Formal Specification and Verification of Knowledge and its Application

Ralph Miarka
miarka@aix520.informatik.uni-leipzig.de

Leipzig University
Faculty for Mathematics and Computer Science
Institute for Computer Science

Master's Thesis

supervised by

Prof. Dr. Heinrich Herre, Leipzig University

and

Dr. Peter Hrandek, Siemens AG Austria

Leipzig, July 1998

Formale Wissensspezifikation und -verifikation und deren Anwendung

Ralph Miarka

miarka@aix520.informatik.uni-leipzig.de

Universität Leipzig

Fakultät für Mathematik und Informatik

Institut für Informatik

DIPLOMARBEIT

betreut von

Prof. Dr. Heinrich Herre, Universität Leipzig

und

Dr. Peter Hrandek, Siemens AG Österreich

Leipzig, Juli 1998

Abstract

Software must become more reliable. It is widely accepted that, if correctly applied, formal methods can improve the reliability of programs. Unfortunately, formal methods have yet little impact on the practice. The aim of this work is to investigate how methods, tools, and results of knowledge engineering can be applied in specification engineering.

We investigate the relationship between knowledge engineering and formal specification. We compare aims, concepts, and problems of both disciplines of computer science. In particular, we examine a theory of verification originally developed for the verification of knowledge bases. Further, we look at a specification tool designed for the development of knowledge based systems, which in turn is a knowledge representation system itself. Finally, we put this thesis into the context of the Standard Siemens Development Methodology.

The results of this thesis are manyfold. First, we found that knowledge engineering and formal specification have much in common. Basically, both meet at the point of representing facts and relationships in an abstract manner. Our comparison leads to the view that many achievements of knowledge engineering should be used in formal specification.

Second, we extended the work on a theory of verification of knowledge bases by considering universal theories in general. Such extension allows to consider formal specifications as well. Further, we extended the query language. In that way, more complex sentences can be proved. The theory we adopted considers a world description to be given explicitly. In that way the theory serves the purpose of an early error detection, as suggested by [Jackson and Wing, 1996].

Third, the specification and verification example showed the practical applicability of a knowledge engineering tool in formal specification. Finally, the discussion on the use of formal methods within the Standard Siemens Development Methodology related formal methods to the industry. We found that it is not too difficult to use formal methods within an existing development methodology. We also emphasised on the ability to outsource the process of creating a formal specification and the verification and validation process.

The right use of formal methods can reduce software development costs. The right use of results made in knowledge engineering can reduce development costs within specification engineering. This thesis provides some ideas as well as solutions to this problem.

Zusammenfassung

Software muß zuverlässiger werden. Es ist allgemein anerkannt, daß formale Methoden, sofern sie richtig angewandt werden, die Sicherheit und Stabilität von Software verbessern können. Jedoch werden formale Methoden noch selten für industrielle Projekte genutzt. Das Ziel dieser Arbeit ist die Untersuchung, ob und wie Methoden, Werkzeuge und Ergebnisse aus dem Bereich der Wissensrepräsentation auf das Gebiet der formalen Spezifikation übertragen werden können.

Zuerst vergleichen wir Ziele, Konzepte und Probleme der Wissenrepräsentation mit denen der formalen Spezifikation. Danach untersuchen wir einige Eigenschaften von Spezifikationen und Wissensbasen, wie zum Beispiel Konsistenz, Vollständigkeit und Korrektheit. Weiterhin betrachten wir ein Werkzeug zur Spezifikation von wissensbasierten Systemen, welches selbst ein Wissensrepräsentationssystem ist. Abschließend ordnen wir diese Arbeit in die Standard Siemens Entwicklungsmethode (stdSEM) ein.

Die Ergebnisse dieser Arbeit sind vielfältig. Erstens, Wissensrepräsentation und formale Spezifikation haben viele Gemeinsamkeiten. Die wohl wichtigste ist, daß beide Methoden sich mit der abstrakten Darstellung von Fakten und Beziehungen beschäftigen. Unser Vergleich führt zu der Ansicht, daß Ergebnisse der Wissensrepräsentation auch im Bereich der Spezifikation genutzt werden können.

Zweitens, im Rahmen der Untersuchung von Eigenschaften von Spezifikationen ist es uns gelungen, eine Arbeit zur Verifikation von Wissensbasen so zu erweitern, daß sie auch für die Verifikation von Spezifikationen genutzt werden kann. Dabei stehen universale Theorien im Vordergrund unserer Betrachtungen. Die entwickelte Theorie stützt sich auf eine explizit gegebene Weltbeschreibung. Dadurch wird eher eine frühzeitige Fehlererkennung ermöglicht, wie sie von [Jackson and Wing, 1996] gefordert wird.

Drittens konnten wir die Nutzbarkeit eines Werkzeuges der Wissensrepräsentation für die Spezifikation anhand eines Beispieles belegen. Schließlich zeigten wir, daß formale Methoden in eine bestehende Entwicklungmethode eingebunden werden können. Darüber hinaus belegten wir die Möglichkeit, den Prozeß der formalen Spezifikation und Verifikation auszulagern und an andere Firmen zu übertragen.

Die richtige Nutzung formaler Methoden kann Kosten in der Softwareentwicklung senken. Die richtige Nutzung der Ergebnisse der Wissensrepräsentation kann Kosten im Bereich der Entwicklung formaler Methoden reduzieren. Die vorliegende Arbeit bietet Ideen und Lösungen für dieses Problem.

Contents

List of Figures

List of Tables

Preface

> Proficiency in any art or science is not attained until its history is known. Often a student and a designer finds, after weary hours of thought, that the problems over which he studied were considered and mastered by others, years or centuries before, perhaps with better results than his own. [Tyrell, 1911]

My motivation for this thesis is best expressed by the above quotation. It was in 1994/95 that I was introduced to formal specification methods. In my opinion, such methods form the future of software development. Unfortunately, formal methods are not a major topic of education at the University of Leipzig. However, knowledge representation is heavily investigated by the working group 'Formal Concepts' lead by Prof. Heinrich Herre. While attending several lectures on knowledge representation and formal reasoning, I realised that most of the presented problems were similar to those of formal specification and verification of software. Therefore, I decided to investigate this relationship in more detail.

Since February 1996 I was working several times for the Siemens AG Austria as a member of the Department 'Programming Environments, Databases, and Scientific Systems' lead by Dr. Peter Hrandek. We both agree on the point that new software development methods are needed. In particular formal verification of software forms a major part of interest, since software reliability must constantly be improved. Therefore, the work on this thesis was done under supervision of both Dr. Peter Hrandek and Prof. Heinrich Herre.

Acknowledgements

I am thankful to Prof. Heinrich Herre and to Dr. Peter Hrandek for supervising my work. Both gave me a complete free hand to choose the path this thesis should take. Further, I thank Jan Treur and Lourens van der Meij from the Vrije Universiteit Amsterdam for their support when dealing with DESIRE.

I also thank Prof. John Derrick from the University of Kent at Canterbury, for the chance to take part in a project on formal methods, and Prof. Andrew Frank for

the possibility to attend the GeoPo'97 - an internal conference by the Department of Geoinformation Systems, TU Vienna. In particular his hints on how to write a thesis were very valuable.

I thank Steffen and Thomas Bittner for the fruitful discussions on some parts of this thesis. Further, I am thankful to Katja Brunsch, Dr. Sabine Timpf, and Hans-Jürgen Kiesch for reading preliminary versions of this thesis and for improving the English of this work.

Finally, I would like to thank my friends and, of course, my parents for their continued support. Thank you all.

Ralph Miarka
Leipzig, July 1998

Chapter 1

Introduction

Software errors can be very expensive. Sometimes a company damages their image, sometimes they lose much money but at other times, humans might lose their lives. In July 1996, a software error was responsible for the loss of the European Space Agency's Ariane 5, carrying satellites worth 500 million dollars [Easterbrook, 1996; Welt, 1996]. It was earlier this year that Bill Gates demonstrated Windows 98, Microsoft's new operating system, when it malfunctioned [Welt, 1998]. Although it seems that this was not as serious as the Ariane accident, one has to consider that Windows 98 will run on about 90 percent of all personal computers world wide.

It was against this background that Jürgen Kuri wrote in the German computer magazine c't [ct, 1998, p. 3]:

> Normalität kehrt ein in eine Branche, die sich lange etwas auf ihre Sonderrolle zugute hielt. [...] Software und Hardware sind nichts Besonderes: Etwas Besonderes sind ihre Probleme und Fehler.
>
> [Aber] wenn sie keinen Sonderstatus mehr hat, [dann] muß sie sich auch an den Standards messen lassen, die für andere Bereiche gelten. Die Ausrede, keine Software könne fehlerfrei sein, zieht nicht mehr. Das gilt schließlich auch für jedes komplexe technische System. Mehr noch: In all diesen technischen Systemen steckt inzwischen Software, die fehlerfrei zu funktionieren hat.
>
> Windows CE in Waschmaschinen und Autos ist schließlich kein Spaß mehr. Weder Hausfrau noch Hausmann akzeptieren, wenn der Vollwaschgang abstürzt. Die Anwender von Technik, und davon ist die EDV inzwischen nur noch eine Facette, lassen sich nicht mehr mit dummen Sprüchen abspeisen.

In other words, software became part of our lives, like washing machines and cars. Therefore, software has to meet the same standards as other technical devices, hence software must become more reliable.

1.1 Objective and Aims

It is widely accepted that, if correctly applied, formal methods can increase the reliability of programs. Techniques of formal specification and verification have been studied since the end of the 1960s. However, it seems that formal methods have yet little impact on the practice. Suggested causes include lack of adequate tools, lack of mathematical sophistication of the engineers, incompatibility with current development techniques, high costs of application, as well as over-selling by advocates.

As a consequence of these causes, a lightweight approach to formal methods was proposed by [Jackson and Wing, 1996]. This approach emphasises on partiality, e.g. partiality of language, partiality of modelling, partiality of analysis, and partiality in composition. As another consequence, specialisation of the engineers was demanded. [Wing, 1985] introduced the idea of specification firms and [Easterbrook, 1996] suggested independent verification and validation agents.

In our opinion, specification engineering can get further impulses from other fields of computer science. In particular we believe that many problems and techniques of knowledge engineering are similar to those in formal specification. Hence, tools and methods used in knowledge engineering could be applied in specification engineering.

The aim of this work is to investigate how tools, methods, and results of knowledge engineering can be used for specification engineering. Such knowledge transfer can save much time in the development of formal specification towards an accepted and used method in software development. In that way this work will be a contribution to the work on improving formal methods towards industrial use.

1.2 Results

The results of this thesis are diverse. First, we compare knowledge engineering and specification engineering as two branches of computer science that have much in common. Basically, both fields deal with the formal representation of knowledge of a domain. We do not claim that this comparison will be complete but it might start a discussion on the issue. Further, many tools and methods for knowledge engineering were developed and applied, and today, knowledge engineering itself is an industry. In our opinion, specification engineering can benefit from knowledge engineering to become an industry itself.

In order to provide further evidence, we will extend the work by [Leemans et al., 1993] and [Treur and Willems, 1994] on the verification of knowledge bases. We will abstract from knowledge bases and consider logical theories, which can also be formal specifications. The main idea behind the introduced verification method is the consideration of a world description. This means, not all situations but all possible or desired situations will be checked. In that way our theory serves the purpose of an early error detection, as it is suggested by [Jackson and Wing, 1996].

Further, a specification example will be given. For this we will use a specification language designed for specifying knowledge based systems. Additionally, a development environment exists, which can be considered as a knowledge representation system. In this way, we show that it is possible to develop specifications within knowledge representation systems. The specification will also be verified according to the presented notions.

Finally, we will show that the suggested approach fits well into stdSEM, the Standard Siemens Development Methodology [stdSEM, 1997], which in turn shows that our approach could be used in practice. As a special case, we show that it is possible to outsource the process of formal specification and verification. This is closely related to the ideas of both specification firms and verification and validation agents.

1.3 Outline of the Thesis

The structure of this thesis is the following: In the next chapter we will compare knowledge engineering and specification engineering. We will show that both disciplines of computer science are closely related, and we propose to investigate formal specification in the framework of knowledge engineering.

In chapter 3 we will investigate some notions of formal verification, like consistency, correctness, and decisiveness. The presented verification method considers a world description explicitly given as well as a set of goals that need to be verified. Previous work on this topic was especially related to the verification of knowledge bases. This will be generalised by considering arbitrary universal theories as well as arbitrary quantifier free formulas or existential formulas as goals.

In chapter 4 we will first introduce the specification framework DESIRE and its development environment Destool. Afterwards, we will introduce a simple telecommunication world which will be specified using DESIRE and Destool. Finally, this specification will be verified according to the notions of verification that we introduce in chapter 3.

The results of this work will be summarised and discussed in chapter 5. There we will also relate this work to stdSEM, the Standard Siemens Development Methodology. Finally, in chapter 6, we are goint to make some concluding remarks and we will present possible future research directions.

Chapter 2

Comparing Specification & Knowledge Engineering

In the late 1960s the notion of software engineering was first introduced. In the classical sense, a team is responsible for building a software product using engineering principles, including all technical and non-technical aspects [Sommerville, 1992]. Thus the notion of software engineering stands for a set of software building methodologies.

Later, the idea of formal software specification was established as a method for improving software reliability. Unfortunately, only few researchers [Wing, 1985; Easterbrook, 1996] seem to look at specification as an independent process that can be carried out by specialised firms. Considering building a specification to be an engineering task leads to the notion of specification engineering, a notion that is hardly used today.

Knowledge engineering is a discipline of artificial intelligence which is under research since the end of the 1950s. Up to now, knowledge engineering has gained much recognition. For instance, companies used knowledge based systems to save millions of dollars and the development of knowledge based systems became an industry itself. Therefore, to be a knowledge engineer has long been an accepted profession and many tools and guidelines to construct knowledge based systems, e.g. expert systems, were developed and used.

In this chapter we will give a short introduction to specification engineering and knowledge engineering. We will show the aims, problems, and concepts of both approaches. Further, we will introduce the participants of both engineering tasks, as well as the properties of good representations. When comparing both notions we will put special emphasis on the similarities. This will lead to the view that many notions, methods, and tools of knowledge engineering can contribute to specification engineering.

2.1 Introduction to Specification Engineering

Specification engineering is the branch of software engineering that uses the principles of formal methods to build specifications. It is characterised by the process of building and evaluating a description of a problem, i.e. a specification, with engineering principles. Specification engineering will be carried out by specification engineers. We will restrict ourself to software engineering, though specification engineering can also be applied to hardware development. In this section we will describe why specification engineering is needed and how it is performed. In doing so, we will emphasise industrial needs. Finally, we will show some limitations of today's approaches.

2.1.1 The Aim of Specification Engineering

The final product of specification engineering is an evaluated and accepted description of a problem, i.e. a specification. A specification should form an adequate basis for the further program development. Thus it must give an exact description of what the software should do. Then it is left to the programmer to decide how to achieve the specified goals. A specification must also be understandable for the customer, because it becomes part of the software documentation and most often, it is the common ground for the contract between customer and supplier.

Formality is an important concept in specification engineering. The question why formal methods should be used is often discussed. Already the fact that a specification must be an exact description of a problem might be reason enough but a better argumentation that hits specification engineering at the heart of its notion is given by [Holloway, 1997]:

> Software engineers strive to be true engineers; true engineers use appropriate mathematics; therefore, software engineers should use appropriate mathematics. Thus, given that formal methods is the mathematics of software, software engineers should use appropriate formal methods.

