second International Conference on Software Engineering* (Los Alamitos, California, 1976), IEEE Computer Society.

154. TILBORG, A. M. V., AND KOOB, G. M. *Foundations of Real-Time Computing–Formal Specification and Methods*. Kluwer Academic Publishers, ISBN 0-7923-9167-5., 1991.

155. TILLEY, S. R., WONG, K., STOREY, M. A. D., , AND MILLER, H. A. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering* (Dec. 1994), 501–520.

156. TURNER, K. J. DILL-digital logic in LOTOS. *Formal Description Techniques, FORTE VII* (Oct. 1993).

157. VAN EIJK, P. H. J., VISSERS, C. A., AND DIAZ, M. *The Formal Description Technique LOTOS*. Elsevier Science Publishers, 1989.

158. WARD, M. A definition of abstraction. Technical report, Durham University, 1992.

159. WARD, M. Abstracting a specification from code. *Software Maintenance: Research and Practice 5* (1993), 101–122.

160. WARD, M., MUNRO, M., AND CALLISS, F. W. The maintainer's assistant. In *IEEE Conference on Software Maintenance-1989* (Miami, Florida, 1989), pp. 307–315.

161. WILSON, G. V. *Practical Parallel Programming*. The MIT Press, ISBN 0-262-23186-7, 1995.

162. WORDSWORTH, J. *Software Engineering with B*. Addison Wesley Longman, ISBN 0-201-40356-0., 1996.

163. YANG, H. The supporting environment for A reverse engineering system — the maintainer's assistant. In *IEEE Conference on Software Maintenance-1991* (Sorrento, Italy, Oct. 1991), pp. 13–22.

164. YANG, H. *Acquiring Data Designs from Existing Data Intensive Programs*. Ph.D. thesis, Durham University, 1994.

165. YANG, H. Formal methods and software maintenance — some experience with the REFORM project. In *Workshop on Formal Methods, Position Paper* (Montery, USA, Sept. 1994).

166. YANG, H., AND BENNETT, K. H. Extension of A transformation system for maintenance — dealing with data-intensive programs. In *IEEE International Conference on Software Maintenance (ICSM '94)* (Victoria, Canada, Sept. 1994).

167. YANG, H., AND BENNETT, K. H. Aquairing entity-relationship attribute diagrams from code and data through program transformation. In *IEEE International Conference on Software Maintenance (ICSM '95)* (Nice, France, Oct. 1995).

168. YANG, H., LIU, X., AND ZEDAN, H. Tackling the abstraction problem for reverse engineering in a system re-engineering approach. In *the proceedings of the IEEE Conference on Software Maintenance (ICSM'98)* (Metropolitan, Washington D.C., USA, November 1998), IEEE Computer Society.

169. YANG, H., AND LUKER, P. Measuring abstractness for reverse engineering in a re-engineering tool. In *IEEE International Conference on Software Maintenance* (Bari, Italy, October 1997).

170. ZEDAN, H., AND HEPING, H. An executable specification language for fast prototyping parallel responsive systems. *Computer Language Vol 22* (01 1996), 1–13.

171. ZHOU, S., YANG, H., LUKER, P., AND HE, X. A useful approach to developing reverse engineering metrics. In *IEEE Computer Software and Application Conference (COMPSAC'99)* (Arizona, USA, Oct. 1999).

172. ZUSE, H. *Software Complexity — Measures and Methods*. Walter de Gruyter, New York, 1991.

Appendix A

Proofs

A.1 Monotonicity of Abstractions

Assume $\mathcal{D} \succeq_f \mathcal{C}$ and $\mathcal{A} \succeq_f \mathcal{B}$. Let $\odot$ be a context, $\odot$ is monitonic with respect to $\succeq_f$ if $\mathcal{D} \odot \mathcal{A} \succeq_f \mathcal{C} \odot \mathcal{B}$ where $\odot = \wedge \mid \vee \mid ; \mid \| \mid \Rightarrow$(conclusion part). Since $\|= \wedge$, $\|$ does not need to be proved separately.

The proof is as follows:

f=WA: $\odot$ is monotonic with respect to $\succeq_{WA}$.

$$
\begin{aligned}
[\![\mathcal{D} \wedge \mathcal{A}]\!] &= [\![\mathcal{D}\,]\!] \wedge [\![\,\mathcal{A}]\!] \\
&\Rightarrow [\![\mathcal{D}\,]\!] \wedge [\![\,\mathcal{B}]\!] \qquad \text{(with } [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]\text{)} \\
&\Rightarrow [\![\mathcal{C}\,]\!] \wedge [\![\,\mathcal{B}]\!] \qquad \text{(with } [\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!]\text{)} \\
&\Rightarrow [\![\mathcal{C} \wedge \mathcal{B}]\!]
\end{aligned}
$$

hence, $\mathcal{D} \wedge \mathcal{A} \succeq_{WA} \mathcal{C} \wedge \mathcal{B}$

$$
\begin{aligned}
[\![\mathcal{D} \vee \mathcal{A}]\!] &= [\![\mathcal{D}\,]\!] \vee [\![\,\mathcal{A}]\!] \\
&\Rightarrow [\![\mathcal{D}\,]\!] \vee [\![\,\mathcal{B}]\!] \qquad \text{(with } [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]\text{)} \\
&\Rightarrow [\![\mathcal{C}\,]\!] \vee [\![\,\mathcal{B}]\!] \qquad \text{(with } [\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!]\text{)} \\
&\Rightarrow [\![\mathcal{C} \vee \mathcal{B}]\!]
\end{aligned}
$$

hence, $\mathcal{D} \vee \mathcal{A} \succeq_{WA} \mathcal{C} \vee \mathcal{B}$

$$[\![\mathcal{D}\ ;\ \mathcal{A}]\!] = [\![\mathcal{D}]\!]\ ;\ [\![\mathcal{A}]\!]$$
$$\Rightarrow [\![\mathcal{D}]\!]\ ;\ [\![\mathcal{B}]\!] \qquad (\text{with } [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$$
$$\Rightarrow [\![\mathcal{C}]\!]\ ;\ [\![\mathcal{B}]\!] \qquad (\text{with } [\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!])$$
$$\Rightarrow [\![\mathcal{C}\ ;\ \mathcal{B}]\!]$$

hence, $\mathcal{D}\ ;\ \mathcal{A} \succeq_{WA} \mathcal{C}\ ;\ \mathcal{B}$

$$[\![\mathcal{D} \Rightarrow \mathcal{A}]\!] = \neg\ [\![\ \mathcal{D}\ ]\!] \vee ([\![\mathcal{D}\ ]\!] \wedge [\![\ \mathcal{A}]\!])$$
$$\Rightarrow \neg\ [\![\ \mathcal{D}\ ]\!] \vee ([\![\mathcal{D}\ ]\!] \wedge [\![\ \mathcal{B}]\!]) \qquad (\text{with } [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$$
$$\Rightarrow [\![\mathcal{D} \Rightarrow \mathcal{B}]\!]$$

hence, $\mathcal{D} \Rightarrow \mathcal{A} \succeq_{WA} \mathcal{D} \Rightarrow \mathcal{B}$

$$[\![\mathcal{C} \Rightarrow \mathcal{A}]\!] \Rightarrow [\![\mathcal{C} \Rightarrow \mathcal{B}]\!] \quad (\text{with } [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$$
$$\Rightarrow [\![\mathcal{D} \Rightarrow \mathcal{B}]\!] \quad (\text{with } [\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!])$$

hence, $\mathcal{C} \Rightarrow \mathcal{A} \succeq_{WA} \mathcal{D} \Rightarrow \mathcal{B}$

Hence, $\mathcal{D} \odot \mathcal{A} \succeq_{WA} \mathcal{C} \odot \mathcal{B}$ for $\odot = \wedge \mid \vee \mid ; \mid \|$

and $\mathcal{D} \odot \mathcal{A} \succeq_{WA} \mathcal{D} \odot \mathcal{B}$ for $\odot = \Rightarrow$

f=TA: $\odot$ is monotonic with respect to $\succeq_{TA}$ if $\odot$ is monotonic with respect to R_T;

Assuming that

$$\mathcal{A} \succeq_{TA} \mathcal{B} \stackrel{\triangle}{=} ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]) \wedge R_T(T(\mathcal{A}), T(\mathcal{B}))$$
$$\mathcal{D} \succeq_{TA} \mathcal{C} \stackrel{\triangle}{=} ([\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!]) \wedge R_T(T(\mathcal{D}), T(\mathcal{C}))$$

Hence, we have $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$, i.e., $\mathcal{A} \succeq_{WA} \mathcal{B}$ as part of the definition,

and, $[\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!]$, i.e., $\mathcal{D} \succeq_{WA} \mathcal{C}$ as part of the definition.

Since WA is monotonic over $\odot$,

Hence, $[\![(\mathcal{D} \odot \mathcal{A})]\!] \Rightarrow [\![(\mathcal{C} \odot \mathcal{B})]\!]$

From the above definitions, we have $R_T(T(\mathcal{A}), T(\mathcal{B}))$ and $R_T(T(\mathcal{D}), T(\mathcal{C}))$

For the R_T over which $\odot$ is monotonic, $R_T(T(\mathcal{D} \odot \mathcal{A}), T(\mathcal{C} \odot \mathcal{B}))$ is also true,

Hence, $\mathcal{D} \odot \mathcal{A} \succeq_{TA} \mathcal{C} \odot \mathcal{B}$

f=SA: $\odot$ is monotonic with respect to $\succeq_{SA}$;

Assuming

$$\mathcal{D} \succeq_{SA} \mathcal{C} \triangleq [\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!] \text{ and } \#\textit{seq-op}(\mathcal{D}) > \#\textit{seq-op}(\mathcal{C})$$
$$\text{or } \#\textit{par-op}(\mathcal{D}) > \#\textit{par-op}(\mathcal{C}), \text{ and}$$
$$\mathcal{A} \succeq_{SA} \mathcal{B} \triangleq [\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!] \text{ and } \#\textit{seq-op}(\mathcal{A}) > \#\textit{seq-op}(\mathcal{B})$$
$$\text{or } \#\textit{par-op}(\mathcal{A}) > \#\textit{par-op}(\mathcal{B})$$

Hence, we have $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$, i.e., $\mathcal{A} \succeq_{WA} \mathcal{B}$ as part of the definition,

and, $[\![\mathcal{D}]\!] \Rightarrow [\![\mathcal{C}]\!]$, i.e., $\mathcal{D} \succeq_{WA} \mathcal{C}$ as part of the definition.

Since WA is monotonic over $\odot$,

Hence, $[\![(\mathcal{D} \odot \mathcal{A})]\!] \Rightarrow [\![(\mathcal{C} \odot \mathcal{B})]\!]$

From the above definitions, we have

$$\#\textit{seq-op}(\mathcal{D}) > \#\textit{seq-op}(\mathcal{C}) \text{ or } \#\textit{par-op}(\mathcal{D}) > \#\textit{par-op}(\mathcal{C}), \text{ and}$$
$$\#\textit{seq-op}(\mathcal{A}) > \#\textit{seq-op}(\mathcal{B}) \text{ or } \#\textit{par-op}(\mathcal{A}) > \#\textit{par-op}(\mathcal{B})$$

Hence, there must be:

$$\#\textit{seq-op}(\mathcal{D} \odot \mathcal{A}) > \#\textit{seq-op}(\mathcal{C} \odot \mathcal{B}), \text{ or}$$
$$\#\textit{par-op}(\mathcal{D} \odot \mathcal{A}) > \#\textit{par-op}(\mathcal{C} \odot \mathcal{B})$$

Hence, combine the two sub-conclusions, we have:

$$\mathcal{D} \odot \mathcal{A} \succeq_{SA} \mathcal{C} \odot \mathcal{B}$$

f=HA:

Assume $\mathcal{A} \succeq_{HA} \mathcal{B} \triangleq (\exists x \bullet [\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$ and

$$\mathcal{D} \succeq_{HA} \mathcal{C} \triangleq (\exists y \bullet [\![\mathcal{D}]\!]) \Rightarrow [\![\mathcal{C}]\!]$$

Since WA is monotonic with respect to $\odot$

Hence, $[\![(\mathcal{D} \odot \mathcal{A})]\!] \Rightarrow [\![(\mathcal{C} \odot \mathcal{B})]\!]$

Since $\exists$ is extensible over $\vee$ and ;

Hence, $(\exists x, y \bullet [\![\mathcal{D} \vee \mathcal{A}]\!]) \Rightarrow [\![\mathcal{C} \vee \mathcal{B}]\!]$

and $(\exists x, y \bullet [\![\mathcal{D} \ ; \ \mathcal{A}]\!]) \Rightarrow [\![\mathcal{C} \ ; \ \mathcal{B}]\!]$

However, since $\exists$ is not extensible over $\wedge$ and therefore $\Rightarrow$

Hence, $(\exists x, y \bullet [\![\mathcal{D} \wedge \mathcal{A}]\!]) \Rightarrow [\![\mathcal{C} \wedge \mathcal{B}]\!]$ is false

and, $(\exists x, y \bullet [\![\mathcal{D} \Rightarrow \mathcal{A}]\!]) \Rightarrow [\![\mathcal{C} \Rightarrow \mathcal{B}]\!]$ is false

f=DA on r: $\odot$ is not monotonic with respect to $\succeq_{DA}$.

Here is a counter example to the monotonicity.