This reasoning obviously holds for specification engineers too, because we introduced specification engineering as a discipline of software engineering. Hence, specification engineers should use appropriate formal methods and the outcome of their work will be a formal specification.

2.1.2 The Participants in Specification Engineering

In classical software engineering often only two players are identified: the customer, who is also the user, and the software engineer, who is also the programmer. We consider three participants:

1. the customer,
2. the programmer, and
3. the specification engineer.

Actually the user of the software product is often an extra player, too, but we assume that the customer is in contact with the end-user. In that way the user does not form an active part in specification engineering as we introduce it.

The Customer's Role

[Turski and Maibaum, 1987, p. 19] describe the ideal customer as:

> ... capable of, and willing to, analyse the application domain, write a consistent and sufficiently complete descriptive theory of it and also prove to his eternal satisfaction that this theory is a correct abstraction of the application domain.

In practice, however, such customers are rare. We assume the customer to possess good knowledge of his domain of interest and to have an idea of what he wants the system to achieve. Normally, this idea is provided by the customer in form of a vague requirements specification given in natural language. The final specification and its validation are products of cooperation between the specification engineer and the customer.

The Programmer's Role

The programmer is the user of the specification. According to the formal description of the problem he writes a computer program. Usually, the programmer does not have much knowledge about the application domain. His work will be judged primarily with respect to the specification. Normally, his communication partner is the specification engineer.

The Specifier's Role

The specification engineer is a specialist in representation. He has sufficient experiences in applying different specification languages and tools to software projects. The specifier is the central contact to the customer and also to the programmer.

The specification engineer is usually not a specialist in the domain at hand. Therefore, it is his task to obtain the information needed to construct a formal representation from the customer or from documents the customer provides. [Easterbrook et al., 1998] report their experiences using specifiers not familiar with the application domain. The advantage is that the engineer does not share the same assumptions with the customer. Therefore, the engineer will question everything that is not made explicit or left to more than one interpretation. In that way, many minor problems will be revealed, especially unstated assumptions and inconsistent usage of terminology.

After constructing the specification, the specifier proves certain properties of the specification, like consistency and correctness. He might also be responsible for creating a prototype that can be further validated. Note, this does not mean that the specification language itself should be executable. (We recommend to read [Hayes and Jones, 1989] and [Fuchs, 1992] for a discussion on this issue.)

2.1.3 Properties of a Good Specification

Specifications have to fulfil a set of characteristics to be valuable in the software development process. [IEEE, 1984] identifies the following general properties:

1. Unambiguity - each requirement stated in a specification has only one interpretation.
2. Completeness - all significant requirements are included, and responses to all possible inputs are defined.
3. Consistency - there are no requirements that contradict each other.
4. Modifiability - structure and style of the specification support changes to be made completely and consistently.

Another property is verifiability. In our view a specification is verifiable, if it is possible to reason formally about it[1]. Thus, verification is a formal and internal view related to concepts like completeness and consistency. In this way it corresponds to the generally accepted question:

[1]Note that there exists a second meaning of verifiability, which is related to the correspondence of the final program to its specification.

'Are we building the product right?' [Boehm, 1981]

In contrast to verification the notion of validation is used. Validation is an informal and external view, associated with the notion of satisfaction of the customer. Validation is best characterised by the question:

'Are we building the right product?' [Boehm, 1981]

The process including both verification and validation is called evaluation.

2.1.4 Fundamental Concepts in Specification Engineering

Specification engineering shares many fundamental concepts with other areas in computer science. [Clarke and Wing, 1996] identify the following:

- Abstraction - process of removing details from a representation;
- Composition - of methods, specifications, models, theories and proofs;
- Decomposition - of global properties into local ones;
- Reusability - of models and theories; this can be compared with program modules;
- Combination - of mathematical theories, to specify different requirements with different, preferably better suited, languages, like it is investigated within the field of viewpoint specifications, e.g. [Ainsworth et al., 1994] and [Bowman et al., 1995];
- Data structures and algorithms - for efficient formal reasoning.

The further development of these concepts will also lead to an improvement in specification engineering.

2.1.5 Problems in Applying Specification Engineering

In specification engineering we face many difficulties. The major problem is the acceptance by the industry. [Clarke and Wing, 1996] identify three general problems that have to be solved:

1. Integration of methods. Though specification engineering is a formal process it has to be combined with informal methods, graphical representation formalisms, and natural language. Further, the integration of tools, like model checking tools and automated theorem provers has to be considered.

2. Integration with the software development process. Specification engineering is an approach within software engineering. Especially when using formal methods 'Thou shalt not abandon thy traditional development methods', as [Bowen and Hinchey, 1995b] point out. Instead, formal methods should complement methods already used. [NASA, 1995] describe when and how to use formal methods in a project. Further, it is worth exploring how results from specification engineering can be used in later phases of the development process, like in the testing phase.

3. Education and technology transfer. [Holloway and Butler, 1996] identify a 'build it and they will come' expectation on the part of the formal methods developers but as in other areas, the success of a method depends on the practitioners. Therefore, it is necessary to train students and engineers, and to establish links between industry and universities. In that way, tools and methods, experiences and problems, as well as educated staff can be exchanged.

2.1.6 Demands on Tools and Methods

[Bowen and Stavridou, 1993] identify two further reasons for the low acceptance of formal methods: first, formal specification is time consuming and second, highly cost-intensive. A related problem is that these costs can hardly be predicted, because no generally accepted cost models exist [Craigen et al., 1993]. Further, the notations used in formal specification rely much on mathematics and formal logic; notations, the customer is usually not familiar with. Therefore, the specification cannot be used as a communication medium.

To overcome the introduced problems, [Clarke and Wing, 1996] demand the following criteria for tools and methods used in specification engineering:

- Early payback - benefits as soon as formal methods are used;
- Incremental gain for incremental effort;
- Multiple use - of tools, methods, and experiences in different projects;
- Integrated use - of tools and methods within the applied software development methodology;
- Ease of use - of tools, like compilers in programming;
- Efficiency - especially time efficiency of the tools;
- Ease of learning - of notations and tools;

- Error detection oriented - finding errors is more desirable than certifying correctness;
- Focused analysis - focused on special aspects of the system;
- Evolutionary development - allowing partial specification and selected verification.

2.1.7 A Lightweight Approach to Formal Specification

In order to fulfil some of the demanded properties and therefore, to make formal specification more attractive for the industry, a lightweight approach to formal methods was introduced by [Jackson and Wing, 1996]. They suggest:

1. Partiality in language. Specification languages are often considered as general mathematical notations. Unfortunately, this generality is reached at the expense of clarity and analysis. Thus it makes some specification languages unsuitable for practical use.
2. Partiality in modelling. It is important to realise that a complete formalisation of a problem is infeasible. Therefore, it is necessary to decide on the parts that merit the costs of formal specification.
3. Partial analysis. No sufficiently expressive specification language can be decidable and therefore, a sound and complete verification is impossible. Since most specifications contain errors, one has to ask for the properties that should be verified. Consequently one should concentrate on detecting errors reliably early in the development process.
4. Partiality in composition. Often a single partial specification is not sufficient for a large system. The idea is to construct a specification of interlocking partial specifications. Unfortunately, the mechanism for proving properties of partial specifications, like consistency, is understood for only some specification languages [Boiten et al., 1997].

Recently, a paper describing the experiences of the lightweight application of formal methods was published by [Easterbrook et al., 1998]. They present three case studies of the successful application of formal methods at NASA. The studies concern the specification, verification, and validation of fault protection software for the International Space Station and the Cassine deep space mission.

2.2 Introduction to Knowledge Engineering

[Russel and Norvig, 1995] define knowledge engineering as the process of building a knowledge base. [Fensel and van Harmelen, 1994] extend this by defining knowledge engineering as the branch of software engineering which deals with the construction of knowledge based systems. We will explicitly include the process of verification and validation of knowledge bases to these definitions. Knowledge engineering is carried out by knowledge engineers. In this section we will introduce the principles of knowledge engineering, its notions and its methods.

2.2.1 The Aim of Knowledge Engineering

Basically, there are two main approaches: the functionally orientated view and the modelling view. Whereas the first is related to the way of how human reasoning can be represented in a computer, the second deals with the problem of modelling systems in the world. In this way, 'the content of a knowledge base refers to an objective reality instead of an agent's "mind"' and 'knowledge is much more related to the classical notion of truth intended as correspondence to the real world, and less dependent on the particular way an intelligent agent pursues its goals' [Guarino, 1995].

We consider both views relevant but it must be possible to represent them separately. Actually, this is most often done. In knowledge representation systems the knowledge base and the inference mechanism are distinguished: one to hold the facts and rules of the world, and the other to model the reasoning process. In this thesis we will mainly concentrate on the modelling view.

2.2.2 The Participants in Knowledge Engineering

In knowledge engineering we primarily distinguish three participants:

1. the user,
2. the expert, and
3. the knowledge engineer.

The User's Role

The user of a knowledge engineering product is usually untrained or at least not a specialist in the domain. He needs the expert knowledge occasionally but cannot consult the expert. Therefore, the user needs the knowledge based system to obtain the information he wants. It is also important that the reasons for suggested decisions are clearly presented.

The Expert's Role

Experts are the specialists in their domain. They have special capabilities to solve problems even if there does not exist a single solution to it. Experts use heuristic knowledge and experiences and have a good general knowledge.

Unfortunately, experts are often unconscious about the reasons why they did something or not. Further, many experts are not trained in using representation tools, so that knowledge acquisition often becomes tricky.

The Knowledge Engineer's Role

A knowledge engineer is someone who investigates a particular domain, determines what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

Often, the knowledge engineer is trained in representation but is not an expert in the domain in question [Russel and Norvig, 1995]. It is his task to elicit the required knowledge from the expert, to structure it, and to create the formal representation. Finally, the knowledge engineer is responsible for verification and validation of the knowledge base.

2.2.3 Knowledge Representation

Knowledge representation is a mapping of parts of the world in a computer-tractable form. Each individual representation is called a sentence and sentences are expressed in a language called a knowledge representation language. Knowledge based systems are used to store knowledge, to reason about it, and to make it accessible. The process of obtaining the knowledge is called knowledge acquisition.

Knowledge Representation Systems

Knowledge representation systems, also called knowledge based systems, are designed to represent knowledge and to infer new facts from given data. [Dignum and van de Riet, 1991] define a knowledge based system as follows:

> A knowledge based system is a system that maintains a source of information (the knowledge base) in a way such that a user can communicate with the system as if he communicates with another user having access to that information.

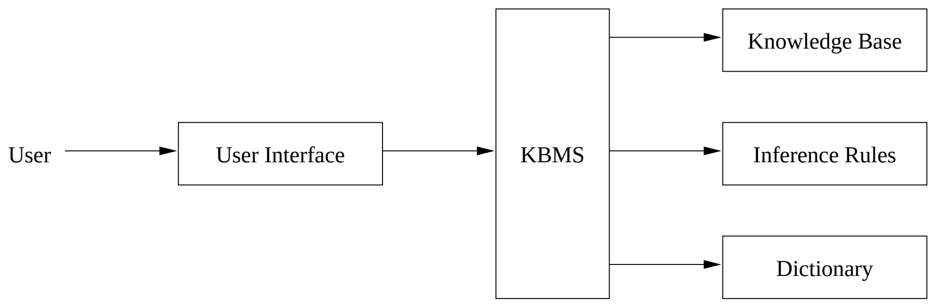


Figure 2.1: Components of a Knowledge Representation System

This is a very general definition. Unfortunately, it does not give an idea about the components of a knowledge based system. Therefore, figure 2.1 gives an overview of the components of a knowledge based system. In addition we will shortly describe each of those now:

- The user interface. Ideally it contains both a natural language interface and a graphical interface. In that way the user can easily communicate with the system but is also able to obtain an abstract overview of some parts of the knowledge given in the system.
- The knowledge base management system. The KBMS manages all the other components. We distinguish the following activities: on the one hand, it handles the queries of the user interface, and on the other hand, it handles the updates on the knowledge base. This also includes the process of managing the integrity of the knowledge base, the inference rules, and the dictionary.
- The dictionary. The elements of the language to describe the knowledge, like words and symbols, are given in the dictionary. This is similar to the notion of a signature of a formal language.
- The inference rules. They determine which inferences can be made by using the knowledge base. Often they are integrated in the system in such a way that they cannot be changed. If the knowledge is represented in first-order logic, the inference rules might contain the Modus Ponens but also rules from the resolution calculus.
- The knowledge base. This is, of course, the most important part of the knowledge based system. A knowledge base is a set of statements that describe facts and rules that hold in the actual world, as well as a set of constraints that must hold in all possible worlds. This corresponds to what [Russel and Norvig, 1995] call general knowledge about the domain and a description of the specific problem instances.

Knowledge Acquisition

Knowledge acquisition is the process of gathering the knowledge required in the knowledge based system. According to the way the knowledge is collected, [Horn, 1990] distinguishes four forms of knowledge acquisition:

1. The knowledge engineer collects the knowledge from the expert.
2. The expert is capable of inserting the knowledge by himself.
3. An inference algorithm extracts new knowledge from already present data.
4. A text analysing program extracts the knowledge from documents.

In the first case, the knowledge engineer will usually interview the expert. One problem, for instance, is that the knowledge engineer has to understand the expert's terminology. We already mentioned the problems of the second case, when we introduced the expert's role in knowledge engineering. The last two cases belong to the group of automated knowledge acquisition. We believe that these approaches are promising but not yet applicable. Nevertheless, the collected knowledge has to be expressed in a suitable language.

Knowledge Representation Languages

In order to express the knowledge in a computer-tractable form, we need a knowledge representation language with a well defined syntax and semantics, i.e. a formal language. [Russel and Norvig, 1995, p. 158] consider such a language to be a logic, because most of the principles of logic apply at this general level to formal languages.

A good knowledge representation language should combine the advantages of both formal and natural language. First, it should be as expressive and concise as natural language, so that everything can be said, and second, it should be as unambiguous and context insensitive as a formal language, so that everything is interpreted in only one way. Finally, the knowledge representation language should be efficient in the sense that there exists an inference procedure that can derive new information from the given knowledge within a reasonable time.

2.2.4 Properties of a Good Knowledge Base

A good knowledge base has to possess some properties, formal and informal ones. First of all, it must be unambiguous, clear, and correct [Russel and Norvig, 1995]. Further, the knowledge should be presented in a flexible and modular way, so that it can easily be changed. It must be processible and transferable, such that facts can be derived and communicated to the user. Ideally, it supports the representation of uncertain knowledge [Friedrich et al., 1990].

Additionally, we would like the knowledge base to be consistent and complete. Other properties are introduced by [Treur and Willems, 1994]: empirical foundedness and well-informedness. A knowledge base is empirically founded if it is able to give always the right answer to a question, and it is well-informed if it does not contain superfluous information, like redundant or subsumed rules.

2.2.5 Fundamental Concepts in Knowledge Engineering

Knowledge engineering is based on fundamental concepts of many areas in computer science and other sciences, like psychology, philosophy, and linguistics. Such concepts are:

- Abstraction - from unnecessary details;
- Composition - of tasks, knowledge bases, as well as verification methods;
- Decomposition - of tasks and knowledge;
- Reusability - of knowledge and therefore knowledge bases, in that way 'knowledge can in principle acquire a value *per se*' [Guarino, 1995];
- Data structures and algorithms - for knowledge representation and inference procedures;
- Integration - of achievements of other sciences, like philosophy and linguistics.