Assume that $\mathcal{A}_1 \succeq_{DA-r_1} \mathcal{A}'_1$, and $\mathcal{A}_2 \succeq_{DA-r_2} \mathcal{A}'_2$.

However, r_1 is not applicable on $\mathcal{A}_2$ and r_2 is not applicable on $\mathcal{A}_1$.

Hence, $\mathcal{A}_1 \not\succeq_{DA-r_2} \mathcal{A}''_1$, and $\mathcal{A}_2 \not\succeq_{DA-r_1} \mathcal{A}''_2$,

here $\mathcal{A}''_1$ and $\mathcal{A}''_2$ represent any possible representations.

Therefore, $\mathcal{A}_1 \odot \mathcal{A}_2 \not\succeq_{DA-r_1} \mathcal{A}'_1 \odot \mathcal{A}''_2$ and

$$\mathcal{A}_1 \odot \mathcal{A}_2 \not\succeq_{DA-r_2} \mathcal{A}''_1 \odot \mathcal{A}'_2.$$

Hence, both $\mathcal{CX}(\mathcal{A}_1, \mathcal{A}_2) \succeq_{DA-r_1} \mathcal{CX}(\mathcal{A}'_1, \mathcal{A}''_2)$ and

$$\mathcal{CX}(\mathcal{A}_1, \mathcal{A}_2) \succeq_{DA-r_2} \mathcal{CX}(\mathcal{A}''_1, \mathcal{A}'_2)$$

are false.

A.2 Relations between Abstractions

Proof:

1. If $\mathcal{A} \succeq_{TA} \mathcal{B}$ then $\mathcal{A} \succeq_{WA} \mathcal{B}$

Since $\mathcal{A} \succeq_{TA} \mathcal{B} \triangleq ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]) \wedge R_T(T(\mathcal{A}), T(\mathcal{B}))$

Hence, $\mathcal{A} \succeq_{TA} \mathcal{B} \Rightarrow ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$

From the definition of weakening abstraction: $\mathcal{A} \succeq_{WA} \mathcal{B} \triangleq ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$,

therefore, $\mathcal{A} \succeq_{TA} \mathcal{B} \Rightarrow \mathcal{A} \succeq_{WA} \mathcal{B}$

2. There are two kinds of SA:

   (a) Structural abstraction on sequential composition:

   $\mathcal{C}' \succeq_{SA} \mathcal{C} \triangleq [\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$ and $\#seq\text{-}op(\mathcal{C}') > \#seq\text{-}op(\mathcal{C})$, and

   (b) Structural abstraction on parallel composition:

   $\mathcal{C}' \succeq_{SA} \mathcal{C} \triangleq [\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$ and $\#par\text{-}op(\mathcal{C}') > \#par\text{-}op(\mathcal{C})$.

   Both the definitions have $[\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!]$ as a part of them.

   Since weakening abstraction is defined as: $\mathcal{C}' \succeq_{WA} \mathcal{C} \triangleq ([\![\mathcal{C}']\!] \Rightarrow [\![\mathcal{C}]\!])$

   hence, $\mathcal{C}' \succeq_{SA} \mathcal{C} \Rightarrow \mathcal{C}' \succeq_{WA} \mathcal{C}$

3. HA:

   From HA definition $\mathcal{A} \succeq_{HA} \mathcal{B} \triangleq (\exists x \bullet [\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$,

   we have: $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$

   Since weakening abstraction is defined as: $\mathcal{A} \succeq_{WA} \mathcal{B} \triangleq ([\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!])$

   Hence, $\mathcal{A} \succeq_{HA} \mathcal{B} \Rightarrow \mathcal{A} \succeq_{WA} \mathcal{B}$

4. DA:

   When r is recursive, DA degrades to WA. However, this contradicts with the healthiness obligation and is not allowed in practical reverse engineering.

   DA is defined as: $\mathcal{A} \succeq_{DA-r} \mathcal{B} \triangleq r([\![\mathcal{A}]\!]) \Rightarrow [\![\mathcal{B}]\!]$

where $r = \{(x, y) : x \in X, y \in Y, X = \{states\ of\ \mathcal{A}\}, Y = \{states\ of\ \mathcal{B}\}\}$.

If $r = \{(x, y) : x = y \wedge x \in X, y \in Y, X = \{states\ of\ \mathcal{A}\}, Y = \{states\ of\ \mathcal{B}\}\}$

then $r([\![\mathcal{A}]\!]) = [\![\mathcal{A}]\!]$

Hence, $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$

Hence, $\mathcal{A} \succeq_{WA} \mathcal{B}$

A.3 Further Abstraction Rules

The following proof is based on weakening abstraction, that is, $\succeq$ is assumed as $\succeq_{WA}$.

1. Transitive

$\mathcal{A} \succeq \mathcal{B}$

$\mathcal{B} \succeq \mathcal{C}$

$\mathcal{A} \succeq \mathcal{C}$

This rule indicates that abstraction relations are transitive.

Proof:

Since $\mathcal{A} \succeq \mathcal{B}$, we have $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{B}]\!]$

Since $\mathcal{B} \succeq \mathcal{C}$, we have $[\![\mathcal{B}]\!] \Rightarrow [\![\mathcal{C}]\!]$

Hence, $[\![\mathcal{A}]\!] \Rightarrow [\![\mathcal{C}]\!]$

Hence, $\mathcal{A} \succeq \mathcal{C}$

2. Monotonic

$\mathcal{A} \succeq \mathcal{B}$

$\mathcal{D} \succeq \mathcal{C}$

$\mathcal{CX} = \wedge | \vee | \ ; \ | \ \|$

$\mathcal{CX}(\mathcal{A}, \mathcal{D}) \succeq \mathcal{CX}(\mathcal{B}, \mathcal{C})$

The proof is given in the appendix "Monotonicity of Abstraction".

3. Sequence Folding

$$[\![\mathcal{A}\ ;\ \mathcal{B}]\!] \Rightarrow [\![\mathcal{A} \wedge \mathcal{B}]\!]$$

$$\mathcal{A}\ ;\ \mathcal{B} \succeq \mathcal{A} \wedge \mathcal{B}$$

If no contradiction is caused when substituting the sequential composition between two representations to conjunction composition, then the sequence can be folded through conjunction.

Proof:

From the premise, $[\![\mathcal{A}\ ;\ \mathcal{B}]\!]$ has no contradiction with $[\![\mathcal{A} \wedge \mathcal{B}]\!]$

Therefore, we have $[\![\mathcal{A}\ ;\ \mathcal{B}]\!] \Rightarrow [\![\mathcal{A} \wedge \mathcal{B}]\!]$

Hence, $\mathcal{A}\ ;\ \mathcal{B} \succeq \mathcal{A} \wedge \mathcal{B}$

4. Specification Combination

$$(W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (W_1 \cup W_2) : \Phi_1 \wedge \Phi_2$$

$$(W_1 : \Phi_1) \vee (W_2 : \Phi_2) \succeq (W_1 \cup W_2) : \Phi_1 \vee \Phi_2$$

Proof:

Since $W : f \triangleq frame(W) \wedge f$

Hence $(W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (frame(W_1) \wedge \Phi_1) \wedge (frame(W_2) \wedge \Phi_2)$

Hence $(W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (frame(W_1) \wedge frame(W_2)) \wedge (\Phi_1 \wedge \Phi_2)$

Hence $(W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (frame(W_1 \cup W_2)) \wedge (\Phi_1 \wedge \Phi_2)$

Hence $(W_1 : \Phi_1) \wedge (W_2 : \Phi_2) = (W_1 \cup W_2) : \Phi_1 \wedge \Phi_2$

Since $W : f \triangleq frame(W) \wedge f$

Hence $(W_1 : \Phi_1) \vee (W_2 : \Phi_2) = (frame(W_1) \wedge \Phi_1) \vee (frame(W_2) \wedge \Phi_2)$

Since $(frame(W_1) \wedge \Phi_1) \vee (frame(W_2) \wedge \Phi_2) \Rightarrow$

$$(frame(W_1) \wedge frame(W_2)) \wedge (\Phi_1 \vee \Phi_2)$$

Hence $(W_1 : \Phi_1) \vee (W_2 : \Phi_2) \Rightarrow (frame(W_1 \cup W_2)) \wedge (\Phi_1 \vee \Phi_2)$

Hence $(W_1 : \Phi_1) \vee (W_2 : \Phi_2) \Rightarrow (W_1 \cup W_2) : \Phi_1 \vee \Phi_2$

Hence $(W_1 : \Phi_1) \vee (W_2 : \Phi_2) \succeq (W_1 \cup W_2) : \Phi_1 \vee \Phi_2$

5. Weakening

This is a quite general abstraction rule, which includes the following sub rules:

- State Test and Exception Handling
- User Interface Format
- Semantic Core
- Concise Specification
- Comment Revise
- Trivial Elements
- Domain Function
- Efficiency-Improving Details

The general formal representation of these rule is as follows:

$\mathcal{A} \succeq \Phi$

$\Phi \Rightarrow \Psi$

$\mathcal{A} \succeq \Psi$

Proof:

Since $\mathcal{A} \succeq \Phi$, hence $[\![\mathcal{A}]\!] \Rightarrow \Phi$

From the premise, we have $\Phi \Rightarrow \Psi$

Hence, $[\![\mathcal{A}]\!] \Rightarrow \Psi$

Hence, $\mathcal{A} \succeq \Psi$

6. Conjunction

$$\frac{\mathcal{A} \succeq \Phi \qquad \mathcal{A} \succeq \Psi}{\mathcal{A} \succeq \Phi \wedge \Psi}$$

Proof:

Since $\mathcal{A} \succeq \Phi$, hence $[\![\mathcal{A}]\!] \Rightarrow \Phi$

Since $\mathcal{A} \succeq \Psi$, hence $[\![\mathcal{A}]\!] \Rightarrow \Psi$

Hence, $[\![\mathcal{A}]\!] \Rightarrow \Phi \wedge \Psi$

Hence, $\mathcal{A} \succeq \Phi \wedge \Psi$

7. Specification

$$(W : \Phi) \wedge \mathsf{stable}\,(s) = (W - s) : \Phi \quad (\text{if } s \text{ not in } \Phi)$$

Proof:

Since $W : f \triangleq \mathit{frame}(W) \wedge f$

Hence $(W : \Phi) \wedge \mathsf{stable}\,(s) = \mathit{frame}(W) \wedge \Phi \wedge \mathsf{stable}\,(s)$

Since $\mathsf{stable}\,(s)$ and s not in Φ

Hence $\mathit{frame}(W) \wedge \Phi \wedge \mathsf{stable}\,(s) = \mathit{frame}(W - s) \wedge \Phi$

Hence $(W : \Phi) \wedge \mathsf{stable}\,(s) = \mathit{frame}(W - s) \wedge \Phi$

Hence $(W : \Phi) \wedge \mathsf{stable}\,(s) = (W - s) : \Phi$

This rule eliminates the redundant variables in a specification.

8. Sequential

8.1 $\mathbf{empty}\ ;\ \mathcal{A} = \mathcal{A} = \mathcal{A}\ ;\ \mathbf{empty}$

8.2 $\mathcal{A}\ ;\ (\mathcal{B}\ ;\ \mathcal{C}) = (\mathcal{A}\ ;\ \mathcal{B})\ ;\ \mathcal{C}$

8.3 $\mathcal{A}_1 \; ; \; (\mathcal{A}_2 \vee \mathcal{A}_3) \; ; \; \mathcal{A}_4 = (\mathcal{A}_1 \; ; \; \mathcal{A}_2 \; ; \; \mathcal{A}_4) \vee (\mathcal{A}_1 \; ; \; \mathcal{A}_3 \; ; \; \mathcal{A}_4)$

Proof:

empty $; \; \mathcal{A} = \mathcal{A} = \mathcal{A}$; **empty** is an ITL axiom.

$\mathcal{A} \; ; \; (\mathcal{B} \; ; \; \mathcal{C}) = \mathcal{A} \; ; \; \mathcal{B} \; ; \; \mathcal{C}$

$(\mathcal{A} \; ; \; \mathcal{B}) \; ; \; \mathcal{C} = \mathcal{A} \; ; \; \mathcal{B} \; ; \; \mathcal{C}$

Hence, $\mathcal{A} \; ; \; (\mathcal{B} \; ; \; \mathcal{C}) = (\mathcal{A} \; ; \; \mathcal{B}) \; ; \; \mathcal{C}$

From the formal semantics of $f_1 \; ; \; f_2$, there must be

$\mathcal{A}_1 \; ; \; (\mathcal{A}_2 \vee \mathcal{A}_3) \; ; \; \mathcal{A}_4 = (\mathcal{A}_1 \; ; \; \mathcal{A}_2 \; ; \; \mathcal{A}_4) \vee (\mathcal{A}_1 \; ; \; \mathcal{A}_3 \; ; \; \mathcal{A}_4)$

These rules indicate that sequential composition operator has empty as a unit and is associative and distributive over nondeterministic choice.