The further development of these concepts will also lead to an improvement of knowledge engineering. Further, these concepts are not only valuable to knowledge engineering. Psychology, linguistics, and philosophy can in turn profit from investigations on these concepts as well.

We just presented a brief introduction to knowledge engineering. We mentioned aims, participants, and concepts of knowledge engineering. Further, we introduced a general framework of a knowledge representation system, as well as properties of knowledge bases. The interested reader might have noticed some similarities to specification engineering. We provided these information in order to give the necessary background for the comparison of knowledge and specification engineering which will be drawn in the next section.

2.3 Differences and Similarities

Sometimes, knowledge engineering is compared to software engineering. Here many differences can be found. We will take another route and compare knowledge engineering with specification engineering. While reading the last two sections, the reader might have noticed that both disciplines have much in common. However, we will first look at some differences between both approaches.

2.3.1 Differences

The main difference between the two approaches lies in their goals. While specification engineering is concerned with the development of a formal representation of an artificial system, i.e. a program, knowledge engineering deals with the formal development of a system that represents a part of the real world and reasons about it.

Therefore, the validation processes of the systems differ. Specifications have to be validated against the idea of a customer, whereas a knowledge base is validated against the real world, actually against an idea of the real world. On the other hand, programs become part of the world, and therefore, knowledge about a program is also knowledge about the world. Thus, a specification is in fact validated against a possible future world.

Further, knowledge engineering includes the development of an appropriate user interface, an inference engine, and a knowledge base. Specification engineering only develops specifications. There are, however, notable exceptions to this rule. The specification language ″Larch″, for instance, was designed with tool support in mind [Jackson and Wing, 1996]. Such tools are user interfaces and automated theorem provers. In general, a theorem prover is the implementation of an inference relation, i.e. it is an inference engine. Therefore, the view on specification engineering can to be extended in order to capture the development of appropriate tools as well.

The participants in knowledge engineering and specification engineering are surely different. On the one hand, we have programmers, customers, and specifiers and on the other hand, there are users, experts, and knowledge engineers. Of course, the user is not a programmer, and the expert will hardly buy the knowledge based systems.

A specification is traditionally a description of the I/O behaviour of a system, without considering how this behaviour can be realised. In contrast, in knowledge engineering one is much concerned with the processing of knowledge. [Fensel and van Harmelen, 1994] identified this as another distinguishing property of both approaches.

2.3.2 Similarities

After presenting some differences between knowledge engineering and specification engineering, we will now turn to the similarities. The reader will see that some differences are not as strict as they appear to be, and some other differences should preferably be overcome.

The "What" and "How" Problem

First, we discuss the argument given by [Fensel and van Harmelen, 1994]. The problem they address is called the "What" and "How" problem in specification engineering. [Sannella, 1988] writes:

> "high-level specifications" are descriptions of *what* is required. This is contrasted with "programs" which suggest *how* the desired result is to be computed.

In knowledge engineering both cases are represented separately. On the one hand, there is the knowledge base that only includes the knowledge about *what* is true, and on the other hand, the inference procedure figures out *how* to turn the facts into solutions to problems. In specification engineering there does not exist an inference procedure but there certainly exists a proof theory of the specification language. By implementing this proof theory, we have an inference procedure. Such implementations are automated theorem provers, tools which are often applied in specification engineering. Hence, we could show that both approaches deal with the "What" and "How" problem.

Similarities of the Representation

Another sign of evidence for the similarity of knowledge engineering and specification engineering is the similarity of the representation languages used. First of all, first-order logic is often applied in both disciplines. Further, declarative programming languages, e.g. functional languages like Miranda [Turner, 1986], or logic languages like PROLOG [Kowalski, 1979; Bratko, 1990] are used preferably. [Fuchs, 1992] argues in favour of such languages for formal specification and [Hu, 1987] introduces such languages for the development of knowledge based systems. In fact, because both activities deal with the representation of problems, it is very natural to use the same languages.

Turning to purely theoretical aspects of knowledge engineering and specification, we find that both basically deal with formal logic, its model theory and its proof theory. From a theoretical point of view, a knowledge base is a formal theory

[Goltz and Herre, 1990] and a specification is one, too [Turski and Maibaum, 1987]. The possibility to infer new facts from knowledge bases and specifications, and thus the possibility to verify them, depends on the proof theory of the applied logic.

The Properties and Problems

Knowledge bases and specifications share many properties. Both should be unambiguous, complete, consistent, modular, and verifiable. Further, clarity and reusability of the representations as well as expressiveness of the language are desired properties. Both knowledge engineering and specification engineering use abstraction, composition, decomposition, and combination of mathematical theories, like different formal logics, as fundamental concepts. Improvements made on these concepts are directly beneficial to both approaches.

Another fundamental problem in both disciplines is that of explicit and implicit knowledge, denoted as explicit and implicit requirements in specification engineering. With implicit knowledge we do not mean the information that can be derived by the inference mechanism or by a computer program. It is the knowledge that an expert or a customer is unconscious about. These are assumptions not shared with the engineer and hence not included into the representation of the problem. To detect and to remove implicit knowledge from a representation is a very important task.

Tools

As we said earlier, knowledge engineering includes the development of an appropriate user interface, an inference engine, and a knowledge base. Specification engineering only develops specifications. Though this has been mainly true up to the present specification engineering should face the problem of tool development harder. Here we propose the development of suitable user interfaces and proof tools, to fulfil the demands of ‘ease of use’ and ‘ease of learning’. We could imagine, a general specification tool looks like those given in figure 2.2.

This picture is closely related to a another commonly accepted depiction of a knowledge representation system. If one incorporates the data dictionary into the formal specification tool and states the inference rules of the proof tool explicitly, then this representation resembles the knowledge based system given in figure 2.1.

Similarities of the Participants

The user and the programmer are the users of the products they get, i.e. a specification or a knowledge base. Though user and programmer might give some

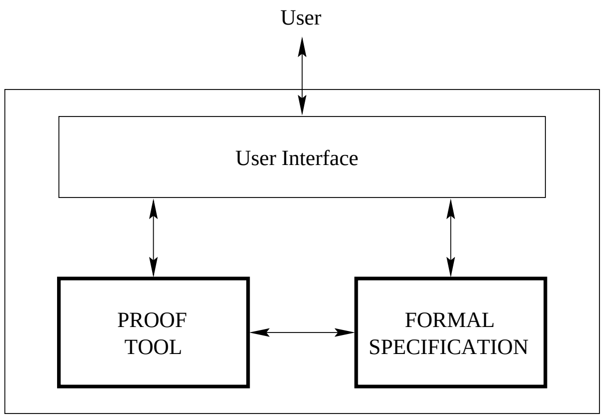


Figure 2.2: An Outline of a Specification Tool

comments, e.g. on the user interface or on the efficiency, they are usually not involved in the process of making decisions about the functionality of a system. Both user and programmer have usually not enough domain knowledge to do so.

Second, the expert and the customer are the domain experts. Both expect the system to behave in a certain way. They define the requirements and constraints on the system and validate the final product. Finally, the engineers have to capture the ideas of the expert and the customer. They create a formal representation of these ideas and verify it.

Summary

In analogy to the table presented by [Russel and Norvig, 1995, p. 219] we will summarise knowledge engineering and specification engineering as activities that essentially consist of the following four steps:

(1)	Choosing a representation formalism
(2)	Building a theory
(3)	Deciding on a proof theory
(4)	Inferring facts

Table 2.1: The Essence of Specification and Knowledge Engineering

The key point of both activities is to write down a description of a problem and then to use the definition of the language to derive consequences. In both cases the engineer only has to decide what objects and relations are worth representing, and which relations hold among which objects.

2.4 Conclusion

'Early payback', 'ease of use', and 'ease of learning' are some of the demands on specification engineering. In order to fulfil these demands we certainly need specification tools and therefore, we have to extent the view on specification engineering. Specification engineering should not only deal with the construction of specifications but also with the construction of specification and verification tools. Unfortunately, building new tools is always an expensive task.

Therefore, and as a conclusion of this chapter, we propose to investigate notions, methods, and tools from knowledge engineering for their applicability in specification engineering. We believe that achievements made in knowledge engineering can be beneficial to specification engineering. Applying formal methods to knowledge engineering has been proved successful [Fensel and van Harmelen, 1994]. We shall now investigate the other direction.

Chapter 3

Verifying Static Properties of Specifications or Knowledge Bases

In chapter 2 we identified composition and decomposition as fundamental concepts in specification and knowledge engineering. The idea is to construct specifications or knowledge bases of interlocking parts, e.g. partial specifications or knowledge bases. For example, [Abadi and Lamport, 1993] examine how to compose specifications and [Brazier et al., 1995] investigate how to use the composition principle for structuring knowledge based systems.

Once a specification or knowledge base is decomposed, it is also possible to decompose the proofs for them. [Engelfriet et al., 1997], [Cornelissen et al., 1997], and [Jonker and Treur, 1998] introduced the compositional verification method as a framework for verifying composed systems. The idea is to prove the properties of interest for a higher level on the basis of assumptions at the lower level that guarantee these properties. This procedure will be applied until primitive components are reached, which can then be verified by using other techniques.

We will investigate properties of primitive components. Our work will be based on [Treur and Willems, 1994] but extends their study in two ways: first, we study not only knowledge bases but universal theories in general. Hence, this work can also be used for verifying formal specifications. Second, we will use a set of goals which will be given explicitly. This set of goals does not only contain literals but also quantifier-free sentences or even more complex sentences.

First, we will give a short overview of necessary background, like notions from model and proof theory. Afterwards, we will introduce notions closer related to our approach, e.g. the notion of a world description. Further, we will introduce the notion of forcing, which is essential for the theory we will develop. Finally, we are going to formally define several properties a logic theory should fulfil.

3.1 Preliminaries

We assume familiarity with the first-order predicate calculus and the standard notions of set theory. For an introduction, we recommend to read [Chang and Keisler, 1990], [Goltz and Herre, 1990], or [Rothmaler, 1995].

3.1.1 Syntax

A signature, denoted by Σ, is a set of symbols for relations, functions, and constants. $L(\Sigma)$ is the first-order language based on Σ. If Σ is not specified we write simply L. Atoms $At(\Sigma)$ and literals $Lit(\Sigma)$ are defined in the usual way. Atoms and literals are called ground if they do not contain any variables. We distinguish three subsets of atoms: input atoms $InAt(\Sigma)$, internal atoms $InternalAt(\Sigma)$, and output atoms $OutAt(\Sigma)$. The sets of input atoms and internal atoms as well as internal atoms and output atoms are distinct, which must not hold for the sets of input atoms and output atoms. A similar distinction is used for literals.

Formulas can be constructed from atoms, logical connectives, like $\neg$, $\wedge$, $\vee$, $\rightarrow$, and the quantifiers $\forall$ and $\exists$. A formula containing no free variables is called a sentence. Formulas of the form $L_1 \wedge \ldots \wedge L_n \rightarrow L$, where $L_1, \ldots, L_n, L$ are literals, are called program formulas. A formula is said to be open if it contains no quantifiers. Open formulas can be considered as universal prenex sentences. A universal theory is a set of open formulas. The existential closure of an open formula F is denoted by $\exists(F)$ and is called an existential sentence. A $\Sigma_1(Q)$-sentence is of the form $\exists\, xF(x)$, where $F(x)$ is a conjunction of literals.

For later use we adopt the following abbreviations. For a set S of program formulas, let $PF(S)$ be the set of all program formulas of the language $L(S)$, $KL(S)$ the set of conjunctions of literals of S, and $Ex(S)$ the set of all prenex existential formulas (including the set $KL(S)$) whose quantifier free part belongs to $KL(S)$.

3.1.2 Notions from Model Theory

Herbrand Universe and Herbrand Base

Let S be a universal theory, i.e. a set of open formulas, and $\Sigma(S)$ its signature. Then $U(S)$, the Herbrand universe of S, is the set of all ground terms constructible from the function or constant symbols of S, i.e. the set of all variable-free terms of $\Sigma(S)$. Without loss of generality, we can assume that there always exists a constant symbol in Σ, i.e. $U(S)$ is never empty. For example, if

$$S = \forall\, x \,\forall\, y \;\, p(a, f(x), g(y, b)), \text{ with } \Sigma(S) = \{p, f, g, a, b\},$$

x and y are variables, then

$$U(S) = \{a, b, f(a), g(a, a), g(a, b), f(f(a)), \ldots\}.$$

The Herbrand base $B(S)$ of a universal theory S is the set of all ground atoms of the signature of S. For example, if

$$S = p(a, g(x)) \wedge q(f(x, b)), \text{ with } \Sigma(S) = \{p, q, f, g, a, b\},$$

x and y are variables, then

$$B(S) = \{p(a, a), p(a, b), p(g(a), b), q(f(g(a), g(g(b)))), \ldots\}.$$

Herbrand Structure and Herbrand Interpretation

A structure $\mathcal{A}$ is a pair $(A, I^{\mathcal{A}})$ consisting of a set A, called universe, and an interpretation function $I^{\mathcal{A}}$. A structure $\mathcal{A} = (A, I^{\mathcal{A}})$ is called Herbrand structure for S, if

1. $A = U(S)$,
2. $I^{\mathcal{A}}(a) = a$ for all constants, and
3. $I^{\mathcal{A}}(f(t_1, \ldots, t_n)) = f(I^{\mathcal{A}}(t_1), \ldots, I^{\mathcal{A}}(t_n))$ for all function symbols f and terms t_i.

Obviously, a H-interpretation I assigns its syntactical representation, i.e. its name, without any evaluation to the constant symbols and function symbols.

It follows from above that a H-interpretation I of an universal theory S is a subset of $B(S)$, i.e. $I \subseteq B(S)$. Let I be the union $I_1 \cup \neg\, I_0$ of two disjunctive sets $I_1, I_0 \subseteq B(S)$, where I_1 is the truth set and I_0 is the false set of I (where $\neg\, I_0 = \{\neg\, \varphi \mid \varphi \in I_0\}$). If $I_1 \cup I_0 = B(S)$, then I is a 2-valued interpretation, else I is a 3-valued interpretation, where atoms occurring in $I_u = B(S) - (I_1 \cup I_0)$ are evaluated as unknown.

I is also used as a truth assignment $I : B(S) \to \{0, 1, u\}$, defining for $a \in B(S)$, $I(a) = x$ iff $a \in I_x$, $x \in \{0, 1, u\}$. The truth of a sentence can be expressed by using the strong Kleene truth tables (see table 3.1).

p	¬ p
1	0
0	1
u	u

p∧q	1	0	u
1	1	0	u
0	0	0	0
u	u	0	u

p∨q	1	0	u
1	1	1	1
0	1	0	u
u	1	u	u

p→q	1	0	u
1	1	0	u
0	1	1	1
u	1	u	u

Table 3.1: Strong Kleene Truth Tables

Truth Ordering and Knowledge Ordering

Usually, two partial orderings for the set $\{0, 1, u\}$ of truth values are distinguished. The truth ordering $<_t$ defined as $0 <_t u <_t 1$ is used to evaluate the truth values of sentences and the knowledge ordering $<_k$ defined as $u <_k 0, u <_k 1$ is used to compare interpretations and models.

The partial knowledge ordering $<_k$ can be extended to 3-value interpretations by defining $I \leq_k I'$ iff $I(a) \leq_k I'(a)$ for every atom $a \in B(S)$. Then, $\leq_k$ reduces to the set inclusion relation for interpretations, because $I \leq_k I'$ iff $I \subseteq I'$ [Witteveen, 1992]. Instead of $I \leq_k I'$ we will also write $I \leq I'$.