9. Delay

$\mathsf{delay}_{d_1} ; \; \mathsf{delay}_{d_2} = \mathsf{delay}_{d_1+d_2}$

$\mathbf{skip} = \mathsf{delay}_1$

Proof:

Since $\mathsf{delay}_d \stackrel{\wedge}{=} len = d$

Hence $\mathsf{delay}_{d_1} ; \; \mathsf{delay}_{d_2} = (len = d_1 \; ; \; len = d_2)$

$= (len = d_1 + d_2)$

Since $\mathsf{delay}_{d_1+d_2} = (len = d_1 + d_2)$

Hence $\mathsf{delay}_{d_1} ; \; \mathsf{delay}_{d_2} = \mathsf{delay}_{d_1+d_2}$

$\mathsf{skip} \stackrel{\wedge}{=} len = 1$

$\mathsf{delay}_1 = (len = 1)$

Hence $\mathsf{skip} = \mathsf{delay}_1$

10. Parallel

10.1 $\mathcal{A} \parallel \mathcal{B} = \mathcal{B} \parallel \mathcal{A}$

10.2 $\mathcal{A} \parallel (\mathcal{B} \parallel \mathcal{C}) = (\mathcal{A} \parallel \mathcal{B}) \parallel \mathcal{C}$

10.3 $\mathcal{A} \parallel \mathit{true} = \mathcal{A}$

10.4 $\mathcal{A} \parallel (\mathcal{B} \vee \mathcal{C}) = (\mathcal{A} \parallel \mathcal{B}) \vee (\mathcal{A} \parallel \mathcal{C})$

10.5 $\mathcal{A} \parallel \mathcal{B} \succeq \mathcal{A}' \parallel \mathcal{B}$, for any $\mathcal{B}$ if $\mathcal{A} \succeq \mathcal{A}'$

10.6 $(G \Rightarrow \Phi_1) \parallel (G' \Rightarrow \Phi_2) \succeq (G \wedge G') \Rightarrow \Phi_1 \wedge \Phi_2$

Proof:

The basis of the proof of the above rules is $\mathcal{P} \parallel \mathcal{Q} \stackrel{\wedge}{=} \mathcal{P} \wedge \mathcal{Q}$

$$\begin{aligned}\mathcal{A} \parallel \mathcal{B} &= \mathcal{A} \wedge \mathcal{B} \\ &= \mathcal{B} \wedge \mathcal{A} \\ &= \mathcal{B} \parallel \mathcal{A}\end{aligned}$$

$$\begin{aligned}\mathcal{A} \parallel (\mathcal{B} \parallel \mathcal{C}) &= \mathcal{A} \wedge (\mathcal{B} \wedge \mathcal{C}) \\ &= (\mathcal{A} \wedge \mathcal{B}) \wedge \mathcal{C} \\ &= (\mathcal{A} \parallel \mathcal{B}) \parallel \mathcal{C}\end{aligned}$$

$$\begin{aligned}\mathcal{A} \parallel \mathit{true} &= \mathcal{A} \wedge \mathit{true} \\ &= \mathcal{A}\end{aligned}$$

$$\begin{aligned}\mathcal{A} \parallel (\mathcal{B} \vee \mathcal{C}) &= \mathcal{A} \wedge (\mathcal{B} \vee \mathcal{C}) \\ &= (\mathcal{A} \wedge \mathcal{B}) \vee (\mathcal{A} \wedge \mathcal{C}) \\ &= (\mathcal{A} \parallel \mathcal{B}) \vee (\mathcal{A} \parallel \mathcal{C})\end{aligned}$$

$\mathcal{A} \parallel \mathcal{B} = \mathcal{A} \wedge \mathcal{B}$

From premise, there is $\mathcal{A} \succeq \mathcal{A}'$

Hence, $\mathcal{A} \Rightarrow \mathcal{A}'$

Hence, $\mathcal{A} \wedge \mathcal{B} \Rightarrow \mathcal{A}' \wedge \mathcal{B}$

Hence, $\mathcal{A} \wedge \mathcal{B} \succeq \mathcal{A}' \wedge \mathcal{B}$

Hence, $\mathcal{A} \parallel \mathcal{B} \succeq \mathcal{A}' \parallel \mathcal{B}$

$$(G \Rightarrow \Phi_1) \parallel (G' \Rightarrow \Phi_2) = (G \Rightarrow \Phi_1) \wedge (G' \Rightarrow \Phi_2)$$
$$= (\neg G \vee \Phi_1) \wedge (\neg G' \vee \Phi_2)$$
$$= (\neg G \wedge \neg G') \vee (\neg G \wedge \Phi_2) \vee (\Phi_1 \wedge \neg G') \vee (\Phi_1 \wedge \Phi_2)$$
$$(G \wedge G') \Rightarrow \Phi_1 \wedge \Phi_2 = \neg(G \wedge G') \vee (\Phi_1 \wedge \Phi_2)$$
$$= (\neg G \vee \neg G') \vee (\Phi_1 \wedge \Phi_2)$$

Since $(\neg G \wedge \neg G') \vee (\neg G \wedge \Phi_2) \vee (\Phi_1 \wedge \neg G') \vee (\Phi_1 \wedge \Phi_2) \Rightarrow (\neg G \vee \neg G') \vee (\Phi_1 \wedge \Phi_2)$

Hence $(G \Rightarrow \Phi_1) \parallel (G' \Rightarrow \Phi_2) \Rightarrow ((G \wedge G') \Rightarrow \Phi_1 \wedge \Phi_2)$

Hence $(G \Rightarrow \Phi_1) \parallel (G' \Rightarrow \Phi_2) \succeq (G \wedge G') \Rightarrow \Phi_1 \wedge \Phi_2$

11. Signal

11.1 $(\mathcal{A} \trianglerighteq_s^n \mathcal{B}) \parallel (\mathcal{C} \trianglerighteq_s^n \mathcal{D}) = (\mathcal{A} \parallel \mathcal{C}) \trianglerighteq_s^n (\mathcal{B} \parallel \mathcal{D})$

11.2 $\mathcal{A} \trianglerighteq_s^n (\mathcal{C} \trianglerighteq_s^0 \mathcal{B}) = \mathcal{A} \trianglerighteq_s^n \mathcal{B}$

Proof:

In ITL, $\mathcal{A}_1 \trianglerighteq_s^t \mathcal{A}_2 \stackrel{\Delta}{=} (\Delta t \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{A}_1) \vee (\Delta t \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{A}_2)$

Hence $(\mathcal{A} \trianglerighteq_s^n \mathcal{B}) \parallel (\mathcal{C} \trianglerighteq_s^n \mathcal{D}) = (\mathcal{A} \trianglerighteq_s^n \mathcal{B}) \wedge (\mathcal{C} \trianglerighteq_s^n \mathcal{D})$

$= ((\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{A}) \vee (\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{B})) \wedge$

$((\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{C}) \vee (\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{D}))$

$= (\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{A}) \wedge (\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{C}) \vee$

$(\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{A}) \wedge (\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{D}) \vee$

$(\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{B}) \wedge (\Delta n \wedge \mathsf{stable}(\surd s)\ ;\ \mathcal{C}) \vee$

$(\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{B}) \wedge (\Delta n \wedge \neg\, \mathsf{stable}(\surd s)\ ;\ \mathcal{D})$

Since stable $(\surd s) \wedge (\neg$ stable $(\surd s)) = false$

Hence $(\mathcal{A} \trianglerighteq_s^n \mathcal{B}) \parallel (\mathcal{C} \trianglerighteq_s^n \mathcal{D})$

$= (\Delta n \wedge$ stable $(\surd s)\ ;\ (\mathcal{A} \wedge \mathcal{C})) \vee false \vee false \vee (\Delta n \wedge \neg$ stable $(\surd s)\ ;\ (\mathcal{B} \wedge \mathcal{D}))$

$= (\Delta n \wedge$ stable $(\surd s)\ ;\ (\mathcal{A} \wedge \mathcal{C})) \vee (\Delta n \wedge \neg$ stable $(\surd s)\ ;\ (\mathcal{B} \wedge \mathcal{D}))$

$= (\mathcal{A} \wedge \mathcal{C}) \trianglerighteq_s^n (\mathcal{B} \wedge \mathcal{D})$

$= (\mathcal{A} \parallel \mathcal{C}) \trianglerighteq_s^n (\mathcal{B} \parallel \mathcal{D})$

$\mathcal{A} \trianglerighteq_s^n (\mathcal{C} \trianglerighteq_s^0 \mathcal{B}) = (\Delta n \wedge$ stable $(\surd s)\ ;\ \mathcal{A}) \vee$

$(\Delta n \wedge \neg$ stable $(\surd s)\ ;\ (\Delta 0 \wedge$ stable $(\surd s)\ ;\ \mathcal{C}) \vee (\Delta 0 \wedge \neg$ stable $(\surd s)\ ;\ \mathcal{B}))$

Since shunt s can not change within 0 time unit,

stable $(\surd s)$ will keep its original state when $\Delta 0$

Hence $\mathcal{A} \trianglerighteq_s^n (\mathcal{C} \trianglerighteq_s^0 \mathcal{B}) = (\Delta n \wedge$ stable $(\surd s)\ ;\ \mathcal{A}) \vee (\Delta n \wedge \neg$ stable $(\surd s)\ ;\ \mathcal{B})$

Hence $\mathcal{A} \trianglerighteq_s^n (\mathcal{C} \trianglerighteq_s^0 \mathcal{B}) = \mathcal{A} \trianglerighteq_s^n \mathcal{B}$

12. Non-deterministic choice

12.1 $\mathcal{P} \vee \mathcal{P} = \mathcal{P}$

12.2 $\mathcal{P} \vee \mathcal{Q} = \mathcal{Q} \vee \mathcal{P}$

12.3 $\mathcal{P} \vee (\mathcal{Q} \vee \mathcal{R}) = (\mathcal{P} \vee \mathcal{Q}) \vee \mathcal{R}$

12.4 $true \vee \mathcal{P} = true$

Proof:

Normal first order logic axioms.

13. Iteration

$\mu_{n+1}\,\mathcal{A} = \mathcal{A}\ ;\ \mu_n\,\mathcal{A} = \mu_n\,\mathcal{A}\ ;\ \mathcal{A}$

Proof:

$\mu_{n+1}\,\mathcal{A} = (\mathcal{A})^{\mu_{n+1}}$

$$= (\mathcal{A})^{\mu_n} \; ; \; \mathcal{A}$$

$$= \mu_n \mathcal{A} \; ; \; \mathcal{A}$$

$$\mu_{n+1} \mathcal{A} = (\mathcal{A})^{\mu_{n+1}}$$

$$= \mathcal{A} \; ; \; (\mathcal{A})^{\mu_n}$$

$$= \mathcal{A} \; ; \; (\mu_n \mathcal{A})$$

Appendix B

Code/Specification of Case Studies

B.1 Lexical Scanner

B.1.1 Source Code in PASCAL

The source code of the lexical scanner in section 8.2 is as follows:

```
program scanner(input, output);
const
        debug = false; {debug flag}
        maxcharsperline = 140; {max characters per line}
        maxexponent = 200; {allowable exponent for real numbers}
        quote = ""; {for literal strings}
        tokenlenmax = 80; {token buffer size}
        version = 'scanner0.4-a basic lexical scanner';

type
    tokenclass =
        (delimiter, identifier, integerconstant,
        literal, realconstant, tendoffile, tendofline);
    tokenrec    =
        record
          blankptr    :0..tokenlenmax;
                       {used to blank fill buffer}
        tbptr       :0..tokenlenmax; {index of last char added}
        tokenbuffer:0..packed array[1..tokenlenmax] of char;
        case class :tokenclass of
              integerconstant  :(integervalue: integer);
              realconstant     :(realvalue: real)
    end; {of case and record tokenrec}


errorclass =
    (errnone, erroct, errnodigit, errbigint, errexposize,
    errexpochar, errmissingquote, errlongliteral, errlast);
lineindex  = 0..maxcharsperline;
linebufrec =
```

```
record
  ch         : char; {the line buffer char}
  charptr    : lineindex; {next char to be processed}
  echo       : boolean; {true->echo each line to output}
  endoffile  : boolean; {true->at end of file}
  endofline  : boolean; {true->at end of line}
  errorline  : array[lineindex] of errorclass;
  errorset   : set of errorclass; {for the whole file}
  fileerror  : boolean; {true if the file had an error}
  length     : lineindex; {length of line}
  line       : array[lineindex] of char;
               {the one line buffer}
  linecount  : integer; {counts input lines}
  lineerror  : boolean;
  {set true if an error is found on the line}
  pfrac      : lineindex;
               {ptr to first digit of frac part}
  pint       : line index;
               {ptr to first nonezero char of number}
  pnum       : lineindex;
               {ptr to the first char of a number}
end; {of record linebufferec}


var
    linebuffer     : linebufrec; {a one line buffer}
    token          : tokenrec; {holds a lexical token}

  {initialize

       initialize line buffer
  }

procedure initialize(var linebuffer: linebufrec);
var
       i           : lineindex; {loop index}
begin
    if debug then writeln('initializingh line buffer');
with linebuffer do
begin
    echo:= true; {we will echo the input lines}
    lineerror:= false;
    linecount:= 0;
    for i:= 0 to maxcharsperline do
  begin
       line[i] := '';
       errorline[i] := errnone
    end; {of for}

    errorset:= [];
    fileerror:= false;
    endoffile:= false;
    endofline:= true;
    pnum     := 0
  end {of with}
end; {of procedure initialize}


  {getnextline

       read a new line into the linebuffer
  }

procedure getnextline(var linebuffer: linefufrec);

  {
```

```
      write a line and its line number to output

}

procedure printline(var linebuffer: linebufrec);
var
      i     : lineindex; {loop index}
begin
  with linebuffer do
  begin
      write(linecount: 6,'');
      for i:= 1 to length do write(line[i]);
      writeln
  end
end;  {of procedure printline}


  {printerrorline

      print pointers to errors, add to errorset,
      clear lineerror

  }
procedure printerrorline(var linebuffer: linebufrec);
var
  column : integer; {output column number}
  i      : integer; {loop index}
  j      : integer; {loop index}
  num    : integer; {ord(errclass)}
begin
  column:= 0;
  with linebuffer do
  begin
    printline(linebuffer); {this could be removed later}
    write('*****': 6, ''); {space over line number}
    for i:=1 to length +1 do
      if errorline[i]<>errnone then
      begin
        errorset:= errorset+[errorline[i]];
        num:= ord(errorline[i]); {errornumber}
        if i>column then
        begin
          for j:= column + 2 to i do write(''); {tab}
          write('|');
          column:= i
        end
        else
        begin
          write(',');
          column:= column + 1
        end
        write(num:1); {use a 1 or 2 char field}
        column:= column+1;
        if num>9 then column:= column+1;
        errorline[i]:= errnone
      end;  {of if and for}
    writeln;
    lineerror:= false;
    fileerror:= true
  end  {of with}
end;  {of procedure printerrorline}


begin {of procedure getnextline}
  if debug then writeln('getting new line');
  with linebuffer do
```

```
begin
  if lineerror then printerrorline(linebuffer);
  {last line had errors}
  if not eof(input) then {read the file}
  begin
      length:= 0;
      while not eoln(input) do
      {line overflow assumed impossible}
      begin
        length:= length+1;
        read(input, line[length])
      end;
      readln(input); {get next line so eof can be checked}

      {delete any trailing blanks}
      line[0]:= '*';
      while line[length]='' do length:= length-1;
      line[length+1]:= ''; {ensure endofline returns blank}

      linecount:= linecount+1;
      if echo then printline(linebuffer);

      charptr:= 1;
      ch:= line[charptr];
      endofline:= (charptr>length)
    end {not eof}
    else endoffile:= true
  end  {with}
end; {of procedure getnextline}


  {alphabetic
        function to determine if a character is a letter
  }

  function alphabetic(ch: char): boolean;
  begin
    alphabetic:= ch in ['a'..'z']
  end; {of function alphabetic}


  {numeric

        function to determine if a character is a digit
  }

  function numeric(ch: char): boolean;
  begin
    numeric:= ch in ['0'..'9']
  end; {of function numeric}


  {getnextsymbol

        find next token in linebuffer
  }

  procedure getnextsymbol(var linebuffer: linebufrec;
                          var token: tokenrec);
    {puterror

        place error message at the current buffer pointer
    }

    procedure puterror(error: errorclass;
                       var linebuffer: linebufrec);
```

```
begin
  with linebuffer do
  begin
      lineerror:= true;
      errorline[charptr]:= error
  end
end;  {of procedure puterror}
```

```
{blankfill

      ensure that the token buffer is blank filled
}

procedure blankfill(var token: tokenrec);
begin
  with token do
  begin
    while blankptr>tbptr do
    begin
      tokenbuffer[blankptr]='';
      blankptr:= blankptr-1
    end;
    blankptr:= tbptr
  end  {of with}
end;  {of procedure blankfill}


{getnextcharecter

      read next character from line buffer
      and advance pointer
}

porcedure getnextchar(var linebuffer: linebufrec);
begin
  with linebuffer do
  begin
    if endofline then
      if eof(input) then endoffile:= true
      else getnextline(linebuffer)
    else
    begin
      charptr:= charptr+1;
      if charptr>length then endofline:= true
    end
    ch:= line[charptr]
  end {of with}
end;  {of procedure getnextchar}
```

```
{scanidentifier

      scan alphanumeric characters
      (copying them to tokenbuffer)
}

porcedure scanidentifier(var linebuffer: linebufrec;
                         var token: tokenrec);
begin
  if debug then writeln('scanning identifier');
  with linebuffer, token do
  begin
      class:= identifier;
```

```
      tbptr:= 0;
      repeat  {first char is known to be alphabetic}
        if tbptr<tokenlenmax then
        begin
          tbptr:=tbptr+1;
          tokenbuffer[tpbtr]:= ch
        end;
        getnextchar(linebuffer)
      until not (alphabetic(ch) or numeric(ch))
  end  {with}
end;  {of procedure scanidentifier}


procedure scannumber(var linebuffer: linebufrec;
                     var token: tokenrec);
var
  i        : integer;


{convinteger


 convert part of linebuffer to an integer
 (with no overflow)
}

  procedure convinteger(var linebuffer: linebufrec;
                        base: integer;
                        maxint: integer;
                        first, last:
                        lineindex;
                        var n: integer);
  var
    digit  :0..9; {holds a single digit's worth}
    i      : integer; {loop index}
    x      : real; {used to check for overflow}

  begin
    n:= 0;
    x:= 0.0;
    i:= first;
    while i<last do
    begin
      digit:= ord(linebuffer.line[i])-ord('0');
      if digit >= base then
      begin
        puterror(erroct, linebuffer);
        i:= last {terminate loop}
      end;

      x:= x*base+digit;
      if x<= maxint then n:=n*base+digit
      else
      begin
        puterror(errbigint, linebuffer);
        i:= last
      end;
      i:= i+1
    end {of while}
  end;  {of procedure convinteger}


{scaninteger

      scan a decimal integer
}
```

```
procedure scaninteger(var linebuffer: linebufrec;
                      var token: tokenrec);
begin
  with linebuffer, token do
  begin
    with linebuffer, token do
    begin
      class:= integerconstant;
      convinteger(linebuffer, 10, maxint,
                  pint, charptr, integervalue)
    end
  end;  {of procedure scaninteger}


{scanoctal


      scan an octal number
}

procedure scanoctal(var linebuffer: linebuffrec;
                    var token: tokenrec);
begin
  with linebuffer, token do
  begin
    class:= integerconstant;
    convinteger(linebuffer, 8, maxint, pint,
                charptr, integervalue);
    getnextchar(linebuffer) {skip 'b'}
  end
end;  {of procedure scanoctal}


{scanreal

      scan a real number with/without exponent
}

procedure scanreal(var linebuffer: linebufrec;
                   var token: tokenrec);
var
  expo       : integer;
  fac        : real; {used to compute power of 10}
  i          : integer;
  negexp     : boolean; {true if exponent is <0}
  nexpo      : integer; {normalised exponent}
  r          : real; {used to compute power of 10}
  scale      : integer;
  x          : real; {accumlator}

begin
  if debug then writeln('scanning real number');
  with linebuffer, token do
  begin
    class:= realconstant;

    {do integer part, overflow assumed impossible}
    x:= 0.0;
    expo:= 0;
    for i:= pint to carptr-1 do
      x=x*10.0+ord(line[i]-ord('0');

    nexpo:= charptr-pint;
    scale:= 0;
    if ch='.' then
    begin
      getnextchar(linebuffer); {skip'.'}
```

```
pfrac:= charptr;
if numeric(ch) then
  repeat
    scale:=scale-1;
    x:=x*10.0+ord(ch)-ord('0');
    getnextchar(linebuffer)
  until not numeric(ch)
else puterror(errnodigit, linebuffer);

{check if we must find first nonzero digit}
if nexpo=0 then {integer part was zero}
begin
  i:= pfrac;
  while line[i]= '0' do i:=i+1;
  nexpo:= pfrac-i {=trunc(log10(x))}
end
end;  {fractional part}

{do we have an exponent?}
if ch='e' then
begin
  negexp:= false;
  getnextchar(linebuffer); {skip 'e'}
  if ch='-' then
  begin
    negexp:=true;
    getnextchar(linebuffer) {skip '-'}
  end
  else if ch='+' then getnextchar(linebuffer);

  {build exponent}
  if numeric(ch) then
  begin
    repeat
      expo:= expo*10+ ord(ch)-ord('0');
      getnextchar(linebuffer)
    until not numeric(ch);

    {adjust scale and nexpo}
    if negexp then
    begin
      scale:= scale-expo;
      nexpo:= scale-expo
    end
    else
    begin
      scale:= scale+expo;
      nexpo:= scale+expo
    end
  end
  else puterror(errexpochar, linebuffer)
end; {exponent}

{compute 10**scale using right to
 left binary method}
if abs(nexpo)<=maxexponent then
  if scale<>0 then  {must adjust exponent}
  begin
    r:= 1.0;
    negexp:= scale<0;
    scale:= abs(scale);
    fac:= 10.0;
    repeat
      if odd(scale) then r:=r*fac;
      fac:=sqr(fac);
      scale:= scale div 2
```

```
          until scale=0;
          if negexp then realvalue:= x/r
          else realvalue:= x*r
        end {apply exponent}
        else realvalue:=x
      else puterror(errexposize, linebuffer)
    end {of with}
end;  {of procedure scanreal}


begin {of procedure scannumber}
  if debug then writeln('scanning number');
  with linebuffer, token do
  begin
    tbptr:= 0; {reset token buffer pointer}
    pint:= charptr; {first nonzero char}

    {scan integer part}
    while numeric(ch) do getnextchar(linebuffer);

    if ch<>'b' then
    begin
      if not((ch=',') or (ch='e')) then
              scaninteger(linebuffer, token)
      else scanreal(linebuffer, token)
    end
    else scanoctal(linebuffer, token);

    {copy number into token buffer}
    i:= pnum;
    tbptr:= 0;
    while(i<charptr> and (tbptr<tokenlenmax) do
      begin
        tbptr:= tbptr+1;
        tokenbuffer[tbptr]:= line[i];
        i:=i+1
      end;  {of copy}

      pnum:=0  {enable getnextline}
  end {with}
end;  {of procedure scanumber}

{scanliteral

      read in a literal string
}

procedure scanliteral(var linebuffer: linebufrec;
                      var token: tokenrec);
var
  working  : boolean; {true if the closing quote
                       has not been found}
begin
  if debug then writenln('scanning literal');
  with linebuffer, token do
  begin
    class:= literal;
    tbptr:= 0;
    getnextchar(linebuffer); {skip first quote}
    working:= true;
    while working and not endofline do
    begin
      if ch=quote then {is it two in a row?}
      begin
        getnextchar(linebuffer);
        {if ch is a quote, continue since it
```

```
    is and imbedded one}
  working:= ch= quote
end;
if working then
begin
  if tbptr<tokenlenmax then
  begin
    tbptr:= tbptr+1;
    tokenbuffer[tbptr]:= ch;
    getnextchar(linebuffer)
  end
  else  {string too long}
  begin
    puterror(errlongliteral, linebuffer);
    while (ch<>quote) and not endofline do
        getnextchar(linebuffer);
        {skip over string}
    if ch=quote then
       getnextchar(linebuffer);
    working:= false
  end  {overflow}
 end {of if working}
end;  {of while}
if working then
  puterror(errmissingquote,buffer)
 end {with}
end;  {of procedure scanliteral}


{scandelimiter

      put ch into token buffer and advance
{

procedure scandelimiter(var linebuffer: linebufrec;
                        var token: tokenrec);
begin
  if debug then writeln('scanning delimiter');
  token.class:= delimiter;
  token.tbptr:= 1;
  token.tokenbuffer[token.tbptr]:= linebuffer.ch;
  getnextchar(linebuffer)
end;  {of procedure scandelimiter}


{scanendofline

      return end of line status
}

procedure scanendofline(var linebuffer: linebufrec;
                        var token: tokenrec);
begin
  if debug then writeln('scanning end of line');
  token.class:= tendofline;
  token.tbptr:= 0
end;  {of procedure scan endofline}


{scanfileend

      return end of file status
}

procedure scanfileend(var linebuffer: linebufrec;
                      var token: tokenrec);
```

```
begin
  if debug then writeln('scanning end of file');
  token.class:= tendoffile;
  token.tbptr:= 0
end;  {of procedure scanfileend}



begin  {of procedure getnextsymbol}
  if debug then writeln('getting next symbol.
                         (ch=',linebuffer.ch,')');
  if (token.class=tendofline) or
     (token.class= tendoffile) then
       getnextline(linebuffer);
with linebuffer do
begin
     {scan leading blanks}
     while (ch='') and not endofline do
           getnextchar(linebuffer);

     {classify token based on its first char}
     if alphabetic(ch) then
       scanidentifier(linebuffer, token)
     else
     if numeric(ch) then
       scannumber(linebuffer, token}
     if ch=quote  then
       scanliteral(linebuffer, token)
     else
     if not endofline  then
       scanidelimiter(linebuffer, token)
     else
     if not endoffile then
         scanendofilne(linebuffer, token)
       else
       if endoffile then
         scanfileend(linebuffer, token)
       else
           half
  end;  {with}
  blankfill(token)  {follow token with blanks}
end;  {of procedure getnextsymbol}


{reporterrors


      write a list of errors that
      have been found in the file
}


procedure reporterrors(var linebuffer: linefubrec);
var
  err   : errorclass;  {loop index}
begin
  writenln('***** errors in file:');
  writeln;
  for err:= succ(errnone) to pred(errlast) do
    if err in linebuffer.errorset then
    begin
      write(ord(err)8, ':');
      case err of
        erroct     : write('digit 8 or 9 in
                            octal constant');
        errbigint  : write('integer constant>',
                            'maxint(=',maxint:1,')');
```

```
errexposize: write('abs(real exponent)>'
                   'maxexponent(=',
                    maxexponent:1,')');
errexpochar: write('digit expected in
                    exponent');
errnodigit : write('digit expected
                    after "." ');
errmissingquoter: write('no closing quote
                         in literal');
errlongliteral: write('literal too long
                       (max is',
                       tokenlenmax:1,' chars)')
      end;  {of case}
      writeln

    end;  {of error and for loop}
  writeln;
  writeln('end of error list')
end;  {of procedure reporterrors}



begin  {of program SCANNER}

  page(output);
  writeln(version);
  initialize(linebuffer);
  getnextline(linebuffer);  {read the first line}
  if not linebuffer.endoffile
       then
         token.class:= delimiter

  else token.class:= tendoffile;
  token.blankptr:= tokenlenmax;

  while token.class<>tendoffile do
  begin
    getnextsymbol(linebuffer, token);
    write('', token.tokenbuffer: 20, '->');
    with token do
      case class of
        identifier      : write('ident');
        integerconstant : write('integer=',
                                integervalue);
        realconstant    : write('real= ', realvalue);
        delimiter       : write('delimiter');
        literal         : write('literal');
        endofline       : write('end of line');
        tendoffile      : write('end of rile')
      end;  {of case and with}
    writeln;
  end;  {of while}
  writeln;
  if linebuffer.fileerror then reporterrors(linebuffer);
  writeln('execution of scanner complete');
end.   {program SCANNER}
```