Herbrand-Model

A structure $\mathcal{A}$ is a model for S if every formula A in S is true in $\mathcal{A}$. We define $\mathit{Mod}(S)$ to be the class of all models of S. A Herbrand model, for short H-model, for S is one for which the universe equals $U(S)$. $\mathit{HMod}(S)$ denotes the set of all Herbrand models of S. It holds that $\mathit{HMod}(S) \subseteq \mathit{Mod}(S)$. Herbrand models can be represented by subsets $I \subseteq B(S)$, where I is the H-interpretation and it holds that $I(A) = 1$ for every formula $A \in S$. We call a H-model complete, if the H-interpretation I is a 2-valued interpretation. If I is a 3-valued H-interpretation, it is called a partial H-model.

The Truth Relation

Given an interpretation I, the partial evaluation val_I associated with I is defined as

$$\mathit{val}_I(\varphi) = \begin{cases} 1 & \text{if } \varphi \in I \\ 0 & \text{if } \neg\, \varphi \in I \\ u & \text{else} \end{cases} \qquad \mathit{val}_I(\neg\, \varphi) = \begin{cases} 1 & \text{if } \neg\, \varphi \in I \\ 0 & \text{if } \varphi \in I \\ u & \text{else} \end{cases}$$

for $\varphi \in B(S)$ and (w.r.t. the truth-order)

$$val_I(\varphi \wedge \psi) = min\{val_I(\varphi), val_I(\psi)\},$$

$$val_I(\varphi \vee \psi) = max\{val_I(\varphi), val_I(\psi)\},$$

$$val_I(\varphi \rightarrow \psi) = max\{val_I(\neg\, \varphi), val_I(\psi)\},$$

$$val_I(\exists\, x\varphi(x)) = max\{val_I(\varphi(x/t) : t \in U(S)\},$$

$$val_I(\forall\, x\varphi(x)) = min\{val_I(\varphi(x/t) : t \in U(S)\}.$$

We use the following transformation laws for negated formulas:

$$\neg\,(\varphi \rightarrow \psi) = \varphi \wedge \neg\,\psi, \qquad \neg\,(\neg\,\varphi) = \varphi,$$

$$\neg\,(\varphi \wedge \psi) = \neg\,\varphi \vee \neg\,\psi, \qquad \neg\,\forall\, x\varphi(x) = \exists\, x\neg\,\varphi(x),$$

$$\neg\,(\varphi \vee \psi) = \neg\,\varphi \wedge \neg\,\psi, \qquad \neg\,\exists\, x\varphi(x) = \forall\, x\neg\,\varphi(x).$$

A formula φ is true in an interpretation I, denoted $I \models_3 \varphi$, iff $val_I(\varphi) = 1$. Moreover we use: φ holds in I, φ is a consequence of I, φ follows from I, φ is satisfied in I, I satisfies φ, or finally, I is a model of φ.

Given a set of formula F, we call A a model of F iff A is a model of each φ in F, and we use the notion $A \models_3 F$. Often the truth relation is also defined in a set theoretic way: given a formula φ, then $A \models_3 \varphi$ iff $Mod_3(A) \subseteq Mod_3(\{\varphi\})$, and given a set of formula F, then $A \models_3 F$ iff $Mod_3(A) \subseteq Mod_3(F)$. We omit the index for complete models, e.g. for a 2-valued model A and a formula φ we write $A \models \varphi$ if φ holds in a 2-valued model A.

We write $S \models_H F$ if every Herbrand model of S is a model of F, i.e. $HMod(S) \subseteq Mod(F)$. If S is a universal theory and F is an existential sentence then it holds: $S \models F$ iff $S \models_H F$.

Refinement of Partial Models

While dealing with partial models it is often useful to consider refinements of such models. A model refines another one if it possesses the same or more knowledge, i.e. if it holds $M(a) \leq_k N(a)$ for all atoms a. As we already mentioned, we will write $M \leq N$ (or $N \geq M$) instead. Refinement is a partial ordering, because $\leq_k$ is one.

Lemma 3.1.1 (Refinement)
Let M_1 and M_2 be two arbitrary partial models w.r.t. a signature Σ and φ a formula. Then it holds:

$$M_1 \leq M_2 \quad \Leftrightarrow \quad \forall\, \varphi \in L(\Sigma) \;\; M_1 \models_3 \varphi \Rightarrow M_2 \models_3 \varphi.$$

Proof
($\rightarrow$) We have $M_1 \leq M_2$ and $M_1 \models_3 \varphi$ by assumption, then $M_1 \models_3 \varphi$ iff $Mod_3(M_1) \subseteq Mod_3(\{\varphi\})$. Because $M_1 \leq M_2$, it holds $M_1 \subseteq M_2$ and therefore, $Mod_3(M_2) \subseteq Mod_3(M_1)$. Because $\subseteq$ is transitive, it holds $Mod_3(M_2) \subseteq Mod_3(\{\varphi\})$, hence $M_2 \models_3 \varphi$.

($\leftarrow$) By contradiction. Suppose $M_1 \not\leq M_2$, then $M_1 \not\subseteq M_2$ and therefore, $Mod_3(M_2) \not\subseteq Mod_3(M_1)$. Hence there exists a formula φ such that $M_1 \models_3 \varphi$ but not $M_2 \models_3 \varphi$. This contradicts the assumption that for all φ it holds $M_1 \models_3 \varphi \Rightarrow M_2 \models_3 \varphi$. $\square$

Given a set of models V it is sometimes useful to consider the set $P(V)$ of partial models that can be refined to a model in V.

Example 3.1.1
In the following example, $x : *$ denotes the truth assigment of $*$ to x, where $* \in \{1, 0, u\}$, e.g. $p : 0$ means that p is false. Consider $V = \{N_1, N_2\}$, where $N_1 = \langle p : 0, q : 1\rangle$ and $N_2 = \langle p : 1, q : 0\rangle$. Then $P(V) = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7\}$, where $M_1 = \langle p : 0, q : 1\rangle$, $M_2 = \langle p : 1, q : 0\rangle$, $M_3 = \langle p : 0, q : u\rangle$, $M_4 = \langle p : u, q : 1\rangle$, $M_5 = \langle p : 1, q : u\rangle$, $M_6 = \langle p : u, q : 0\rangle$, and $M_7 = \langle p : u, q : u\rangle$. It is easy to see that every model M_i can be refined to either N_1 or N_2.

Conservativity and Monotonicity

Given a universal theory S and a set of partial models X both of signature Σ, we define the following properties:

1. conservativity w.r.t. X:
   For any partial model $M \in X$ and sentence φ it holds:
   $$M \models_3 \varphi \;\Rightarrow\; M \cup S \models_3 \varphi.$$
2. monotonicity w.r.t. X:
   For any two partial models $M_1, M_2 \in X$ and sentence φ such that $M_1 \leq M_2$ it holds:
   $$M_1 \cup S \models_3 \varphi \;\Rightarrow\; M_2 \cup S \models_3 \varphi.$$

3.1.3 Notions from Proof Theory

Given is a calculus $\mathcal{C}$ with an appropriated concept of proof. $X \vdash_{\mathcal{C}} F$ denotes that a formula F is derivable from a set of formulas X in $\mathcal{C}$. A calculus will be defined as a triple

$$\mathcal{C} = (\mathbf{S}, \mathbf{Q}, \vdash_{\mathcal{C}}),$$

where

1. $\mathbf{S}$ is a collection of sets of formulas, i.e. $\mathbf{S} \subseteq Pow(L(\Sigma))$, called the domain of $\mathcal{C}$. From now on we assume that $\bigcup \mathbf{S}$ contains only universal sentences.
2. $\mathbf{Q}$, the range of $\mathcal{C}$, is the set of formulas representing goals to be proved (or refuted). We suppose $\mathbf{Q} = \mathbf{Q}_0 \cup (\exists)\, \mathbf{Q}_0,$ where $\mathbf{Q}_0$ is a set of open formulas, and we refer to the elements of $\mathbf{Q}$ as query formulas or as goals.
3. Lastly, $\vdash_{\mathcal{C}}$ is the derivability relation, also called inference relation.

One may regard a calculus as a subsystem of a logic $L = (S, \vdash_L, \models_L)$ given by the set S of formulas, by a derivability relation $\vdash_L$, and by a truth relation $\models_L$. The calculi we consider can be regarded as subsystems of classical logic, CL.

Conservativity and Monotonicity

For later use, we need the inference relation $\vdash$ to possess two properties, conservativity and monotonicity. This will be defined for a universal theory S and a set of partial models X both of signature Σ in the following way:

1. conservativity w.r.t. X:
   For any partial model $M \in X$ and sentence φ it holds:
   $$M \models_3 \varphi \;\Rightarrow\; M \cup S \vdash \varphi.$$
2. monotonicity w.r.t. X:
   For any two partial models $M_1, M_2 \in X$ and sentence φ such that $M_1 \leq M_2$ it holds:
   $$M_1 \cup S \vdash \varphi \;\Rightarrow\; M_2 \cup S \vdash \varphi.$$

Constructive Calculi

In verification we are not only interested in the fact that a formula F follows from a theory S but also that F is derivable from S within an appropriate calculus. In order to compute F, a calculus is only appropriate if it is constructive. [Herre and Pearce, 1992] investigate constructive properties of different calculi, like SLD-resolution, SI-resolution, and Forward Chaining. [Herre, 1993a] extends this work by investigating constructive proofs for Π_2-sentences, i.e. for sentences of the form $\forall\, \bar{x}\, \exists\, y G(\bar{x}, y)$.

A calculus $\mathcal{C} = (\mathbf{S}, \mathbf{Q}, \vdash_{\mathcal{C}})$ is Herbrand-correct (H-correct), if for every $S \in \mathbf{S}$ and $F \in \mathbf{Q}$ it holds: if $S \vdash_{\mathcal{C}} F$ then $S \models_H F$. If $\mathcal{C}$ is correct w.r.t. classical logic then $\mathcal{C}$ is H-correct. The converse is not true [Herre, 1993a].

We will now turn to the notions of c-correctness and c-completeness for a calculus as they were presented by [Herre and Pearce, 1992] and [Herre, 1993a]. Here, 'c' in c-correctness and c-completeness stands for 'constructive'. Let $\mathcal{C} = (\mathbf{S}, \mathbf{Q}, \vdash_{\mathcal{C}})$ be a calculus. It is assumed that $\bigcup \mathbf{S}$ contains only universal sentences, and $\vdash_{\mathcal{C}}$ is correct for classical logic or H-correct.

Definition 3.1.1

1. $\mathcal{C}$ is called c-correct if for every theory $S \in \mathbf{S}$ and formula $F \in (\exists)\, \mathbf{Q}_0$, where $F := \exists\, xG(x)$, the following holds: if $S \vdash_{\mathcal{C}} F$ then there is a substitution σ such that $S \vdash_{\mathcal{C}} G\sigma$ and $S \models_{\mathcal{C}} G\sigma$.
2. $\mathcal{C}$ is called c-complete over a logic L if for every theory $S \in \mathbf{S}$ and quantifier free sentence $Q \in \mathbf{Q}_0$ satisfying $S \models_L Q$ the condition $S \vdash_{\mathcal{C}} Q$ is satisfied.
3. $\mathcal{C}$ is called strongly c-complete over a logic L if for every theory $S \in \mathbf{S}$, and open formula $F \in \mathbf{Q}$ it holds: if $S \models_L F$ then $S \vdash_{\mathcal{C}} F$.
4. $\mathcal{C}$ is said to be weakly c-complete over a logic L if for every theory $S \in \mathbf{S}$, quantifier free formula $F \in \mathbf{Q}_0$ satisfying $S \models_L F$ there is an open formula G such that $S \vdash_{\mathcal{C}} G$ with a substitution σ satisfying $F = G\sigma$.

According to [Herre and Pearce, 1992] there are calculi $\mathcal{C} = (\mathbf{S}, \mathbf{Q}, \vdash_{\mathcal{C}})$ satisfying the following conditions:

1. $\mathbf{S}$ contains every set of universal sentences, and $\mathbf{Q} = \Sigma_1$, and
2. $\mathcal{C}$ is c-correct and c-complete.

The Calculus of Forward Chaining

Now we will introduce the calculus of Forward Chaining, $\mathcal{C}(FC)$. It is a simple calculus containing only a few rules and therefore, it plays an important role in the field of expert systems. Forward chaining is defined for sets of program formulas, i.e. for formulas having the form

$$L_1 \wedge \ldots \wedge L_n \to L,$$

where the L_i and L are literals. The calculus of Forward Chaining, $\mathcal{C}(FC)$, is based on the following rules:

(substitution) $\dfrac{D}{D\sigma}$, if $D \in KL(S) \cup PF(S)$, σ a substitution;

(conjunction) $\dfrac{E \quad D}{E \wedge D}$, if $E, D \in KL(S)$;

$$\text{(modus ponens)} \quad \frac{K \quad K \to L}{L}, \quad \text{if } K \in KL(S),\ K \to L \in PF(S);$$

$$(\exists\text{-rule}) \quad \frac{F(x/t)}{\exists\, xF(x)}, \quad \text{if } F \in Ex(S),\ t \text{ a term.}$$

Furthermore, contraction and commuting of literals in formulas from $KL(S)$ are admitted. Then for a formula F we write $S \vdash_{FC} F$ if there is a proof for F from the initial set of program formulas S. The relation $\vdash_{FC}$ can be semantically characterised by the relation $\models_3$. Therefore, $\vdash_{FC}$ is also monotonic and conservative. The calculus $\mathcal{C}(FC)$ is c-correct and c-complete over the logic $L = (S, \vdash_{FC}, \models_3)$ [Herre and Pearce, 1992].

3.2 Notions for a Theory of Verification

In chapter 2 we mentioned that verification is a formal and internal view related to concepts like completeness and consistency. Though it might be possible to check these properties for all possible worlds, it is surely not feasible. Therefore, [Preece et al., 1992], [Treur and Willems, 1994], and [Yue, 1987] relate specifications or knowledge bases to a world description that is explicitly given. Usually, such description will be a sample set of situations that should, or should not be provable. In that way verification will be a process of checking the correspondence of the specification to its world description.

3.2.1 Formal Conceptualisation

As mentioned above, the approach taken here depends on a description of the world. Such a description will be given by a set of situation models, each describing a possible or typical situation in the world. It is assumed that relevant properties and interrelations of a part of the real world are described by the world description.

The process of creating the world description is a crucial step. First, the whole process of verification depends on it. Second, situations can be given in many different ways. For example, it is possible to give a set of axioms that serve as constraints for the situations. In this case, a world description is the set of all possible situations that fulfil these constraints. The problem we face here, is the problem of obtaining the set of constraints independent from the knowledge base or specification. It is possible that the knowledge base or the specification and the world description have the same biases and mistakes. However, in order to define the notions involved in verification it is sufficient to abstract from the process of creating the world description and to assume that the set of situations is given.

An actual situation is described as a Herbrand model N, where N is a truth assignment to the atoms in a signature Σ, i.e. a mapping:

$$N : At(\Sigma) \;\rightarrow\; \{0, 1\}.$$

A domain or world description is the set W of complete situation models, i.e. a set of Herbrand structures based on a certain signature Σ.

Because it is not always possible or desired to describe a situation completely, we also use partial Herbrand models. A partial model M w.r.t. a signature Σ allows to assign u as a truth assignment to atoms that are unknown, i.e. M is a mapping:

$$M : At(\Sigma) \;\rightarrow\; \{0, 1, u\}.$$

A complete model is a partial model that has no unknown atoms. When we refer to models, we mean partial models, otherwise we will denote complete models explicitly. From now on we will use N for complete models and M for partial models.

With partiality it is easy to define input and output models. An input model w.r.t. a signature Σ is a truth assignment that assigns u to all atoms outside $InAt(\Sigma)$. An output model can be defined similarly. For any (partial) model M, $In(M)$ denotes the input model that copies the input truth-assignment of M but assigns u to all other atoms. Similarly the output model $Out(M)$ copies the output truth-assignments but assigns u to all other atoms.

$$In(X)(a) \;=\; \begin{cases} X(a), & a \in InAt(\Sigma) \\ u, & a \notin InAt(\Sigma) \end{cases}$$

$$Out(X)(a) \;=\; \begin{cases} X(a), & a \in OutAt(\Sigma) \\ u, & a \notin OutAt(\Sigma) \end{cases}$$

Connected to the world description W are the sets $In(W)$ and $Out(W)$ as the sets of input models respectively output models that are associated to the situations in W. Now we state some simple but essential properties of input models. All three lemmata can be proved by the definition of refinement.

Lemma 3.2.1
For any partial model M it holds $In(M) \leq M$.

Lemma 3.2.2
For any two partial models M_1, M_2 such that $M_1 \leq M_2$ it holds $In(M_1) \leq In(M_2)$.

Lemma 3.2.3
For any world description W and operator P (as defined on page 26) it holds $P(In(W)) = In(P(W))$.

In the work by [Treur and Willems, 1994] only literals are considered as goals. In order to extend this view and to allow arbitrary quantifier-free sentences or even more complex sentences we need a set of formulas representing goals or queries . Following the definition of a calculus, we use the set $\mathbf{Q}$, $\mathbf{Q} = \mathbf{Q}_0 \cup (\exists)\, \mathbf{Q}_0$, where $\mathbf{Q}_0$ is a set of open formulas, as the set containing the goals that need to be proved.

In chapter 2 we argued that knowledge bases and specifications are both logical theories. Because the work presented in this chapter shall not be restricted to knowledge bases or formal specifications, we will basically consider sets of open formulas $S \subseteq L(\Sigma)$, i.e. universal theories.

3.2.2 Forcing for Verification

The notion of forcing is essential to the theory of verification. Forcing ensures that a goal holds in all possible situations that follow from a partial situation according to the world description. In a way, this sets a standard for the theory, determining what goals should be derivable given some partial model.

Definition 3.2.1 (Forcing)
Let W be a domain description w.r.t. a signature Σ, $M \in P(W)$ is a partial model, and $\varphi \in \mathbf{Q}$ a sentence. The model M *forces* φ within W if φ holds in every complete refinement $N \in W$ with $M \leq N$. We will use the notation $|\!\models_W$ that is defined by:

$$M \,|\!\models_W \varphi \quad \Leftrightarrow \quad \forall\, N \in W\ [M \leq N \Rightarrow N \models \varphi].$$

Example 3.2.1
For example, let p and q be input atoms and r a goal to be proved. Given is the world description $W = \{N_1, N_2, N_3\}$, where $N_1 = \langle p:1, q:1, r:1\rangle$, $N_2 = \langle p:0, q:1, r:1\rangle$, and $N_3 = \langle p:1, q:0, r:0\rangle$, as well as a situation $M = \langle p:u, q:1, r:u\rangle$. It holds that $M \,|\!\models_W r$, because $M \leq N_1$ and $M \leq N_2$, and $N_1 \models r$ and $N_2 \models r$. Because N_3 is not a refinement of M we know that r holds in all complete refinements of M, hence r is forced by M.

In order to deal with forcing, we need to show that some properties hold for the forcing relation. In general, we want the properties of forcing, inference, and truth to be closely related. Therefore, we will show that conservativity and monotonicity also hold for forcing. Actually, monotonicity of forcing might be seen as the reason for demanding monotonicity of the inference relation and of the truth relation.

Lemma 3.2.4
Let W be a world description. For a set of partial models $P(W)$ w.r.t. a signature Σ it holds:

1. For any model $M \in P(W)$ and sentence φ it holds

$$M \models_3 \varphi \Rightarrow M \mid\!\models_W \varphi.$$

2. For any two models $M_1, M_2 \in P(W)$ and sentence φ such that $M_1 \leq M_2$ it holds

$$M_1 \mid\!\models_W \varphi \Rightarrow M_2 \mid\!\models_W \varphi.$$

Proof
(1.) holds by definition of refinement;
(2.) if $N \geq M_2$ then $N \geq M_1$ which with $M_1 \mid\!\models_W \varphi$ implies $N \models \varphi$ for arbitrary $N \in W$, hence $M_2 \mid\!\models_W \varphi$. □

From the above lemma and the lemma 3.2.2, which says that $M_1 \leq M_2$ implies $In(M_1) \leq In(M_2)$, the more specialised corollary follows:

Corollary 3.2.5
1. For any model $M \in P(W)$ and sentence φ it holds:

$$In(M) \models_3 \varphi \Rightarrow In(M) \mid\!\models_W \varphi.$$

2. For any models $M \in P(W)$ and $N \in W$, and output sentence φ such that $M \leq N$ it holds:

$$In(M) \mid\!\models_W \varphi \Rightarrow In(N) \mid\!\models_W \varphi.$$

Lemma 3.2.6
For any complete model $N \in W$ and sentence φ it holds:

$$In(N) \mid\!\models_W \varphi \Rightarrow N \models \varphi.$$

Proof
The definition of forcing proves the lemma, because N is the only refinement of $In(N)$. □

3.3 Correctness and Soundness

The first notions we will investigate are correctness and soundness. Both notions are related to the correspondence of the theory to the world description. We will show these notions to be equivalent under some circumstances.

3.3.1 Correctness

A theory is called to be correct if it holds in the world. Here, we are dealing with world descriptions, and therefore, we call a theory correct if it holds in every situation determined by the world description. Because of the compositional framework we consider, input information for every situation have to be given. This view leads to the following definition of correctness:

Definition 3.3.1 (Correctness)
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. S is *correct* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall N \in W \;\; \mathit{In}(N) \cup S \models_3 \varphi \;\Rightarrow\; N \models \varphi.$$

3.3.2 Soundness

Closely related to the notion of correctness is the notion of soundness. While correctness demands a goal to be true in a situation, soundness is related to the forcing relation. We distinguish two kinds of soundness, according to the possible situations.

The first kind of soundness will be called weak soundness. It is called weak, because we only consider complete situation models.

Definition 3.3.2 (Weak Soundness)
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. S is *weakly sound* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall N \in W \;\; \mathit{In}(N) \cup S \models_3 \varphi \;\Rightarrow\; \mathit{In}(N) \mid\!\models_W \varphi.$$

By lemma 3.2.6 we know that $\mathit{In}(N) \mid\!\models_W \varphi$ implies $N \models \varphi$. Therefore, the following lemma holds:

Lemma 3.3.1
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. If S is weakly sound w.r.t. W, then S is correct w.r.t. W.

Because we often deal with partial situations, it is desired to ensure soundness in such cases, too. For this reason, we introduce the notion of strong soundness.

Definition 3.3.3 (Strong Soundness)
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. S is *strongly sound* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall M \in P(W) \;\; \mathit{In}(M) \cup S \models_3 \varphi \;\Rightarrow\; \mathit{In}(M) \mid\!\models_W \varphi.$$

The next lemma follows from the fact that $W \subseteq P(W)$. By this fact we know that if S is strongly sound w.r.t. W, then weak soundness holds for all situations $N \in W$, hence S is weakly sound w.r.t. W.

Lemma 3.3.2
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. If S is strongly sound w.r.t. W, then S is weakly sound w.r.t. W.

The presented definitions gradually refine from the truth relation, over the forcing relation to the use of partial models. Strong soundness seems to be the strongest definition, because it checks all possible partial models and it uses the forcing relation. In fact, the notions of correctness, weak soundness, and strong soundness are equivalent, because the truth relation $\models_3$ is monotonic.

Theorem 3.3.1
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory, then it is equivalent:

1. S is strongly sound w.r.t. W
2. S is weakly sound w.r.t. W
3. S is correct w.r.t. W

Proof
$(1 \rightarrow 2)$. By lemma 3.3.2 (strong completeness implies weak completeness).

$(2 \rightarrow 3)$. By lemma 3.3.1 (weak completeness implies correctness).

$(3 \rightarrow 1)$. Given a partial model $M \in P(W)$ such that $In(M) \cup S \models_3 \varphi$. Consider an arbitrary complete refinement $N \in W$ such that $In(M) \leq N$. By lemma 3.2.2 it holds $In(In(M)) \leq In(N)$, and $In(In(M)) = In(M)$ implies $In(M) \leq In(N)$. From $In(M) \cup S \models_3 \varphi$ and by monotonicity we can conclude $In(N) \cup S \models_3 \varphi$, and by correctness this implies $N \models \varphi$. Because N was taken arbitrary, it follows that for all models $N \in W$, it holds $In(M) \leq N \;\Rightarrow\; N \models \varphi$, and by definition of forcing it holds $In(M) \Vdash_W \varphi$. □

3.4 Consistency

We turn now to the notion of consistency. Though consistency was thoroughly studied in logic, it is not entirely clear what it means to say that a specification or knowledge base is consistent. In this section we study the notion of consistency in more detail. We will start by presenting a general definition of consistency and a derivation of it. Afterwards, we will present some special cases of inconsistency for theories consisting of program formulas.

3.4.1 The Notion of Consistency

Consistency is a notion that stands for the absence of contradictory information. Classically, a logical theory is called to be consistent, iff there exists a model satisfying the theory. In other words, a logical theory T is consistent iff there is no formula A such that $T \vDash A$ and $T \vDash \neg A$ at the same time. This definition also covers T itself, because any formula $A \in T$ satisfies $T \vDash A$, thus if both $A \in T$ and $\neg A \in T$ hold, then the set T is inconsistent.

In practice, this definition is often too restricted and therefore, consistency is defined as a property with respect to a set of situations. Often a situation leading to inconsistency does not occur and therefore, we regard such a theory to be consistent w.r.t. this situation. For example, a simple theory contains only the sentences $a \Rightarrow b$ and $a \Rightarrow \neg b$. This theory is inconsistent if we consider every possible situation but in the case that a will never be established, we call this theory consistent.

Definition 3.4.1 (Consistency)
Let X be any set of models. A universal theory S is called consistent w.r.t. X if there is no model $M \in X$ and formula $\varphi \in \mathbf{Q}$ such that

$$M \cup S \vDash_3 \varphi \text{ and } M \cup S \vDash_3 \neg \varphi.$$

The advantage of this view is the restriction of the domain that needs to be checked. This leads to more efficiency in consistency checking. Of course, problems arise if the same theory will be reused in a domain with different situations. In such cases, consistency has to be proved again.

We will now relate the above definition to the world description.

Lemma 3.4.1
If a universal theory S is weakly sound w.r.t. a world description W, then it is also consistent w.r.t. $In(W)$.

Proof
Suppose S is weakly sound w.r.t. W and not consistent w.r.t. $In(W)$. Then there exists a model $M \in In(W)$ such that $M \cup S \vDash_3 \varphi$ and $M \cup S \vDash_3 \neg \varphi$ hold. Now $In(W) = \{In(N) \mid N \in W\}$, hence there must exist a complete model $N \in W$ such that $In(N) \cup S \vDash_3 \varphi$ and $In(N) \cup S \vDash_3 \neg \varphi$. By weak soundness we get $In(N) \mid\!\vDash_W \varphi$ and $In(N) \mid\!\vDash_W \neg \varphi$ and by lemma 3.2.6 we have $N \vDash \varphi$ and $N \vDash \neg \varphi$. This contradicts the property of N being a model, hence S must be consistent w.r.t. $In(W)$. □

Ambivalence

[Preece et al., 1992] identify inconsistency as a special case of what they call ambivalence. Ambivalence is defined as follows:

> A knowledge base K is *ambivalent* iff for some permissible environment, we can infer an impermissible set of hypotheses.

This means, given some situation and a logical theory, it is possible to infer goals that exclude each other. It is easy to see that inconsistency is a special case of ambivalence. The set of goals that contains a goal and its negation is always impermissible. Thus, if a theory is inconsistent, then it is ambivalent. The reverse must not hold. For example, take the theory that contains $\mathit{student}(x) \Rightarrow \mathit{undergrad}(x)$ and $\mathit{student}(x) \Rightarrow \mathit{postgrad}(x)$. Now this theory is consistent, but given the impermissible set $\{\mathit{postgrad}(x), \mathit{undergrad}(x)\}$ of goals, it is ambivalent.

3.4.2 Special Cases of Inconsistency

Let us assume that a theory S contains only program formulas, i.e. formulas of the form

$$L_1 \wedge \ldots \wedge L_n \rightarrow L,$$

where L_i and L are literals. Each such formula is also called rule, or production rule. If $i = 0$ then L is called a fact.

We will introduce some special cases of inconsistency as they might arise in systems using production rules. Production rule systems are commonly used in expert systems and specification, e.g. [Reichgelt, 1991, pp. 80–114] and [Fütty, 1997], as well as [Leemans et al., 1993] and [Preece et al., 1992]. The examples we will present were taken from the two last named papers.

Contradiction

1. Contradicting fact, e.g. $P(a)$ and $\neg\, P(a)$.
2. Contradicting rule-pair, e.g.
   $P(x) \wedge Q(x) \rightarrow R(x)$ and $P(x) \wedge Q(x) \rightarrow \neg\, R(x)$.
3. Contradicting chains of inference, e.g.
   $P(x) \rightarrow Q(x) \rightarrow \ldots \rightarrow R(x)$ and $P(x) \rightarrow O(x) \rightarrow \ldots \rightarrow \neg\, R(x)$.

Self-contradicting rules

1. Self-contradicting rule, e.g. $P(a) \wedge Q(a) \rightarrow \neg P(a)$.
2. Self-contradicting chain of inference, e.g.
   $P(a) \rightarrow Q(a)$ and $Q(x) \rightarrow \neg P(x)$.

   If $P(a)$ holds, we can infer $\neg P(a)$.

Self-contradicting antecedents

1. Self-contradicting antecedents, e.g. $P(x) \wedge Q(x) \wedge \neg P(x) \rightarrow R(x)$.
2. Self-contradicting antecedents in chains, e.g.
   $\neg P(x) \rightarrow Q(x)$; $Q(x) \wedge P(x) \rightarrow R(x)$.

As a special case consider the following example: $P \wedge Q \rightarrow O$; $P \wedge R \rightarrow \neg O$. Given a situation in which $P, \neg Q$, and R are true simultaneously, then the example is inconsistent. Assuming that there is no such situation, then these rules can coexists in a specification or knowledge base.

3.5 Weak Completeness and Gaps

There can be many reasons for incomplete specifications or knowledge bases. We distinguish three causes for incompleteness: firstly, the engineer simply forgets to include information; secondly, the engineer does not want to decide on an issue; and thirdly, the engineer does not know how to decide on an issue. As a result, specifications or knowledge bases often contain gaps.

Verification of completeness checks whether true facts in the world (given by the world description) are a consequence of the specification or knowledge base. In [Leemans et al., 1993] three different types of completeness are distinguished, weak-completeness, decisiveness (section 3.6.2), and strong completeness (section 3.7.2).