B.1.2 Translated CSL Code

The CSL code of the lexical scanner is as follows:

```
proc scanner() {

comment: "Constants";
        debug := false;
        maxcharsperline := 140;
        maxexponent := 200;
        quote := "";
        tokenlenmax := 80;
        version := 'scanner0.4-a basic lexical scanner';

comment: "Enumeration type simulation";
        delimiter := 100;  identifier := 101; integerconstant := 102;
        literal := 103; realconstant :=104; tendoffile := 105;
        tendofline := 106;

        errnone := 200;  erroct := 201; errnodigit := 202; errbigint := 203;
        errexposize := 204;  errexpochar := 205; errmissingquote := 206;
        errlongliteral := 207; errlast := 208;

comment: "Record definition";

    struct tokenrec {
        int: blankptr;
        int: tbptr;
        char:  array tokenbuffer[tokenlenmax];
        int: class;
        int:  integervalue;
        real: realvalue;
      };

    struct linebufrec {
        char: ch;
        int: charptr;
        boolean:  echo;
        boolean:  endoffile;
        boolean:  endofline;
        int:  array errorline[lineindex];
        set:  errorset[1000];
        boolean:  fileerror;
        int: length;
        char: array line[lineindex];
        integer:  linecount;
        boolean:  lineerror;
        int:  pfrac;
        int:  pint;
        int:  pnum;
       };


comment: "Global variables";
    linebufrec: linebuffer;
    tokenrec: token;


comment: " initialize line buffer";

proc initialize(Out linebuffer: linebufrec) {

    int: i;

    if debug then !p writeln('initializing line buffer') fi;

    linebuffer.echo := true;
    linebuffer.lineerror := false;
    linebuffer.linecount := 0;

    for i := 0 to maxcharsperline do
```

```
        linebuffer.line[i] := '';
        linebuffer.errorline[i] := errnone
    od;

    linebuffer.errorset := [];
    linebuffer.fileerror := false;
    linebuffer.endoffile := false;
    linebuffer.endofline := true;
    linebuffer.pnum := 0

};

comment: " read a new line into the linebuffer ";

proc getnextline(Out linebuffer: linefufrec) {

  comment: " write a line and its line number to output ";

  proc printline(Out linebuffer: linebufrec) {

        i     : lineindex;

        !p write(linecount: 6,'');
        for i:= 1 to linebuffer.length do !p write(line[i]) od;
        !p writeln

    };


  comment: "  print pointers to errors, add to errorset, clear lineerror ";

proc printerrorline(var linebuffer: linebufrec) {

        int: column : integer;
        int: i      : integer;
        int: j      : integer;
        int: num    : integer;

        column:= 0;

        printline(Out linebuffer);
        !p write('*****': 6, '');

        for i:=1 to linebuffer.length +1 do
          if linebuffer.errorline[i]<>errnone then
                linebuffer.errorset:= linebuffer.errorset+[linebuffer.errorline[i]];
                num:= ord(linebuffer.errorline[i]);
                if i>column then
                  for j:= column + 2 to i do !p write('')  od;
                  !p write('|');
                  column:= i
                else
                  !p write(',');
                  column:= column + 1
                fi;
                !p write(num:1);
                column := column+1;
                if num>9 then column:= column+1 fi;
                linebuffer.errorline[i]:= errnone
         fi
       od;
       !p writeln;
       linebuffer.lineerror := false;
       linebuffer.fileerror:= true

};
```

```
comment: " begin of procedure getnextline ";

  if debug then !p writeln('getting new line') fi;

  if linebuffer.lineerror then printerrorline(Out linebuffer) fi;
  if not !p eof(input) then
        linebuffer.length:= 0;
        while not !p eoln(input) do
          linebuffer.length:= linebuffer.length+1;
          read(input, linebuffer.line[linebuffer.length])
        od;
        !p readln(input);

        comment: "delete any trailing blanks";
        linebuffer.line[0]:= '*';
        while linebuffer.line[length]='' do linebuffer.length := linebuffer.length-1  od;
        linebuffer.line[linebuffer.length] := '';

        linebuffer.linecount:= linebuffer.linecount+1;
        if linebuffer.echo then printline(Out linebuffer) fi;

        linebuffer.charptr := 1;
        linebuffer.ch := linebuffer.line[linebuffer.charptr];
        linebuffer.endofline := (linebuffer.charptr>linebuffer.length)
   else linebuffer.endoffile:= true
  fi
  };


comment:  "function to determine if a character is a letter";

  func alphabetic(In ch: char): boolean
   {
     alphabetic := (ch>='a') and (ch<='z')
   };

comment: " function to determine if a character is a digit";

  func numeric(In ch: char): boolean;
   {
    numeric:= (ch>='0') and (ch<='9')
   };

comment: " find next token in linebuffer";

 proc getnextsymbol(Out linebuffer: linebufrec; token: tokenrec) {

   comment: "place error message at the current buffer pointer";

   proc puterror(In error: errorclass;  Out linebuffer: linebufrec)
     {
        linebuffer.lineerror:= true;
        linebuffer.errorline[linebuffer.charptr]:= error
   };

comment: " ensure that the token buffer is blank filled";

proc blankfill(Out token: tokenrec) {

   while token.blankptr>token.tbptr do
     token.tokenbuffer[token.blankptr] := '';
     token.blankptr := token.blankptr-1
   od;
   token.blankptr:= token.tbptr
 };
```

```
comment: "read next character from line buffer and advance pointer";

  proc getnextchar(Out linebuffer: linebufrec) {

      if linebuffer.endofline then
        if !p eof(input) then linebuffer.endoffile:= true
        else getnextchar(Out linebuffer) fi
      else
        linebuffer.charptr:= linebuffer.charptr+1;
        if linebuffer.charptr>linebuffer.length
        then linebuffer.endofline:= true
        fi
      fi;
      linebuffer.ch := linebuffer.line[linebuffer.charptr]

   };

comment: " scan alphanumeric characters (copying them to tokenbuffer)";

  proc scanidentifier(Out linebuffer: linebufrec; token: tokenrec)
   {
        if debug then !p writeln('scanning identifier') fi;

        token.class := identifier;
        token.tbptr := 0;
        while (alphabetic(linebuffer.ch) or numeric(linebuffer.ch))
          if token.tbptr<tokenlenmax then
              token.tbptr:=token.tbptr+1;
              token.tokenbuffer[token.tpbtr]:= linebuffer.ch
          fi;
          getnextchar(Out linebuffer)
        od

   };

comment: "convert a decimal or octal number, or a real to internal form";

  proc scannumber(Out linebuffer: linebufrec; token: tokenrec)
   {
      integer: i;


  comment: " convert part of linebuffer to an integer (with no overflow)";

    proc convinteger(In base: integer; maxint: integer;
                         first, last:  lineindex;
                     Out linebuffer: linebufrec;  n: integer)
     {
        int: digit;
        integer: i;
        real:  x;

        n:= 0;
        x:= 0.0;
        i:= first;
        while i<last do
          digit:= ord(linebuffer.line[i])-ord('0');
          if digit >= base then
              puterror(In erroct, linebuffer);
              i:= last
          fi;

          x:= x*base+digit;
          if x<= maxint then n:=n*base+digit
          else
            puterror(In errbigint, linebuffer);
```

```
          i:= last
        fi;
        i:= i+1
      od
     };

comment: " scan a decimal integer";

  procedure scaninteger(Out linebuffer: linebufrec; token: tokenrec)
   {
      token.class:= integerconstant;
      convinteger(In 10, maxint, pint, charptr,
                  Out linebuffer, integervalue)
   };

comment: " scan an octal number";

  procedure scanoctal(Out linebuffer: linebuffrec; token: tokenrec)
   {
      token.class:= integerconstant;
      convinteger(In 8, maxint, pint, charptr,
                  Out linebuffer, integervalue);
      getnextchar(Out linebuffer)
   };

comment: " scan a real number with/without exponent";

  proc scanreal(Out linebuffer: linebufrec; token: tokenrec)
    {
      int: expo;
      real: fac;
      int:  i;
      boolean:  negexp;
      int:  nexpo;
      real: r;
      int: scale;
      real: x;

      if debug then !p writeln('scanning real number')  fi;
      token.class:= realconstant;

      comment:"do integer part, overflow assumed impossible";
      x:= 0.0;
      expo:= 0;
      for i:= linebuffer.pint to linebuffer.charptr-1 do
              x=x*10.0+ord(linebuffer.line[i]-ord('0')
      od;

      nexpo := linebuffer.charptr-linebuffer.pint;
      scale := 0;
      if linebuffer.ch ='.' then
           getnextchar(Out linebuffer);
           linebuffer.pfrac := linebuffer.charptr;
           if numeric(linebuffer.ch) then
             while (numeric(ch)) do
                 scale:=scale-1;
                 x:=x*10.0+ord(ch)-ord('0');
                 getnextchar(Out linebuffer)
             od
           else puterror(In errnodigit, linebuffer)
           fi;

      comment: "check if we must find first nonzero digit";
       if nexpo=0 then
         i:= linebuffer.pfrac;
         while linebuffer.line[i]= '0' do i:=i+1 od;
         nexpo := linebuffer.pfrac-i
```

```
    fi
fi; comment "fractional ch='.'"

comment: "do we have an exponent?";
if ch='e' then
  negexp := false;
  getnextchar(Out linebuffer);
  if ch='-' then
    negexp:=true;
    getnextchar(Out linebuffer)
  else if ch='+' then getnextchar(Out linebuffer) fi
  fi;

  comment "build exponent";
  if numeric(linebuffer.ch) then
     while numeric(ch) do
       expo:= expo*10+ ord(ch)-ord('0');
       getnextchar(linebuffer)
     od;

     comment: "adjust scale and nexpo";
     if negexp then
       scale:= scale-expo;
       nexpo:= scale-expo
     else
       scale:= scale+expo;
       nexpo:= scale+expo
     fi
 else puterror(In errexpochar, linebuffer)
 fi; comment "process numeric"

fi; comment "exponent"

     comment: "compute 10**scale using right to left binary method";
     if abs(nexpo)<=maxexponent then
       if scale<>0 then
         r:= 1.0;
         negexp:= scale<0;
         scale:= abs(scale);
         fac:= 10.0;
         while scale<>0 do
           if odd(scale) then r:=r*fac;
           fac:=sqr(fac);
           scale:= scale div 2
         od;
         if negexp then realvalue:= x/r
         else realvalue:= x*r
         fi
       else realvalue:=x fi
     else puterror(In errexposize, linebuffer)
    fi
};


comment: "begin of procedure scannumber";

  if debug then !p writeln('scanning number') fi;
  token.tbptr := 0;
  linebuffer.pint := linebuffer.charptr;

  comment: "scan integer part";
  while numeric(linebuffer.ch) do getnextchar(Out linebuffer) od;

  if linebuffer.ch<>'b' then
      if not((linebuffer.ch='.') or (linebuffer.ch='e')) then
              scaninteger(Out linebuffer, token)
```

```
    else scanreal(Out linebuffer, token)
    fi
else scanoctal(linebuffer, token) fi;

comment: " copy number into token buffer";
i:= pnum;
token.tbptr:= 0;
while(i<linebuffer.charptr) and (token.tbptr<tokenlenmax) do
    token.tbptr:= token.tbptr+1;
    tokenbuffer[token.tbptr]:= linebuffer.line[i];
    i:=i+1
  od;
  linebuffer.pnum:=0
};

comment: "read in a literal string";

 proc scanliteral(Out linebuffer: linebufrec; token: tokenrec)
   {
     boolean: working  : boolean;

     if debug then !p writenln('scanning literal')  fi;
     token.class:= literal;
     token.tbptr:= 0;
     getnextchar(linebuffer);
     working:= true;
     while working and not endofline do
       if linebuffer.ch=quote then
         getnextchar(Out linebuffer);
         working:= (linebuffer.ch= quote)
       fi;
       if working then
         if token.tbptr<tokenlenmax then
           token.tbptr:= token.tbptr+1;
           tokenbuffer[token.tbptr]:= linebuffer.ch;
           getnextchar(Out linebuffer)
         else
           puterror(In errlongliteral, linebuffer);
           while (linebuffer.ch<>quote) and not endofline do
               getnextchar(Out linebuffer)
           od;
           comment:"skip over string";
           if linebuffer.ch=quote then
               getnextchar(Out linebuffer)
           fi;
           working:= false
         fi
       fi
     od;
     if working then
       puterror(In errmissingquote,buffer)
     fi
};

  };

comment: "put ch into token buffer and advance";

  proc scandelimiter(Out linebuffer: linebufrec; token: tokenrec)
   {
    if debug then !p writeln('scanning delimiter')  fi;
    token.class:= delimiter;
    token.tbptr:= 1;
    token.tokenbuffer[token.tbptr]:= linebuffer.ch;
    getnextchar(Out linebuffer)
   };

comment: "return end of line status";
```

```
  proc scanendofline(Out linebuffer: linebufrec; token: tokenrec)
   {
    if debug then !p writeln('scanning end of line')  fi;
    token.class:= tendofline;
    token.tbptr:= 0
   };

comment: " return end of file status";

  proc scanfileend(Out linebuffer: linebufrec; token: tokenrec)
   {
    if debug then !p writeln('scanning end of file')  fi;
    token.class:= tendoffile;
    token.tbptr:= 0
   };


comment: "  begin of procedure getnextsymbol";
    if debug then !p writeln('getting next symbol. (ch=',linebuffer.ch,')') fi;
    if (token.class=tendofline) or
       (token.class= tendoffile) then
          getnextline(Out linebuffer)
    fi;

    comment: "scan leading blanks";
    while (linebuffer.ch='') and not linebuffer.endofline do
      getnextchar(Out linebuffer)
    od;
    comment: " classify token based on its first char";
    if alphabetic(linebuffer.ch) then
         scanidentifier(Out linebuffer, token)
    else
      if numeric(linebuffer.ch) then
         scannumber(Out linebuffer, token}
      else if linebuffer.ch=quote  then
              scanliteral(Out linebuffer, token)
           else
              if not linebuffer.endofline  then
                 scandelimiter(Out linebuffer, token)
              else
                if not linebuffer.endoffile then
                  scanendofilne(Out linebuffer, token)
                else
                  if linebuffer.endoffile then
                     scanfileend(Out linebuffer, token)
                  else
                     halt
                  fi
                fi
              fi
           fi
        fi
      fi;
     blankfill(Out token)
  };

comment: "write a list of errors that have been found in the file";

  proc reporterrors(Out linebuffer: linefubrec)
   {
     int : err;

     !p writenln('***** errors in file:');
     !p writeln;
     for err:= succ(errnone) to pred(errlast) do
       if err in linebuffer.errorset then
         write(ord(err)8, ':');
```

```
case err of
 erroct      : !p write('digit 8 or 9 in
                    octal constant');
 errbigint   : !p write('integer constant>',
                    'maxint(=',maxint:1,')');
 errexposize: !p write('abs(real exponent)>'
                     'maxexponent(=',
                      maxexponent:1,')');
 errexpochar: !p write('digit expected in
                      exponent');
 errnodigit : !p write('digit expected
                      after "." ');
          errmissingquoter: !p write('no closing quote
                                 in literal');
          errlongliteral: !p write('literal too long
                               (max is',
                               tokenlenmax:1,' chars)')
         end;
        !p writeln
       fi
      od;
    !p writeln;
    !p writeln
  };

comment: "begin of program SCANNER";

  !p page(output);
  !p writeln(version);
  initialize(Out linebuffer);
  getnextline(Out linebuffer);
  if not linebuffer.endoffile
  then  token.class:= delimiter
  else token.class:= tendoffile
  fi;
  token.blankptr:= tokenlenmax;

  while token.class<>tendoffile do
    getnextsymbol(Out linebuffer, token);
    !p write('', token.tokenbuffer: 20, '->');

      case token.class of
        identifier      : !p write('ident');
        integerconstant : !p write('integer=',
                               integervalue);
        realconstant    : !p write('real= ', realvalue);
        delimiter       : !p write('delimiter');
        literal         : !p write('literal');
        endofline       : !p write('end of line');
        tendoffile      : !p write('end of rile')
      end;
    !p writeln;
  od;
  !p writeln;
  if linebuffer.fileerror then reporterrors(Out linebuffer) fi;
  !p writeln('execution of scanner complete');

};
```