3.5.1 Weak Completeness

A theory is weakly complete, e.g. it contains gaps, if for a given input model a conclusion holds which is not a consequence of the theory. Basically, weak completeness can be defined as a relation between forcing and truth.

Definition 3.5.1 (Weak Completeness)
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. S is *weakly complete* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall N \in W \quad In(N) \Vdash_W \varphi \;\Rightarrow\; In(N) \cup S \vDash_3 \varphi.$$

Weak completeness is the reverse notion of weak soundness. Both notions show the relation of forcing and truth most clearly. The difference is the direction of the implication. In the following sections we will also define the notions of decisiveness (section 3.6.2) and strong completeness (section 3.7.2). Unfortunately, the three notions of completeness are not equivalent but we will show under which additional assumptions this is the case. However, first we turn to some special kinds of weak completeness.

3.5.2 Gaps in Specifications

Again, we will assume that a theory S contains only program formulas and we will call such formulas rules, or facts if there are no literals in the antecedent of a rule. Further, we assume the signature of S to be given explicitly.

[Leemans et al., 1993] identify five cases for indicating gaps, i.e. incomplete information: unnecessary literals, illegal literals, dead-ends, unreachable rules, unreachable literals. [Preece et al., 1992] also deal with these notions, though they put them into two different groups called redundancy and deficiency. We will deal with redundancy as a special case of well-informedness later (section 3.7).

Unnecessary literals

A literal in the signature is unnecessary if it is not part of any antecedent or consequent of any rule. There is one exception: if the unused literal is part of the input and output signature of the specification, then it is still necessary, because it probably passes its information on to the output. Detecting unnecessary literals can be compared to the warning message 'defined but not used' in some programming languages, like Pascal.

Illegal literals

A literal is illegal if it occurs in the specification but not in its signature. We strengthen this definition for two special cases: first, a literal is illegal, if it is part of the antecedent of a rule and not in the input signature or internal signature; and second, it is illegal, if it is part of the consequence of a rule and not in the output signature or internal signature. In both cases, the illegal literal might be in a signature but not in the necessary one. Detecting illegal literals can be compared to the warning message 'variable not declared' in some programming languages, like Pascal. As a side-effect of detecting illegal literals misspelled literals might be found, and in this way a kind of a syntax check is performed as well.

Dead-ends

A dead-end, i.e. a rule with unusable consequent, occurs if a literal appears only in the consequent of a rule but is not declared to be an output literal. For example, if the theory S w.r.t. a signature Σ contains the rules $P(x) \rightarrow Q(X)$ and $R(x) \rightarrow O(x)$, with $P(x), R(x) \in \mathit{InLit}(\Sigma)$, $O(x) \in \mathit{OutLit}(\Sigma)$ but $Q(x) \notin \mathit{OutLit}(\Sigma)$. In such case, $Q(x)$ is called a dead-end, if there is no further rule that contains $Q(x)$ in its antecedent.

Unreachable rules

A rule is unreachable if one of its conditions is not in the consequence of any other rule, and if it is not an input literal. This means, the rule will never be used in an inference. Now there are two reasons for that: first, there is a missing rule to conclude the condition needed; or second, this rule might be redundant, if its consequences are not needed anymore. Detecting unreachable rules does not depend on any input models w.r.t. Σ.

Unreachable literals

A literal is unreachable if it is only part of the consequence of an unreachable rule. Checking for unreachable literals could be merged into the check for unreachable rules, because every unreachable literal is part of an unreachable rule but not vice versa. The difference to an unreachable rule is that a literal might still be reachable through another rule. For example, consider a theory containing only two rules $P(x) \rightarrow R(x)$ and $Q(x) \rightarrow R(x)$, with $P(x) \in \mathit{InLit}(\Sigma)$, $R(x) \in \mathit{OutLit}(\Sigma)$, and $Q(x) \notin \mathit{InLit}(\Sigma)$. The second rule is unreachable but $R(x)$ is still reachable via rule one, therefore, $R(x)$ is a reachable literal, although it is the consequence of an unreachable rule.

3.6 Empirical Foundedness and Decisiveness

On the whole, the verification method discussed in this chapter checks whether or not the specification fits the domain description, or, in other words, whether or not the right conclusions can always be drawn.

3.6.1 Empirical Foundedness

In order to perform these checks, it is important to know whether the world description used is detailed enough. It simply might be the case that the domain description is underspecified and therefore, no specification can be found that derives the right conclusions. This problem will be captured by the notion of empirical foundedness of the world description.

Definition 3.6.1 (Empirical Foundedness)
A domain description W w.r.t. a signature Σ is *empirically founded* if for every goal $\varphi \in \mathbf{Q}$ it holds:

$$\forall N \in W \quad N \models \varphi \;\Rightarrow\; \mathit{In}(N) \mid\!\models_W \varphi.$$

Not only do we want a world description to be empirically founded, we moreover want the actual theory to perform this test. This is expressed by the notion of decisiveness.

3.6.2 Decisiveness

A specification is decisive, if each goal that is true in a model of the domain can be inferred from the input. The notion of decisiveness was first presented by [Treur, 1988] and applied to model-based diagnosis by [Herre, 1993b].

Definition 3.6.2 (Decisiveness)
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. S is *decisive* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall N \in W \quad N \models \varphi \;\Rightarrow\; \mathit{In}(N) \cup S \models_3 \varphi.$$

If S is correct w.r.t. W then an equivalent definition is:

$$\forall N \in W \quad \mathit{In}(N) \cup S \models_3 \varphi \;\vee\; \mathit{In}(N) \cup S \models_3 \neg\, \varphi.$$

Decisiveness is the reverse notion of correctness. Furthermore, decisiveness can be considered as a kind of completeness. If a goal φ is true in a model N, and φ does not hold by the input literals of N and the theory S, then S is incomplete, and probably a proposition needs to be added; or W is too large, which means, it contains a counter example that may be deleted.

The definition of decisiveness parallels the definition of empirical foundedness. Actually, the property that distinguishes decisiveness from weak completeness is empirical foundedness.

Theorem 3.6.1
Let W be a domain description w.r.t. a signature Σ, and let S be a sound theory. The following statements are equivalent:

1. S is decisive w.r.t. W.
2. S is weakly complete w.r.t. W and W is empirically founded.

Proof
$(1 \rightarrow 2)$. Given is a domain description W w.r.t. a signature Σ, a universal theory S that is decisive, and a goal $\varphi \in \mathbf{Q}$. We need to prove weak completeness and empirical foundedness separately.
First, we show weak completeness. Consider an arbitrary situation model $N \in W$ such that $In(N) \mid\models_W \varphi$. Because, for any model $N \in W$ and goal φ it holds $In(N) \mid\models_W \varphi \Rightarrow N \models \varphi$, it follows $N \models \varphi$, and by decisiveness it follows $In(N) \cup S \models_3 \varphi$. Hence, S is weakly complete w.r.t. W.
Now, we show empirical foundedness. Consider an arbitrary situation model $N \in W$ such that $N \models \varphi$. By decisiveness it follows $In(N) \cup S \models_3 \varphi$, and $In(N) \mid\models_W \varphi$ by soundness of S. Hence, W is empirically founded.

$(2 \rightarrow 1)$. Given is a empirically founded domain description W w.r.t. a signature Σ, a universal theory S that is weakly complete w.r.t. W, and a goal $\varphi \in \mathbf{Q}$. Consider an arbitrary situation model $N \in W$ such that $N \models \varphi$. Because W is empirically founded it follows $In(N) \mid\models_W \varphi$, and it follows $In(N) \cup S \models_3 \varphi$, because S is weakly complete. As N was chosen arbitrary, decisiveness is satisfied.
$\square$

3.7 Well-Informedness and Strong Completeness

Naturally, a specification or knowledge base should be as simple as possible. It should not have too many input facts and not too many rules, because this leads to unclear specifications or knowledge bases. For instance, a specification or knowledge base can contain superfluous rules, like redundant or subsumed rules. This does not naturally lead to incorrectness or non-decisiveness. On the other hand, inadequate collections of rules or conditions may lead to undesired behaviour.

3.7.1 Well-Informedness

The intuition behind well-informedness is that a goal must hold due to a (not necessarily unique) smallest input which is necessary to force it. Unlike completeness, well-informedness does not depend on a domain description. Rather it is an inherent property of a theory, which is the reason for an arbitrary set of input models in the following definition.

Definition 3.7.1 (Well-informedness)
A universal theory S is *well-informed* w.r.t. a set of complete input models X, if for any partial model $M \in P(X)$ any goal $\varphi \in \mathbf{Q}$ is true for $M \cup S$ if it is true in all refinements N in X of M. With other words, if for all $M \in P(X)$ it holds:

$$[\forall N \in X \quad M \leq N \ \Rightarrow N \cup S \models_3 \varphi] \ \Rightarrow M \cup S \models_3 \varphi.$$

As already argued, the idea of verification is to compare the relations of consequence and forcing. The forcing relation itself obeys a condition similar to well-informedness:

Theorem 3.7.1
Let W be a domain description w.r.t. a signature Σ. For any partial model $M \in P(In(W))$ and any goal $\varphi \in \mathbf{Q}$ that is forced by the model M is also forced by all refinements $N \in W$ of M. With other words, for all $M \in P(In(W))$ it holds:

$$[\forall N \in W \quad M \leq N \ \Rightarrow \ N \mid\models_W \varphi] \ \Rightarrow \ M \mid\models_W \varphi.$$

Proof
Given is a domain description W w.r.t. a signature Σ, a goal $\varphi \in \mathbf{Q}$, and a partial model $M \in P(In(W))$. Consider an arbitrary complete refinement $M \leq N$ with $N \in W$. By the premise of the theorem it follows that $In(N) \mid\models_W \varphi$. Because for any model $N \in W$ and goal φ it holds $In(N) \mid\models_W \varphi \Rightarrow N \models \varphi$, it follows $N \models \varphi$. As an arbitrary N was chosen it follows for all $N \in W$ that $M \leq N \Rightarrow N \models \varphi$, which is the definition of forcing ($M \mid\models_W \varphi$) and proves the theorem. □

3.7.2 Strong Completeness

A property of well-informedness with respect to verification is that it distinguishes a stronger form of completeness from weak completeness.

Definition 3.7.2 (Strong Completeness)
Let W be a domain description w.r.t. a signature Σ. A universal theory S is *strongly complete* w.r.t. W if for all goals $\varphi \in \mathbf{Q}$ it holds:

$$\forall M \in P(In(W)) \quad M \mid\models_W \varphi \ \Rightarrow \ M \cup S \models_3 \varphi.$$

Strong completeness is the reverse notion of strong soundness. As we already mentioned, the notions of completeness are not equivalent. However, well-informedness (along with monotonicity) is identified as the property that distinguishes strong and weak completeness.

Theorem 3.7.2
Let W be a domain description w.r.t. a signature Σ, and let S be a universal theory. If S is sound w.r.t. W, then the following notions are equivalent:

1. S is strongly complete w.r.t. W.
2. S is weakly complete w.r.t. W and well-informed w.r.t. $\mathit{In}(W)$.

Proof
$(1 \rightarrow 2)$. Given is a domain description W w.r.t. a signature Σ, and a specification S that is strongly complete w.r.t. W. We need to prove weak completeness and well informedness separately.
First, we show weak completeness. Because S is strongly complete, it holds for all partial models $M \in P(\mathit{In}(W))$ the completeness property. Because $W \subseteq P(W)$ it follows that completeness holds for all complete models $N \in \mathit{In}(W)$, hence S is weakly complete.
Now, we show well informedness. Given a partial model $M \in P(\mathit{In}(W))$ and a goal $\varphi \in \mathbf{Q}$ such that for all $N \in W$ it holds $M \leq N \Rightarrow N \cup S \models_3 \varphi$. By soundness it follows for all such N that $N \mid\models_W \varphi$, and by theorem 3.7.1 also that $M \mid\models_W \varphi$. Finally, strong completeness implies $M \cup S \models_3 \varphi$, hence S is well-informed.

$(2 \rightarrow 1)$. Given is a domain description W w.r.t. a signature Σ, a universal theory S that is weakly complete w.r.t. W and well-informed w.r.t. $\mathit{In}(W)$. Further, take a model $M \in P(\mathit{In}(W))$ and a goal $\varphi \in \mathbf{Q}$ such that $M \mid\models_W \varphi$. Consider an arbitrary complete refinement $M \leq N$ with $N \in \mathit{In}(W)$. Now $N = \mathit{In}(N')$ for some $N' \in W$. By corollary 3.2.5 it follows $N \mid\models_W \varphi$. Because S is weakly complete it follows $N \cup S \models_3 \varphi$. As N was chosen arbitrary it follows for all $N \in \mathit{In}(W)$ that $M \leq N \Rightarrow N \cup S \models_3 \varphi$, which implies $M \cup S \models_3 \varphi$ by well-informedness. Therefore, $M \mid\models_W \varphi$ implies $M \cup S \models_3 \varphi$, hence S is strongly complete w.r.t. W. □

3.7.3 Special Cases of Well-Informedness

There are several ways in which a theory might contain too much information. In this subsection we assume that a theory S contains only program formulas where each such formula is also called rule. The examples we are going to present can be found in [Leemans et al., 1993] and [Preece et al., 1992].

Circularity

A set of program formulas has circles if it contains some set of rules such that a loop could occur when the rules are applied. Basically, two kinds of circularity can be distinguished:

1. Self-referent rules, e.g. simply $P(x) \to P(x)$ or $P(a) \wedge Q(x) \to P(x)$. Note that this rule could be useful, as it infers a general conclusion $P(x)$ from a specific instance $P(a)$ and $Q(x)$. Nevertheless, the engineer should be made aware of this.
2. Self-referent chain of inferences, e.g. $P(a) \to Q(a)$ and $Q(x) \to P(x)$.

Redundancy

A set of program formulas in a universal theory are redundant if the same inferences can be made, regardless of the presence or absence of the rules. For example:

1. Redundancy in rule pairs, e.g. duplicate rules, like $P(a) \wedge Q(b) \to R(b)$ and $Q(b) \wedge P(a) \to R(b)$.
2. Redundancy in chained inference, e.g. $P(x) \to Q(x) \to R(x)$ and $P(x) \to R(x)$, where $Q(x)$ is not part of the consequence of any other rule.

In the latter case, the second rule is an inference shortcut, possible included for reasons of run-time efficiency. Such shortcuts may be desirable in practice but the designer should be aware of their existence.