B.1.3 Extracted ITL Specification

In this subsection, a compelete specification abstracted from the source code of the lexical scanner is listed. The approach is the same as in the case studies of chapter 8. Abstraction rules are applied to each procedures, and for procedures which are considered as monolithic they are decomposed into resonable sections and each section is abstracted as a procedure.

Procedure: initialise

An new procedure *initline* is defined.

$$\mathbf{initline(line, errorline)} \triangleq \{i\} : i := 0;\ (line[i] :=' \ ' \wedge errorline[i] := erronone\ ;\ i := i + 1)^{maxcharperline}$$

$$\mathbf{initialise(linebuffer)} \succeq \{linebuffer\} :$$

$$linebuffer.echo := true \wedge linebuffer.linerror := false \wedge linebuffer.linecount := 0;$$

$$initline(linebuffer.line, linebuffer.errorline);$$

$$linebuffer.errorset := \phi \wedge linebuffer.fileerror := false \wedge linebuffer.endoffile := false \wedge$$

$$linebuffer.endofline := true \wedge linebuffer.pnum := 0$$

Procedure: printline

$$\mathbf{printline} \succeq \{i, linebuffer\} : print(linecount);\ i := 1;\ (print(line[i]);\ i := i + 1)^{linebuffer.length};\ println$$

Procedure: printerrorline

A new procedure *printerrmsg* is introduced.

$$\mathbf{printerrmsg} \succeq \{linebuffer, column, i, j, num\} : i := 1;$$

$$(linebuffer.errorline[i] <> errnone \wedge (linebuffer.errorset := linebuffer.errorset + [errorline[i]$$

$$(i > column \wedge (j = cloumn + 2;\ (print(' '))^{i-j-2});\ print('|');\ column := i) \vee$$

$$(i <= column \wedge print(',');\ column := column + 1);\ print(num);$$

$$column := column + 1;\ (num > 9 \wedge column := column + 1);$$

$$linebuffer.errorline[i] = errnone;\ i := i + 1)^{linebuffer.length}$$

$\mathbf{printerrorline} \succeq \{linebuffer\} : printline(linebuffer);\ printerrmsg(linebuffer.errorline)$

Procedure: getnextline

A new procedure *getnewline* is introduced.

$$\mathbf{getnewline} \succeq \{linebuffer\} : (\neg endof(input) \wedge (linebuffer.length := 0;$$
$$(linebuffer.length := linebuffer.length + 1;\ read(linebuffer.line[linebuffer]))^{lengthofline};$$
$$linebuffer.line[0] =' *';$$
$$(linebuffer.line[length] =' ' \wedge linebuffer.length := linebuffer.length - 1)^{lengthofline};$$
$$linebuffer.line[length + 1] =' ';\ echo = true \wedge printline(linebuffer);$$
$$linebuffer.charptr := 1;\ linebuffer.ch := linebuffer.line[linebuffer.charptr];$$
$$endofline := (linebuffer.charptr > linebuffer.length))$$
$$\vee(endof(input) \wedge linebuffer.endoffile := true)$$

$$\mathbf{getnextline} \succeq \{linebuffer\} : linebuffer.lineerror = true \wedge printerrorline(linebuffer);$$
$$getnewline(linebuffer)$$

Procedure: alphabetic

$$\mathbf{alphabetic} \succeq \{ch, alphabetic\} : alphabetic = (ch >=' a' \wedge ch <=' z')$$
$$\succeq \{ch, alphabetic\} : alphabetic = (ch = letter)$$

Procedure: numeric

$$\mathbf{numeric} \succeq \{ch, numeric\} : numeric = (ch >=' 0' \wedge ch <=' 9')$$
$$\succeq \{ch, numeric\} : numeric = (ch = number)$$

Procedure: puterror

$\mathbf{puterror} \succeq \{error, linebuffer\} : linebuffer.lineerror := true$

$\wedge linebuffer.errorline[linebuffer.charptr] := error$

Procedure: blankfill

$\mathbf{blankfill} \succeq \{token\} : (token.tokenbuffer[token.blankptr] :=' ';$

$token.blankptr := token.blankptr - 1)^{blankptr-tbptr}$

Procedure: getnextchar

$\mathbf{getnextchar} \succeq \{linebuffer\} : (linebuffer.endofline = true \wedge$

$((eof(input) \wedge linebuffer.endoffile := true) \vee (\neg eof(input) \wedge getnextline(linebuffer)))$

$\vee(linebuffer.endofline = false \wedge (linebuffer.charptr := linebuffer.charptr + 1;$

$(linebuffer.charptr > linebuffer.length) \wedge linebuffer.endofline = true)));$

$linebuffer.ch = linebuffer.line[linebuffer.charptr]$

Abstract away the state test and error handling details, we have a more concise specification:

$\mathbf{getnextchar} \succeq \{linebuffer\} : (linebuffer.endofline = true \wedge \neg eof(input) \wedge getnextline(linebuffer)$

$\vee(linebuffer.endofline = false \wedge (linebuffer.charptr := linebuffer.charptr + 1;$

$(linebuffer.charptr > linebuffer.length) \wedge linebuffer.endofline = true)));$

$linebuffer.ch = linebuffer.line[linebuffer.charptr]$

Most simply, the specification could be the following:

$\mathbf{getnextchar} \succeq \{linebuffer\} : linebuffer.charptr := linebuffer.charptr + 1;$

$linebuffer.ch = linebuffer.line[linebuffer.charptr]$

Procedure: scanidentifier

$$\textbf{scanidentifier} \succeq \{linebuffer, token\} : token.class := identifier;\ token.tbptr := 0;$$
$$((alphabetic(linebuffer.ch) \vee numeric(linebuffer.ch)) \wedge (token.tbptr < tokenlenmax \wedge$$
$$(token.tbptr := token.tbptr + 1;\ token.tokenbuffer[token.tbptr] = linebuffer.ch));$$
$$getnextchar(linebuffer))^*$$

The core of procedure *scanidentifier* is to keep read next character until it is no more a letter or number. Therefore, the final specification could be as follows:

$$\textbf{scanidentifier} \succeq \{linebuffer, token\} : ((alphabetic(linebuffer.ch) \vee numeric(linebuffer.ch)) \wedge$$
$$(token.tbptr := token.tbptr + 1;\ token.tokenbuffer[token.tbptr] = linebuffer.ch));$$
$$getnextchar(linebuffer))^*$$

Procedure: convertinteger

$$\textbf{convertinteger} \succeq \{base, maxint, first, last, linebuffer, n, digit, i, x\} :$$
$$n := 0;\ x := 0.0;\ i := first;\ (i < last \wedge (digit := ord(linebuffer.line[i]) - ord('0');$$
$$digit >= base \wedge (puterror(erroct, linebuffer);\ i = last);\ x := x \times base + digit;$$
$$(x <= maxint \wedge n := n \times base + digit) \vee (x > maxint) \wedge (puterror(errbigint, linebuffer);\ i = last);$$
$$i := i + 1))^*$$

Abstract away the error handling part, the specification will be:

$$\textbf{convertinteger} \succeq \{base, first, last, linebuffer\} :$$
$$((first < last) \wedge (n := n \times base + ord(linebuffer.line[first]) - ord('o'));\ first := first + 1))^*$$

Procedure: scaninteger

$$\textbf{scaninteger} \succeq \{linebuffer, token\} : token.class = integerconstant \wedge$$
$$\bigcirc convinteger(10, maxint, pint, cparptr, linebuffer, integervalue)$$

Procedure: scanoctal

$\text{scanoctal} \succeq \{linebuffer, token\} : token.class = integerconstant \wedge$

$\bigcirc convinteger(8, maxint, pint, charptr, linebuffer, integervalue)$

Procedure: scanreal

The following new procedures are introduced: *dealInteger*, *dealFraction*, *dealExponent*, *calcuReal*.

comment: " scan a real number with/without exponent";

$dealInteger(linebuffer, x, expo) \stackrel{\wedge}{=} \{i\} : x := 0.0 \wedge expo := 0 \wedge i := linebuffer.pint;$

$(x := x \times 10.0 + ord(linebuffer.line[i]) - ord('0') \wedge i := i + 1)^{linebuffer.charptr - linebuffer.pint - 1}$

$dealFraction(linebuffer, nexpo, scale, x) \stackrel{\wedge}{=} nexpo := linebuffer.charptr - linebuffer.pint \wedge scale := 0;$

$linebuffer.ch =' .' \wedge (getnextchar(linebuffer) \wedge linebuffer.pfrac := linebuffer.charptr;$

$(numeric(linebuffer.ch) = true \wedge$

$(numeric(linebuffer.ch) \wedge (scale := scale - 1 \wedge x := x \times 10.0 + ord(ch) - ord('0') \wedge$

$getnextchar(linebuffer)))^*;$

$nexpo = 0 \wedge (i := linebuffer.frac;\ (linebuffer.line[i] =' 0' \wedge i := i + 1)^*;$

$nexpo := linebuffer.pfrac - i));$

$dealExponent(linebuffer, nexpo, negexp, scale) \stackrel{\wedge}{=}$

$\{expo\} : linebuffer.ch =' e' \wedge (negexp := false \wedge getnextchar(linebuffer);$