Subsumedness

We say, a program formula subsumes another if it is more general than the other. For example:

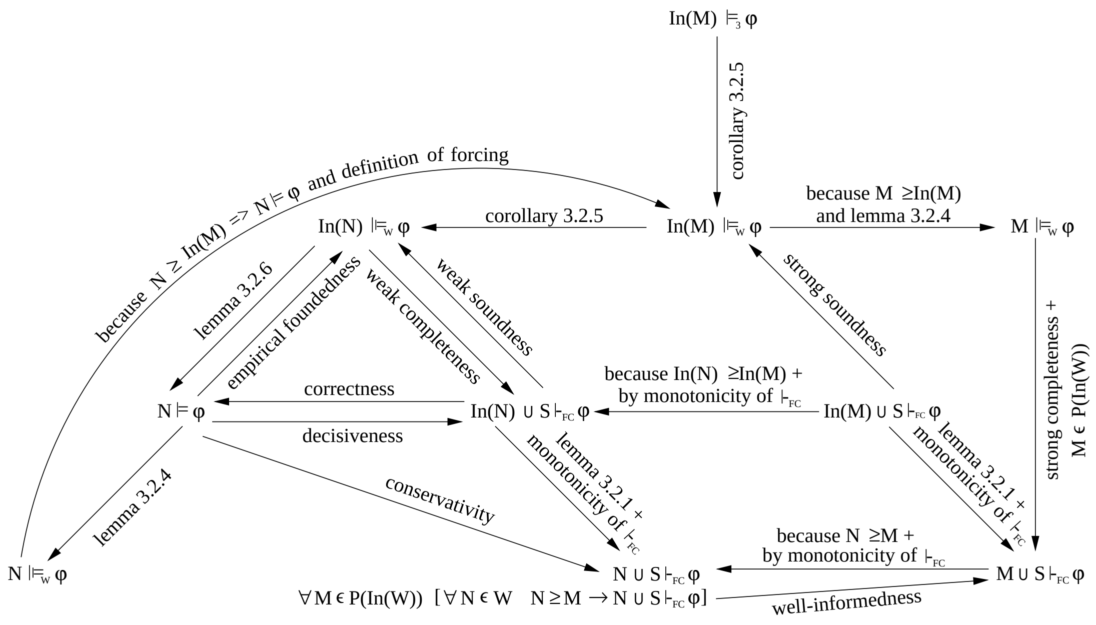
1. Subsumedness in rule pairs, e.g. $P(a) \wedge Q(x) \to R(a)$ is subsumed by $P(x) \to R(x)$.
2. Subsumedness in chained inference, e.g $P(x) \wedge Q(x) \to O(x) \to R(x)$ is subsumed by $P(x) \to R(x)$.

3.8 Conclusion

In the last sections we introduced several properties a logic theory, e.g. a formal specification or a knowledge base, should fulfil. Most of the introduced properties were related to a world description. In contrast to the work by [Leemans et al., 1993], we used a set of goals, containing sentences and existentially quantified sentences. Further, we did not use an inference relation but applied the truth relation in the definitions. Unfortunately, due to the use of the truth relation it is not directly possible to apply a proof tool to check the introduced properties.

However, the truth relation $\models_3$ can be approximated by the inference relation $\vdash_{FC}$. Therefore, all the concepts introduced for $\models_3$ hold also for $\vdash_{FC}$. This means, it possible to check all the presented properties with the help of a proof tool, which is based on the calculus of Forward Chaining.

We will conclude this chapter by giving a pictorial representation of the relationships between the notions of static verification. In the last sections, we used the truth relation for defining properties like correctness and completeness. Note that in figure 3.1 the inference relation $\vdash_{FC}$ is used.



In(M) ⊨₃ φ
corollary 3.2.5
because N ≥ In(M) => N ⊨ φ and definition of forcing
In(N) ⊫w φ
corollary 3.2.5
In(M) ⊫w φ
because M ≥In(M) and lemma 3.2.4
M ⊫w φ
lemma 3.2.6
empirical foundedness
weak completeness
weak soundness
strong soundness
strong completeness + M ∈ P(In(W))
N ⊨ φ
correctness
decisiveness
In(N) ∪ S ⊢FC φ
because In(N) ≥In(M) + by monotonicity of ⊢FC
In(M) ∪ S ⊢FC φ
lemma 3.2.1 + monotonicity of ⊢FC
lemma 3.2.1 + monotonicity of ⊢FC
lemma 3.2.4
conservativity
N ⊫w φ
N ∪ S ⊢FC φ
because N ≥M + by monotonicity of ⊢FC
M ∪ S ⊢FC φ
∀M ∈ P(In(W)) [∀N ∈ W N ≥ M → N ∪ S ⊢FC φ]
well-informedness

Figure 3.1: Relationships between the Notions of Static Verification - consider W to be a world description, $N \in W$ and $M \in P(W)$ with $M \leq N$, $\varphi \in \mathbf{Q}$, and S to be a universal theory

Chapter 4

A Specification and Verification Case Study

In chapter 2 we proposed to use tools, methods, and results from knowledge engineering for specification engineering, and in chapter 3 we presented formally several properties a specification should fulfil. Now we are going to demonstrate our ideas on a practical example.

First, we will introduce DESIRE, which stands for 'framework for DEsign and Specification of Interacting REasoning modules'. DESIRE was developed by the Artificial Intelligence Group at the Vrije Universiteit Amsterdam under the leadership of Prof. Jan Treur. It was designed for the specification of knowledge based systems and is now used for the specification of multi-agent systems. The framework DESIRE is still under development.

The current design environment based on DESIRE contains a formal syntax, graphical editors, an implementation generator, and an execution manager. Information are stored in several knowledge bases, including some for signatures, which are comparable to dictionaries. Therefore, the framework DESIRE corresponds to a knowledge representation system as it was introduced in figure 2.1.

Afterwards, we are going to construct a formal specification of a simple telecommunication network. A similar example was used by [Fütty, 1997] in her diploma thesis, which was also supervised by Dr. Peter Hrandek. Telecommunication examples are often used to demonstrate the applicability of formal methods. This is mainly due to the fact that everyone uses telecommunication facilities. Further, the telecommunication world is distributed and all the players in it are independent of each other, hence it is a multi-agent system.

Finally, we will check whether the knowledge bases used in the specification possess certain properties. This will be done according to the notions defined in chapter 3. The result of this chapter will be a verified specification of a simple telecommunication network.

4.1 DESIRE

DESIRE, which stands for 'framework for DEsign and Specification of Interacting REasoning modules' is both a formal specification framework and, together with its development environment, a knowledge representation system. It was developed at the Vrije Universiteit Amsterdam, aiming at the specification of compositional architectures for knowledge based reasoning systems [Treur and Wetter, 1993] and later applied to the specification of multi-agent systems [Brazier et al., 1997]. In order to support DESIRE, a graphical editor and an implementation generator were incorporated into its development environment. For a comparison of DESIRE with seven other formal specification methods for complex reasoning systems see the book by [Treur and Wetter, 1993]. For an extensive introduction to DESIRE, we recommend the course materials by [Brazier et al., 1996a], [Brazier et al., 1996b], and [Brazier et al., 1997] used at the Vrije Universiteit Amsterdam in the past years.

4.1.1 DESIRE - A Specification Framework for Compositional Systems

DESIRE supports the specification of compositional systems by modelling and specification of knowledge of

- task (de)composition,
- information exchange,
- control (de)composition, e.g. sequencing of tasks,
- task delegation, and
- knowledge (de)composition.

In the following subsections we will discuss these types of knowledge in more detail.

4.1.2 Task Composition

A commonly accepted approach to construct complex systems is the top-down strategy. Here, a complex task will be decomposed until atomar tasks are reached. A task hierarchy defines distinguishable subtasks of the task and the task-subtask relations between them. It is possible to make the task hierarchy explicit, for instance, by depicting it as a tree structure or as a box-in-box structure.

In DESIRE each task will be mapped onto one component. Within DESIRE two types of components, i.e. tasks, can be distinguished: primitive components and composed components. Whereas a primitive component is related to an atomar task, the functionality of a composed component is determined by its sub-components. Again, each of these components can either be primitive or composed, and so on.

For each task the information required as input or produced as output has to be specified. This can be done by defining the input and output information type of a component, also called the signature of a component. Information types can also be composed. The signatures are defined in a predicate logic with a hierarchically ordered sort structure, i.e. order-sorted predicate logic. Units of information are represented by the ground atoms defined in the signature [Engelfriet and Treur, 1997].

Another important element of task decomposition that is emphasised in DESIRE is the reflective nature of tasks. This means, it is essential to distinguish between object-level reasoning about a domain, and meta-level reasoning about the state and goals of a system. This object-meta distinction between tasks can be specified explicitly as an object-meta relation [Brazier et al., 1994].

4.1.3 Information Exchange

Knowledge of information exchange defines which types of information can be transferred between components and the information links by which this can be achieved [Brazier et al., 1997]. Figure 4.1 shows the various types of links that are possible within composed components:

- mediating links,
- private links, and
- task control links (upwards and downwards).

Mediating links and private links together are called information links. Basically, an information link relates truth values of ground atoms of one component to truth values of another component [Brazier et al., 1997]. It is possible to use only partial truth tables for the transformation. All information links have to be named. Furthermore, the information types a link connects and the roles they play within a component also have to be specified.

Mediating Links

Links interacting with the interface of the parent component are called mediating links. Mediating links transfer information from the input interface of a parent component to the input interface of its subcomponents, or to the output interface

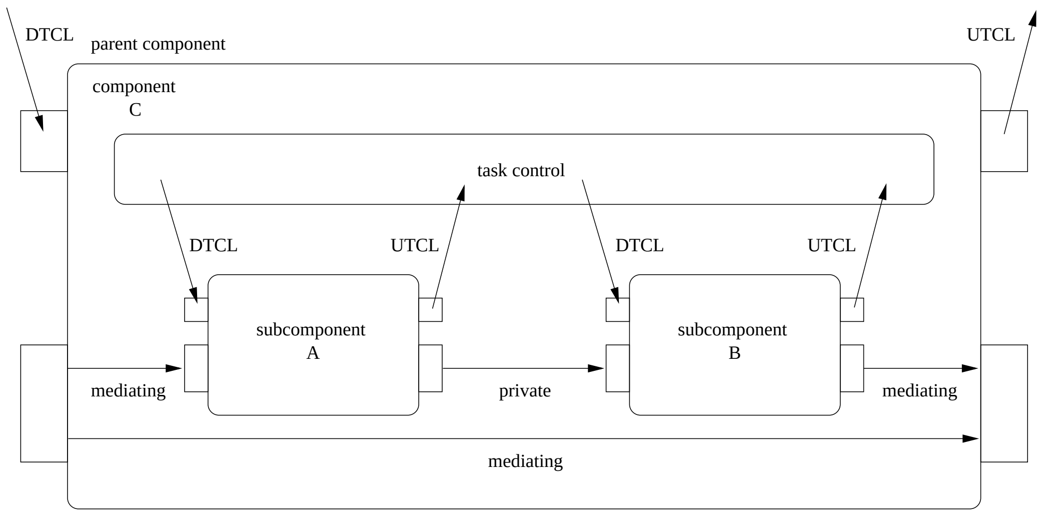


Figure 4.1: Links within a DESIRE Component

of the parent component. They also transfer information from an output interface of a subcomponent to the output interface of the parent component. In figure 4.1 possible mediating links are depicted.

Private Links

Figure 4.1 also shows a private link. Private links transfer information from the output interface of one of the subcomponents (A) to the input interface of one of the subcomponents (B). It is possible to connect the input interface to the output interface of just one component.

Task Control Links

Task control links carry the task control information a component needs. Each component transfers to and receives from its subcomponents task control information. Two different types of task control links can be identified: an Upward Task Control Link (UTCL) and a Downward Task Control Link (DTCL). Apart from the task control links within a component, there are two control links connected to the component itself, a DTCL on its task control input and an UTCL on its task control output.

In figure 4.1 task control links (DTCL and UTCL) are shown between the subcomponents (A and B) of a component (C) and its task control, as well as from the component's task control to its parent task control. Task control links are built-in within DESIRE, this means, they do not have to be specified explicitly.

4.1.4 Task Control

Task control knowledge defines temporal relations between tasks and information transfer. It specifies which tasks have to be activated under which conditions, as well as the conditions for the information flow. Task control knowledge includes different kinds of knowledge. On the one hand, this is knowledge about task activation, i.e. knowledge about when and how a task should be activated. On the other hand, it also includes knowledge of the goals of a task and the extent to which goals should be derived (see table 4.1).

extent	to be derived
all p	all possible targets
every	every target
any	any target
any new	any target not previously derived

Table 4.1: Derivation Extents within DESIRE

The result of a task is determined by the evaluation of a component's success and/or failure to derive its goals within the specified extent. These results form the precondition for another component activation. In DESIRE it is possible to specify the activation of components and links as sequential or parallel processes.

4.1.5 Task Delegation

In a complex system different participants, like users and/or autonomous systems, interact with each other. This interaction is often needed to achieve the desired system behaviour. For instance, a user can give information to the system. Task delegation basically deals with the problem which task should be performed by whom.

The process of task delegation is defined by a set of participants and a relation between tasks and subsets of the set of participants, for an example see table 4.2:

task	participant
World_State_Management	System
World_State_Aquisition	User
Monitor	System

Table 4.2: A Task Delegation Example

4.1.6 Knowledge Structures

Knowledge of knowledge structures (knowledge (de)composition) includes the specification of types of knowledge that are needed for a task performance. This contains knowledge of input and output information structures (see subsection 4.1.2), knowledge of internal knowledge bases for reasoning components, and knowledge of other types of specifications externally represented, for instance, a neural network, a database, a calculation module, or an algorithm.

The contents of a knowledge base consists of general facts and rules. A general fact is a literal, i.e. an atom or a negated atom. A rule is built of a list of literals as antecedent and a list of literals as consequent. If a rule contains variables, these are assumed to be universally quantified over the whole formula. Any statement from many-sorted predicate logic can be transformed into a set of rules, which are, in some sense, equivalent to the original statement [Brazier et al., 1997]. In DESIRE chaining is used for inferring new information from general facts and rules.

In DESIRE it is also possible to include alternative specifications, like a database or an algorithm. The associated component is called a conventional component. The only restriction on the communication between conventional components and reasoning components is given by the input and output format DESIRE requires.

4.1.7 The DESIRE Development Environment

The framework DESIRE is supported by a software environment consisting of a graphical editor, an implementation generator, and a general manager system.

A special feature of the graphical editor, called Destool, is that it supports the specification of complex systems by a hierarchical representation of components. In that way it is possible to restrict the user's view of components to only one level. All components are represented with their input and output interfaces. Furthermore, information links are depicted as arrows. Information of a component or link can be obtained by simply clicking on it. The implementation generator and execution tools can be activated from within Destool.

An important part of the development environment is the implementation generator. First it checks the syntax of the specification, carries out some semantic checks, and, if no errors occurred, it creates a prototype by translating the DESIRE specification into PROLOG [Kowalski, 1979; Bratko, 1990]). This translation is supported by the declarative programming paradigm in both languages. The generated prototype may be executed by the general manager system.

Actually, there exist two types of the general manager system. One (tgm) that executes the prototype only in a terminal window, and another (xtgm) that interacts with Destool to view the execution steps by highlighting the components or links that are active. Both allow the user to interact with the manager. In that way it is possible to debug the prototype by printing several kinds of information about a component or link. Furthermore, a run trace is generated to keep track of the execution steps.

4.2 A Telecommunication Example - Informal Introduction

In this section we will introduce a simple example of a telephone world which consists of a set of users and one exchange, i.e. a system that connects a group of telephones and provides the necessary facilities for making calls.

Examples from telecommunications are often used to illustrate the application of formal specification, e.g. by [Fütty, 1997], by [Holyer, 1991, pp. 110–115], and by [Kleuker, 1995]. This is, in our view, motivated by the following two facts: first, everyone concerned is familiar with the domain, i.e. knows how to use a telephone. Second, due to the fact that the telecommunication market is a fast growing business place, the customer will choose only those providers who can ensure the correct functioning of their systems.

The example we will present is far too simple to be a real world example in terms of today's needs. However, because of its simplicity it gives a good introduction to the use of methods and tools. In this section we will introduce our example informally, and in the next section we will present an example of a formal specification of the telephone world.

4.2.1 The Environment

We consider a simple telecommunication network. This network will consist of a number of users and one exchange. In contrast to the example given by [Fütty, 1997], we will not consider an internal exchange as it is used within companies but a simple external exchange to which the users are directly connected. In practice, users are connected by at least two wires, where the wires are named 'a' and 'b'. Figure 4.2 shows such a simple telecommunication network.