$(linebuffer.ch =' -' \wedge (negexp := true \wedge getnextchar(linebuffer)))$

$\vee(linebuffer.ch =' +' \wedge getnextchar(linebuffer));$

$numeric(linebuffer.ch) \wedge (numeric(linebuffer.ch) \wedge (expo := expo \times 10 + ord(ch) - ord('0');$

$getnextchar(linebuffer)))^*;$

$(negexp \wedge (scale := scale - expo \wedge nexpo := scale - expo))$

$\vee(\neg negexp \wedge (scale := scale + expo \wedge nexpo := scale - expo)));$

$$calcuReal(token, scale, nexpo, negexp, x) \triangleq \{r, fac\} : abs(nexpo) <= maxexponent \wedge (scale <> 0 \wedge$$
$$(r := 1.0 \wedge negexp := scale < 0 \wedge scale := abs(scale) \wedge fac := 10.0;$$
$$(scale <> 0 \wedge (odd(scale) \wedge r := r * fac;\ fac := sqr(fac);\ scale := scale/2))^*;$$
$$(negexp \wedge realvalue := x/r) \vee (\neg negexp \wedge realvalue := x \times r))$$
$$\vee(scale = 0 \wedge realvalue = x)) \vee (abs(nexpo) > maxexponent \wedge puterror(errexposize, linebuffer))$$

$$scanreal(linebuffer, token) \triangleq \{negexp, nexpo, scale, x\} : token.class := realconstant;$$
$$dealInteger(linebuffer, x, expo)\ ;\ dealFraction(linebuffer, nexpo, scale, x);$$
$$dealExponent(linebuffer, nexpo, scale)\ ;\ calcuReal(token, scale, x)$$

Procedure: scanoctal

$$\textbf{scannumber} \succeq \{linebuffer, token, i\} : token.tbptr := 0;\ linebuffer.pint := linebuffer.charptr;$$
$$(numeric(linebuffer.ch) \wedge getnextchar(linebuffer))^*;$$
$$(linebuffer.ch <>' b' \wedge (\neg((linebuffer.ch =' .') \vee (linebuffer.ch =' e')) \wedge scaninteger(linebuffer, token)))$$
$$\vee(linebuffer.ch =' b' \wedge scanoctal(linebuffer, token));\ i := pnum;\ token.tbptr := 0;$$
$$((i < linebuffer.charptr) \wedge (token.tbptr < tokenlenmax) \wedge (token.tbptr := token.tbptr + 1;$$
$$tokenbuffer[token.tbptr] := linebuffer.line[i];\ i := i + 1))^*;\ linebuffer.pnum := 0$$

Take away the trivial details, the core specification is as follows:

$$\textbf{scannumber} \succeq \{linebuffer, token\} : (numeric(linebuffer.ch) \wedge getnextchar(linebuffer))^*;$$
$$(linebuffer.ch <>' b' \wedge linebuffer.ch <>' .' \wedge linebuffer.ch <>' e' \wedge scaninteger(linebuffer, token))\vee$$
$$(linebuffer.ch <>' b') \wedge (linebuffer.ch =' .' \vee linebuffer.ch =' e') \wedge scanreal(linebuffer, token))\vee$$
$$(linebuffer.ch =' b' \wedge scanoctal(linebuffer, token));\ tokenbuffer := linebuffer.line$$

Procedure: scanliteral

$$\textbf{scanliteral} \succeq \{linebuffer, token, working\} :$$
$$token.class := literal;\ token.tbptr := 0;\ getnextchar(linebuffer);$$

$(\neg linebuffer.endofline \wedge working \wedge (linebuffer.ch = quote \wedge$

$(getnextchar(linebuffer);\ working := (ch = quote));$

$(linebuffer.ch = quote \wedge (token.tbptr < token.tokenlenmax \wedge (token.tbptr := token.tbptr + 1;$

$token.tokenbuffer[token.tbptr] := linebuffer.ch;\ getnextchar(linebuffer))) \vee$

$(token.tbptr > token.tokenlenmax \wedge (puterror(errlongliteral, linebuffer);$

$(linebuffer.ch <> quote \wedge \neg linebuffer.endofline \wedge (getnextchar(linebuffer)))^*;$

$linebuffer.ch = quote \wedge working := false))))^*;\ working \wedge puterror(errmissingquote, buffer)$

Delete all state test and error handling details:

$\mathbf{scanliteral} \succeq \{linebuffer, token, working\}$:

$token.class := literal;\ token.tbptr := 0;\ getnextchar(linebuffer);$

$(\neg linebuffer.endofline \wedge working \wedge (linebuffer.ch = quote \wedge$

$(getnextchar(linebuffer);\ working := (ch = quote));$

$(linebuffer.ch = quote \wedge (token.tbptr < token.tokenlenmax \wedge (token.tbptr := token.tbptr + 1;$

$token.tokenbuffer[token.tbptr] := linebuffer.ch;\ getnextchar(linebuffer)))))^*$

Eliminating remaining *working* by rewriting its effect in other way:

$\mathbf{scanliteral} \succeq \{linebuffer, token\}$:

$token.class := literal;\ token.tbptr := 0;\ getnextchar(linebuffer);$

$(\neg linebuffer.endofline \wedge$

$((linebuffer.ch = quote \wedge \bigcirc getnextchar(linebuffer) \wedge \bigcirc linebuffer.ch = quote) \vee$

$(linebuffer.ch \neq quote)) \wedge$

$(token.tbptr := token.tbptr + 1;\ token.tokenbuffer[token.tbptr] := linebuffer.ch;\ getnextchar(linebuffer)))^*$

Procedure: scandelimiter

$\mathbf{scandelimiter} \succeq \{linebuffer, token\} : token.class = deliminter \wedge token.tbptr = 1 \wedge$

$token.tokenbuffer[token.tbptr] = linebuffer.ch \wedge \bigcirc\ getnextchar(linebuffer)$

Procedure: scanendofline

$\mathbf{scanendofline} \succeq \{linebuffer, token\} : token.class := tendofline;\ token.tbptr := 0$

Procedure: scanfileend

$\mathbf{scanfileend} \succeq \{linebuffer, token\} : token.class := tendoffile;\ token.tbptr := 0$

Procedure: getnextsymbol

$\mathbf{getnextsymbol} \succeq \{linebuffer, token\} :$

$(token.class = tendofline \vee token.calss = tendoffile) \wedge getnextline(linebuffer);$

$(linebuffer.ch =' ' \wedge \neg linebuffer.endofline \wedge getnextchar(linebuffer))^*;$

$(alphabetic(linebuffer.ch) \wedge scanidentifier(linebuffer, token))\vee$

$(numeric(linebuffer.ch) \wedge scannumber(linebuffer, token))\vee$

$(linebuffer.ch = quote \wedge scanliteral(linebuffer, token))\vee$

$(\neg linebuffer.endofline \wedge scandelimiter(linebuffer, token))\vee$

$(\neg linebuffer.endoffile \wedge scanendoffile(linebuffer, token))\vee$

$(linebuffer.endoffile \wedge scanfileend(linebuffer, token))$

Procedure: reporterror

$\mathbf{reporterror} \succeq \{linebuffer, err\} : err = succ(errnone);$

$(err \in linebuffer.errorset \wedge (print(ord(err));$

$(err = erroct \wedge print('digit\ 8\ or\ 9\ in\ octal\ constant'))\vee$

$(err = errbigint \wedge print('integer\ constant >',' maxint(=', maxint : 1,')'))\vee$

$(err = errexposize \wedge print('abs(real\ exponent) > maxexponent(='))\vee$

$(err = errexpochar \wedge print('digit\ expected\ in\ )exponent'))\vee$

$(err = errnodigit \wedge print('digit\ expected\ after\ .\ '))\vee$

$(err = errmissingquoter \wedge print('no\ closing\ quote\ in\ literal'))\vee$

$(err = errlongliteral \wedge (print('literal\ too\ long);\ print(max\ is',\ tokenlenmax : 1,'\ chars)'));$

$err = err + 1)^{pred(errlast)-succ(errnone)};$

Procedure: scanner

Finally, let us have a look at the main entrance procedure.

scanner$\succeq \{linebuffer, token\}$:

$page(output);\ println(version);\ initialize(linebuffer);\ getnextline(linebuffer);$

$(\neg linebuffer.endoffile \wedge token.class = delimiter) \vee (linebuffer.endoffile \wedge token.class = endoffile);$

$token.blankptr = tokenlenmax;$

$(token.class <> tendoffile\wedge$

$(getnextsymbol(linebuffer, token);$

$(token.class = identifier \wedge print('indent'))\vee$

$(token.class = integerconstant \wedge print('integer =', integervalue))\vee$

$(token.class = realconstant \wedge print('real =', realvalue))\vee$

$(token.class = delimiter \wedge print('delimiter'))\vee$

$(literal \wedge print('literal'))\vee$

$(token.class = endofline \wedge print('endofline'))\vee$

$(token.class = tendoffile \wedge print('endofrile')))^*;$

$linebuffer.fileerror \wedge reporterrors(linebuffer);$

$print('executionofscannercomplete')$

B.2 Mine Drainage System

B.2.1 CSL Code Translated from Ada

Methane Model

```
proc init()
  {
    comment:"enable device";
    ch4_sensor_status:= enabled;
    ch4_status:=motor_unsafe
  };

proce ch4_process()
  {
    read tm, ch4_level from ch4_sensor;
    if ch4_level>=ch4_Max
    then if ch4_status=motor_safe
         then motor_unsafe();
              operator_console_alarm(In "High-methane" Out);
              ch4_status:=motor_unsafe
         fi
    else if (ch4_level<ch4_Max-jitterrange)
         then motor_safe();
         ch4_status:=motor_safe
         fi
    fi;
    ch4_log(In ch4_level Out)
  };


proc ch4_period()
  {
    init();
    while true do
      duration in 30 ch4_process() end;
      delay (80-30)
    od
  };
```

Monodioxide Module

```
procedure init()
  {
    comment:"enable device";
    co_sensor_status:= enabled
  };

proc co_process()
  {
    init();
    while true do
      duration in 60
        read tm, co_level from co_sensor;
        if co_level>=co_max
        then operator_console_alarm(In "High-co" Out)
        fi;
        co_log(In co_level Out)
      end;
      delay 40
    od
  };
```

Pump Module

```
proce motor_unsafe()
  {
```

```
    if motor_status=On
    then
         motor_log(In 100 Out)
    fi;
    motor_condition:=disabled;
    motor_log(In "motor-unsafe" Out)
  };

proce motor_safe()
  {
    if motor_status=off
    then sw:=On;
         motor_status:=On;
         motor_log(In "motor-started" Out)
    fi;
    motor_condition:=enabled;
    motor_log(In "motor-safe" Out)
  };

proc set_pump(In pump_status: Boolean; Out)
  {
    if pump_status=On
    then if motor_status=off
         then if motor_condition=disabled
              then err_msg(In "pump-not-safe" Out)
              fi;
              if ch4_status=motor_safe
              then motor_status :=On;
                   sw:=On;
                   motor_log(In "motor-started" Out)
              else err_msg(In "pump-not-safe" Out)
              fi
         fi
    else if motor_status=On
         then motor_status:=off;
              if motor_condition=enabled
              then
                   sw:=off;
                   motor_log(In "motor-stopped" Out)
              fi
         fi
    fi
 };
```

Water Flow Module

```
proc init()
  {
    comment:"enable device";
    water_flow_sensor_status:= enabled;
    water_flow_signal:=off;
    current_pump_status:=off;
    last_pump_status:=off
  };

procedure water_flow_process()
  {
    current_pump_status:=motor_status;
    current_pump_condition:=motor_condition;
    read tm, water_flow_signal from water_flow_sensor;
    if (current_pump_status=On) and (last_pump_status=On) and
       (water_flow_signal=off)
    then operator_console_alarm(In "pump-fault" Out)
    else if (current_pump_status=off) and (last_pump_status=off) and
         (water_flow_signal=On)
```

```
          then operator_console_alarm(In "pump-fault" Out)
          fi
    fi;
    last_pump_status:=current_pump_status;
    water_flow_log(In water_flow_signal Out)
  };

procedure water_flow_period()
  {
    init();
    while true do
      duration in 40 water_flow_process() end;
      delay (100-40)
    od
  };
```

Water Level Module

```
proc init()
  {
    comment:"enable device";
    water_level_sensor_status:= enabled;
    HW_interrupt:=enabled;
    LW_interrupt:=enabled;
  };

proc water_level_signal_process(In w_signal:integer Out)
  {
    if (w_signal=High_alarm) and (HW_interrupt=enabled)
    then set_pump(In On Out);
         high_low_water_log(In High_alarm Out);
         LW_interrupt:=enabled;
         HW_interrupt:=disabled
    else if (w_signal=Low_alarm) and (LW_interrupt=enabled)
         then set_pump(In off Out);
              high_low_water_log(In Low_alarm Out);
              LW_interrupt:=disabled;
              HW_interrupt:=enabled
         fi
    fi
  };

procedure water_level_monitoring()
  {
    init();
    while true do
      wait on water_level_sensor for 5 do
        delay 0
      else
        duration in 35 read tm, water_level_signal from water_level_sensor end;
        duration in 160 water_level_signal_process(In water_level_signal Out) end
      end
    od
  };
```

Air Flow Module

```
proc init()
  {
    comment:"enable device";
    air_flow_sensor_status:= enabled
  };
```

```
procedure air_flow_process()
  {
    init();
    while true do
      duration in 100
        read tm, air_flow_signal from air_flow_sensor;
        if air_flow_signal=off
        then operator_console_alarm(In "No-air-flow" Out);
             air_flow_log(In air_flow_signal Out)
        else air_flow_log(In air_flow_signal Out)
        fi
      end
    od
  };
```

Main Procedure

```
proc main()
 {
   Boolean: On, off, disabled, enabled, motor_safe, motor_unsafe;
   integer: High_alarm, Low_alarm, ch4_max, co_max;

   Boolean: motor_status, motor_condition;
   Boolean: sw, pump_status, water_flow_signal;
   Boolean: current_pump_status, current_pump_condition, last_pump_status;
   Shunt: water_flow_sensor, water_level_sensor, ch4_sensor, air_flow_sensor, co_sensor;
   Boolean: water_flow_sensor_status;
   integer: water_level_signal;
   Boolean: ch4_status, ch_high_signal;
   Boolean: HW_interrupt, LW_interrupt;
   integer: ch4_level;
   Boolean: air_flow_sensor_status, co_sensor_status;
   Boolean: air_flow_signal;
   integer: co_level;

   On:=1; enabled:=1; motor_safe:=1;
   off:=0; disabled:=0; motor_unsafe:=0;
   High_alarm:=3; Low_alarm:=2;
   ch4_max:=400; co_max:=800;

   parbegin
       water_flow_period()
     parallel with
       parbegin
           water_level_detect()
         parallel with
           parbegin
              ch4_period()
            parallel with
              parbegin
                co_period()
               parallel with
                air_flow_period()
              parend
           parend
       parend
     parend
   };
```

B.2.2 Extracted ITL Specification

In this subsection, a compelete specification abstracted from the source code of the mine drainage system is listed. The approach is the same as in the case studies of chapter 8. Abstraction rules are applied to each procedures. The methane module and the pump module has been processed in chapter 8.

Methane Model

$$init() \triangleq ch4\text{-}sensor\text{-}status := Enabled \wedge ch4\text{-}status := Motor\text{-}unsafe$$

$$\begin{aligned} &ch4\text{-}process() \triangleq ch4\text{-}level = read(ch4\text{-}sensor); \\ &\quad (ch4\text{-}level \geq ch4\text{-}Max) \wedge ch4\text{-}status = motor\text{-}safe \wedge \\ &\qquad (motor\text{-}unsafe() \wedge operator\text{-}console\text{-}alarm('High\text{-}methane') \wedge ch4\text{-}status := motor\text{-}unsafe) \\ &\quad \vee (ch4\text{-}level < ch4\text{-}Max) \wedge (ch4\text{-}level < ch4\text{-}Max - jitterrange) \wedge (motor\text{-}safe() \wedge \\ &\quad ch4\text{-}status := motor\text{-}safe) \; ; \; ch4\text{-}log(ch4\text{-}level) \end{aligned}$$

$$ch4\text{-}period() \triangleq init() \; ; \; (ch4\text{-}process() \wedge len \leq 30ms \; ; \; len = 30ms)^*$$

Pump Module

$$motor\text{-}unsafe() \triangleq motor\text{-}status = On \wedge (sw := Off \; ; \; motor\text{-}status := Off) \; ; \; motor\text{-}condition := Disabled$$

$$motor\text{-}safe() \triangleq motor\text{-}status = Off \wedge (sw := On \; ; \; motor\text{-}status := On) \; ; \; motor\text{-}condition := Enabled$$

$$\begin{aligned} &set\text{-}pump(pump\text{-}status) \triangleq \\ &\quad (pump\text{-}status = On \wedge motor\text{-}status = Off \wedge ch4\text{-}status = Motor\text{-}safe \wedge \\ &\qquad (motor\text{-}status := On \; ; \; sw := On)) \\ &\quad \vee (pump\text{-}status = Off \wedge motor\text{-}status = On \wedge \\ &\qquad (motor\text{-}status := Off \; ; \; motor\text{-}condition = Enabled \wedge sw := Off)) \end{aligned}$$

Monodioxide Module

The initial specification is as follows after elementary abstraction rules are applied:

$$init() \triangleq co\text{-}sensor\text{-}status := enabled$$

$$co\text{-}process() \triangleq init() \; ; \; ((tm = \surd co\text{-}sensor \wedge co\text{-}level = read(co\text{-}sensor);$$

$co\text{-}level \geq co\text{-}Max \wedge operator\text{-}console\text{-}alarm('High\text{-}co');$
$co\text{-}log(co\text{-}level)) \wedge len \leq 60\ ;\ len = 40ms)^*$

Replace possible "chop" operators with logic conjunctions, leave out the unused timestamp *tm*, and abstract away the log details, the final specification is as follows:

$init() \triangleq co\text{-}sensor\text{-}status := enabled$

$co\text{-}process() \triangleq init()\ ;\ ((co\text{-}level = read(co\text{-}sensor);$
$co\text{-}level \geq co\text{-}Max \wedge operator\text{-}console\text{-}alarm('High\text{-}co'))$
$\wedge len \leq 60\ ;\ len = 40ms)^*$

Water Flow Module

The initial specification is as follows after elementary abstraction rules are applied:

$init() \triangleq water\text{-}flow\text{-}sensor\text{-}status := Enabled\ ;\ water\text{-}flow\text{-}signal := off;$
$current\text{-}pump\text{-}status := off\ ;\ last\text{-}pump\text{-}status := off$

$water\text{-}flow\text{-}process() \triangleq current\text{-}pump\text{-}status := motor\text{-}status\ ;\ current\text{-}pump\text{-}condition := motor\text{-}condition;$
$tm = \surd water\text{-}flow\text{-}sensor \wedge water\text{-}flow\text{-}level = read(water\text{-}flow\text{-}sensor);$
$((current\text{-}pump\text{-}status = on) \wedge (last\text{-}pump\text{-}status = on) \wedge (water\text{-}flow\text{-}signal = off) \wedge$
$operator\text{-}console\text{-}alarm('Pump\text{-}fault'))$
$\vee((current\text{-}pump\text{-}status = off) \wedge (last\text{-}pump\text{-}status = off) \wedge (water\text{-}flow\text{-}signal = on)) \wedge$
$operator\text{-}console\text{-}alarm('Pump\text{-}fault'));$
$last\text{-}pump\text{-}status = current\text{-}pump\text{-}status\ ;\ water\text{-}flow\text{-}log(water\text{-}flow\text{-}signal)$

$water\text{-}flow\text{-}period() \triangleq init()\ ;\ (water\text{-}flow\text{-}process() \wedge len \leq 40ms\ ;\ len = 60ms)^*$

Replace possible "chop" operators with logic conjunctions, leave out the unused timestamp *tm*, and abstract away the log details, the final specification is as follows:

$init() \triangleq water\text{-}flow\text{-}sensor\text{-}status := Enabled \wedge water\text{-}flow\text{-}signal := off \wedge$
$current\text{-}pump\text{-}status := off \wedge last\text{-}pump\text{-}status := off$

$water\text{-}flow\text{-}process() \triangleq current\text{-}pump\text{-}status := motor\text{-}status\ ;\ current\text{-}pump\text{-}condition := motor\text{-}condition;$
$water\text{-}flow\text{-}level = read(water\text{-}flow\text{-}sensor);$
$(((current\text{-}pump\text{-}status = on) \wedge (last\text{-}pump\text{-}status = on) \wedge (water\text{-}flow\text{-}signal = off))$
$\vee((current\text{-}pump\text{-}status = off) \wedge (last\text{-}pump\text{-}status = off) \wedge (water\text{-}flow\text{-}signal = on))\wedge$
$operator\text{-}console\text{-}alarm('Pump\text{-}fault'));$
$last\text{-}pump\text{-}status = current\text{-}pump\text{-}status$

$water\text{-}flow\text{-}period() \triangleq init()\ ;\ (water\text{-}flow\text{-}process() \wedge len \leq 40ms\ ;\ len = 60ms)^*$

Water Level Module

The initial specification is as follows after elementary abstraction rules are applied:

$init() \triangleq water\text{-}level\text{-}sensor\text{-}status := Enabled;$
$HW\text{-}interrupt := enabled\ ;\ LW\text{-}interrupt := enabled$

$water\text{-}level\text{-}signal\text{-}process(w\text{-}signal) \triangleq$
$(w\text{-}signal = High\text{-}alarm \wedge HW\text{-}interrupt = enabled\wedge$
$set\text{-}pump(on)\ ;\ high\text{-}low\text{-}water\text{-}log(High\text{-}alarm);$
$LW\text{-}interrupt := enabled\ ;\ HW\text{-}interrupt := disabled)$
$\vee((w\text{-}signal = Low\text{-}alarm \wedge LW\text{-}interrupt = enabled\wedge$
$set\text{-}pump(off)\ ;\ high\text{-}low\text{-}water\text{-}log(Low\text{-}alarm);$
$LW\text{-}interrupt := disabled\ ;\ HW\text{-}interrupt := enabled)$

$water\text{-}level\text{-}monitoring() \triangleq init();$
$(true \wedge ((\Delta 5 \wedge \mathsf{stable}\ (\surd water\text{-}level\text{-}sensor)\ ;\ len = 0)\vee$
$(\Delta 5 \wedge \neg\, \mathsf{stable}\ (\surd water\text{-}level\text{-}sensor);$
$((tm = \surd water\text{-}level\text{-}sensor \wedge water\text{-}level\text{-}signal = \mathsf{read}\ (water\text{-}level\text{-}sensor)) \wedge len \leq 35;$
$water\text{-}level\text{-}signal\text{-}process(water\text{-}level\text{-}signal) \wedge len \leq 160))))^*$

Replace possible "chop" operators with logic conjunctions, leave out the unused timestamp *tm*, and abstract away the log details, the final specification is as follows:

$init() \triangleq water\text{-}level\text{-}sensor\text{-}status := Enabled\wedge$
$HW\text{-}interrupt := enabled \wedge LW\text{-}interrupt := enabled$

$$water\text{-}level\text{-}signal\text{-}process(w\text{-}signal) \triangleq$$
$$(w\text{-}signal = High\text{-}alarm \wedge HW\text{-}interrupt = enabled \wedge$$
$$set\text{-}pump(on)\ ;\ LW\text{-}interrupt := enabled\ ;\ HW\text{-}interrupt := disabled)$$
$$\vee((w\text{-}signal = Low\text{-}alarm \wedge LW\text{-}interrupt = enabled \wedge$$
$$set\text{-}pump(off)\ ;\ LW\text{-}interrupt := disabled\ ;\ HW\text{-}interrupt := enabled)$$

$$water\text{-}level\text{-}monitoring() \triangleq init();$$
$$((\Delta 5 \wedge \mathsf{stable}\,(\surd water\text{-}level\text{-}sensor)\ ;\ len = 0) \vee$$
$$(\Delta 5 \wedge \neg\,\mathsf{stable}\,(\surd water\text{-}level\text{-}sensor);$$
$$(water\text{-}level\text{-}signal = \mathsf{read}\,(water\text{-}level\text{-}sensor) \wedge len \leq 35;$$
$$water\text{-}level\text{-}signal\text{-}process(water\text{-}level\text{-}signal) \wedge len \leq 160)))^*$$

Air Flow Module

The initial specification is as follows after elementary abstraction rules are applied:

$$init() \triangleq air\text{-}flow\text{-}sensor\text{-}status := enabled$$

$$air\text{-}flow\text{-}process() \triangleq init()\ ;\ ((tm = \surd air\text{-}flow\text{-}sensor \wedge air\text{-}flow\text{-}level = read(air\text{-}flow\text{-}sensor);$$
$$air\text{-}flow\text{-}signal = off \wedge operator\text{-}console\text{-}alarm('No\text{-}air\text{-}flow');$$
$$air\text{-}flow\text{-}log(air\text{-}flow\text{-}signal)$$
$$\vee air\text{-}flow\text{-}log(air\text{-}flow\text{-}signal)) \wedge len \leq 100)^*$$

Replace possible "chop" operators with logic conjunctions, leave out the unused timestamp *tm*, and abstract away the log details, the final specification is as follows:

$$init() \triangleq air\text{-}flow\text{-}sensor\text{-}status := enabled$$

$$air\text{-}flow\text{-}process() \triangleq init()\ ;\ ((air\text{-}flow\text{-}level = read(air\text{-}flow\text{-}sensor);$$
$$air\text{-}flow\text{-}signal = off \wedge operator\text{-}console\text{-}alarm('No\text{-}air\text{-}flow'))) \wedge len \leq 100)^*$$

Main Procedure

After replace the "chop" operators with logic conjunction, the final specification is as follows:

$$\begin{aligned} main() \triangleq\ & On := 1 \wedge enabled := 1 \wedge motor\text{-}safe := 1 \wedge off := 0 \wedge disabled := 0 \wedge motor\text{-}unsafe := 0 \\ & \wedge High\text{-}alarm := 3 \wedge Low\text{-}alarm := 2 \wedge ch4\text{-}Max := 400 \wedge co\text{-}Max := 800; \\ & water\text{-}flow\text{-}period() \wedge water\text{-}level\text{-}detect() \wedge ch4\text{-}period() \wedge co\text{-}period() \wedge air\text{-}flow\text{-}period() \end{aligned}$$