4.2.2 The Telephone

Nearly everyone living in an industrial society knows a telephone. However, because of the great variety of today's phones, we will shortly present the necessary components of the phone we consider. The telephone used for this example is fairly simple and consists of:

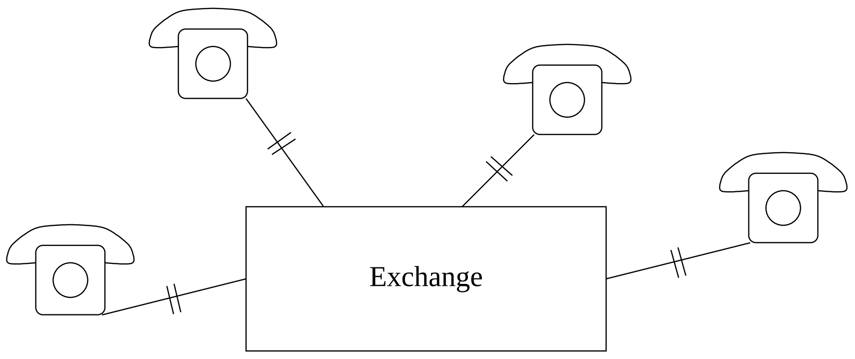


Figure 4.2: A Simple Telecommunication Network

- a receiver, including a speaker and a microphone;
- a bell, which is within the phone;
- a number block, with keys signed with digits from 0 to 9.

After giving the components of the phone, we have to introduce the possible actions associated with each component. First, it is possible to lift or to put down the receiver. Further, one can listen to signals via the speaker and talk to others via the microphone. In our specification we will abstract from speaker and microphone and just consider listening and talking. Second, the bell is able to ring. Third, one can press and release the keys of the number block to dial a number. Again, we consider only an abstract representation of this action. We are only interested in the fact that a number was dialled.

4.2.3 The Customer

The customer is the operator of the phone, hence he will also be called user. Initially, there are two possible tasks:

1. the customer wants to make a call; or
2. the customer receives a call.

In the latter case, the user hears the ringing of the telephone bell. Then the user decides whether to take the call or not. If he takes the call, then he has to lift the receiver and subsequently the user can talk with the caller.

The caller is the customer who makes a call. This task starts by lifting the receiver. Afterwards, the user listens to the signal he receives. First, it might be the case that there is no signal. This indicates an error of the network or a malfunction of the telephone. The user cannot know what happened but he has to put down the receiver. If there is a signal, this might either be an engaged tone or a dialling tone. An engaged tone is a repeated single note and a dialling tone is a continuous sound, either purring or high pitched. In case the user hears an engaged tone, he has to put down the receiver.

If the caller hears a dialling tone, then he dials a phone number. We consider the number to be sent at once. Then the user listens to a second signal. In case there is no signal or the signal is an engaged tone, he has to put down the receiver. However, if the signal is a ringing tone, he has to wait for an answer.

In case there is no answer, he puts down the receiver. If there is an answer both can talk with each other. The talk will be finished if one of both customers puts down the receiver, which in turn is a sign to the other customer to put down the receiver, too. Finally, it is possible to put down the receiver at any time and to finish the task.

4.2.4 The Exchange

The telephone exchange connects the users. When a user lifts the receiver, the exchange looks for a free outgoing line. If there are no lines, it sends an engaged tone to the user, otherwise a dialling tone will be transmitted. Next, the caller sends a phone number. Now the exchange will activate many switches to establish a path to the called user. If there are at any point no lines available, an engaged tone will be send to the caller. This is also the case, if the called user's receiver is already lifted . If not, the exchange sends a signal to the bell of the called user and a ringing tone to the caller. When the called user lifts his receiver, the connection is established. At this time, the bell stops ringing and there will be no ringing tone in the line anymore.

In practice, an exchange is far more complicated and usually more then one exchange station is needed to establish a call. Here, we will abstract from a call that is routed through the exchange station.

4.3 Detailed Specification of the Telecommunication Example using DESIRE

In this section we will give the detailed specification of the telecommunication example. We developed the specification using DESIRE and its development environment, including Destool. Both DESIRE and its development environment were introduced in section 4.1. The complete textual specification can be found in appendix A.

4.3.1 Task Composition

Basically, we distinguish two subtasks. One is called `User` and does what the customer does, and the other is called `Exchange` and performs the tasks of an exchange. Both `User` and `Exchange` are primitive components. Figure 4.3 shows the components as they are created within Destool.

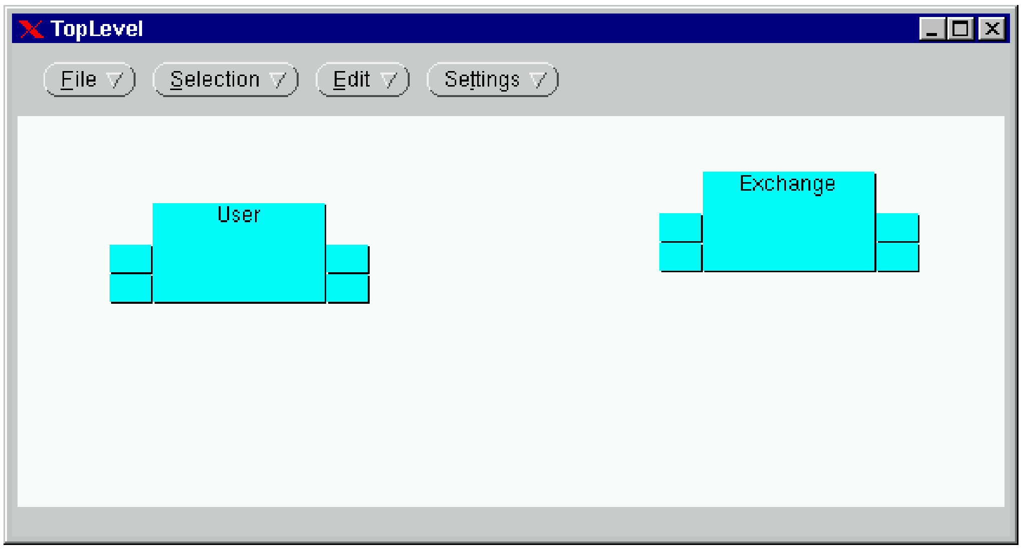


Figure 4.3: Task Composition in Destool

4.3.2 Information Types

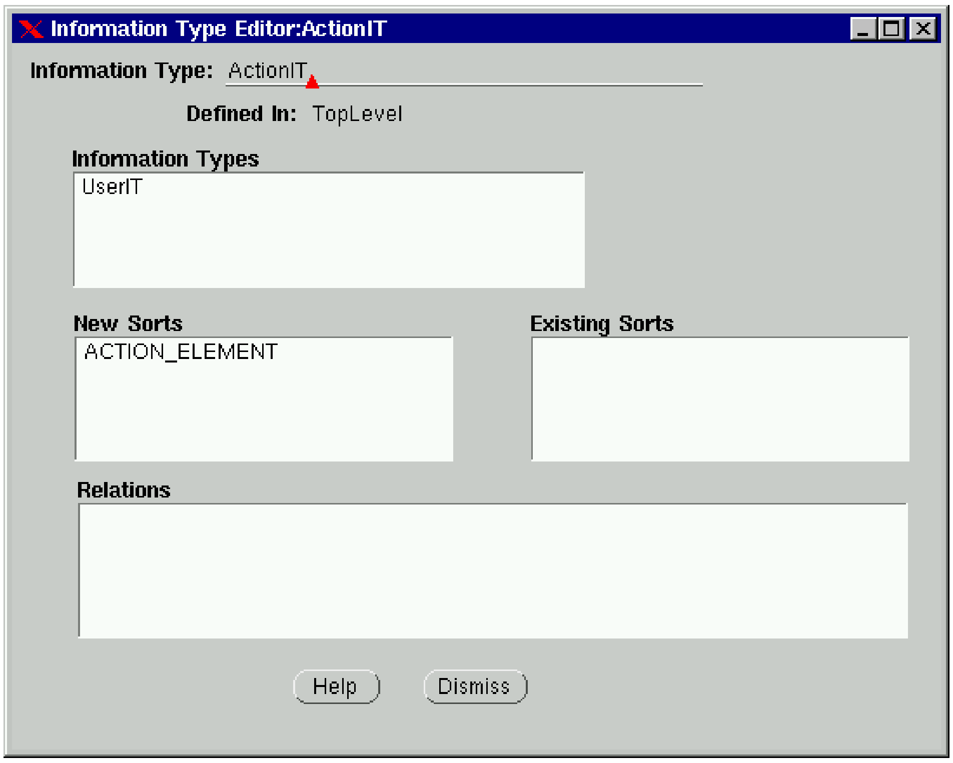
We distinguish two basic types of information: pieces of information and actions. The customer receives pieces of information and performs actions. The exchange receives these actions and provides pieces of information, like signals.

We use `User_In` and `User_Out`, as well as `Exchange_In` and `Exchange_Out`, as the object input, respectively output, information types. These information types are composed and consist of the information types `UserIT` and `SignalIT`, as well as `InfoIT` or `ActionIT` respectively. Further, each user has an additional information type `User_Internal` for internal information, including `User_Input_Information` representing a kind of mental state of a user. The distinction between `In`, `Out`, and `Internal` information types was influenced by the ideas introduced in chapter 3.

Composition of Information Types

Often information types are composed. This improves modularity and makes specification easier. For the specification of the information types we used the concepts of `sorts`, `objects`, `relations`, and `functions`. The information type `UserIT` consists of the sort `USER` and includes the possible users as objects.

According to the possible signals that can be given by an exchange, we distinguish two sorts of the information type `SignalIT`: `SIGNAL_ONE_SORT` and `SIGNAL_TWO_SORT`. `SIGNAL_ONE_SORT` is related to the signals that can occur when the user lifts the receiver, and consists of the objects `dialling_tone` and `engaged_tone`. The second sort consists of the objects `ringing_tone` and `engaged_tone` and is related to the signals after a user dialled a number.


Information Type Editor:ActionIT

Information Type: ActionIT

Defined In: TopLevel

Information Types

UserIT

New Sorts

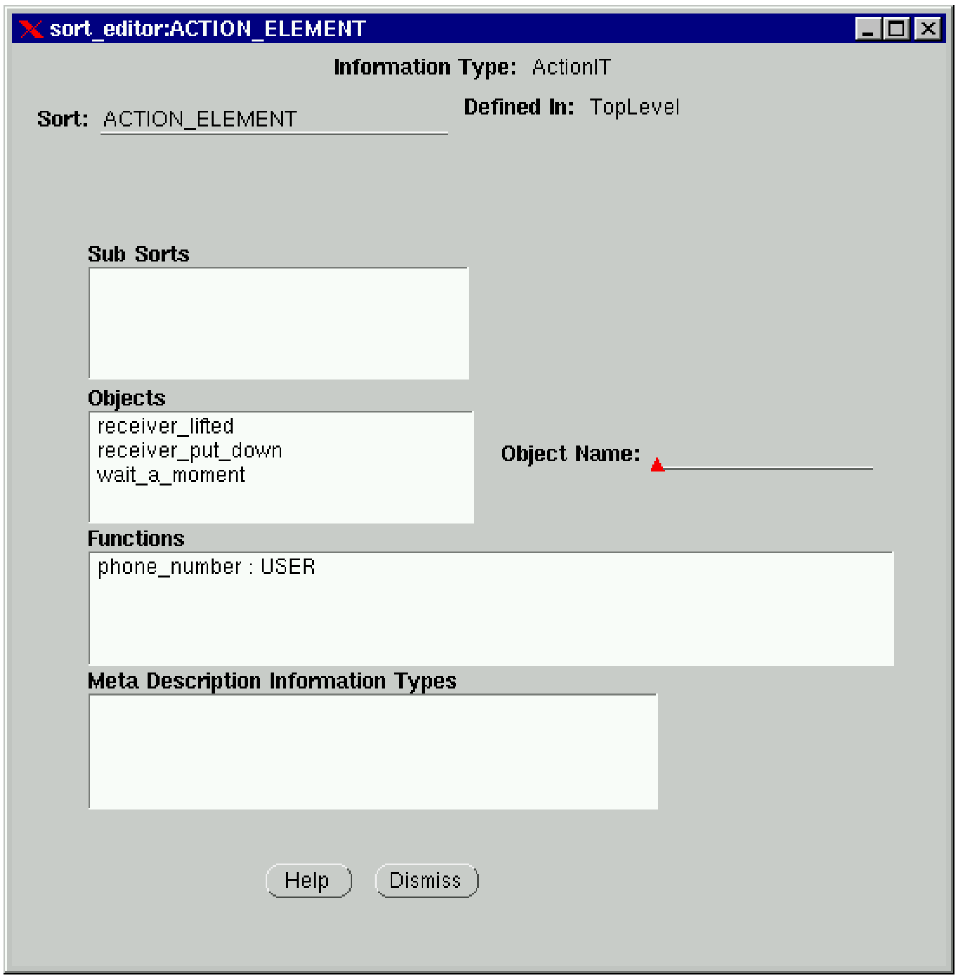
ACTION_ELEMENT

Existing Sorts

Relations

Figure 4.4: Specifying Information Types - the Information Type Editor

Figure 4.4 shows the specification of the information type `ActionIT` within the information type editor of Destool. `ActionIT` incorporates the information type `User_IT` to use the sort `USER`, i.e. to be able to specify functions over `USER`. Further, the sort `ACTION_ELEMENT` is given in `ActionIT`. Figure 4.5 shows the specification of the sort `ACTION_ELEMENT` within the sort editor. Possible actions are: to lift the receiver, to put down the receiver, to wait, and to dial a phone number, where `phone_number` is a function that takes a `USER` and returns an `ACTION_ELEMENT`.


sort_editor:ACTION_ELEMENT

Information Type: ActionIT

Sort: ACTION_ELEMENT

Defined In: TopLevel

Sub Sorts

Objects

receiver_lifted
receiver_put_down
wait_a_moment

Object Name:

Functions

phone_number : USER

Meta Description Information Types

Figure 4.5: Specifying Information Types - the Sort Editor

The information type `InfoIT` consists of the information types `UserIT` and `SignalIT`, as well as the sort `INFO_ELEMENT`. This in turn consists of the object `bell`, to indicate that the bell rings, and the functions `signal1`, `signal2`, and `connection_established`. The first two functions inform the user about the transmitted signals, and the latter if the connection to another user was established.

Finally, `UserIn` and `ExchangeOut` have `InfoIT` as their sub-information type, and `UserOut` and `ExchangeIn` have `ActionIT` as their sub-information type. Additionally, the relations `send` and `received` over `InfoIT` respectively `ActionIT` are defined according to the component.

4.3.3 Information Exchange

In order to specify knowledge of information exchange between processes, we have to give the relations between the output information type of one process and the information type of another process. The relations according to our example are shown in table 4.3, as well as in figure 4.6.

information link	from process	output information type	to process	input information type
`a`	`User`	`UserOut (ActionIT)`	`Exchange`	`ExchangeIn (ActionIT)`
`b`	`Exchange`	`ExchangeOut (InfoIT)`	`User`	`UserIn (InfoIT)`

Table 4.3: Specification of the Information Exchange between the Components `User` and `Exchange`

Note that the figure 4.6 represents less information than the table 4.3: the information types are not shown in figure 4.6.

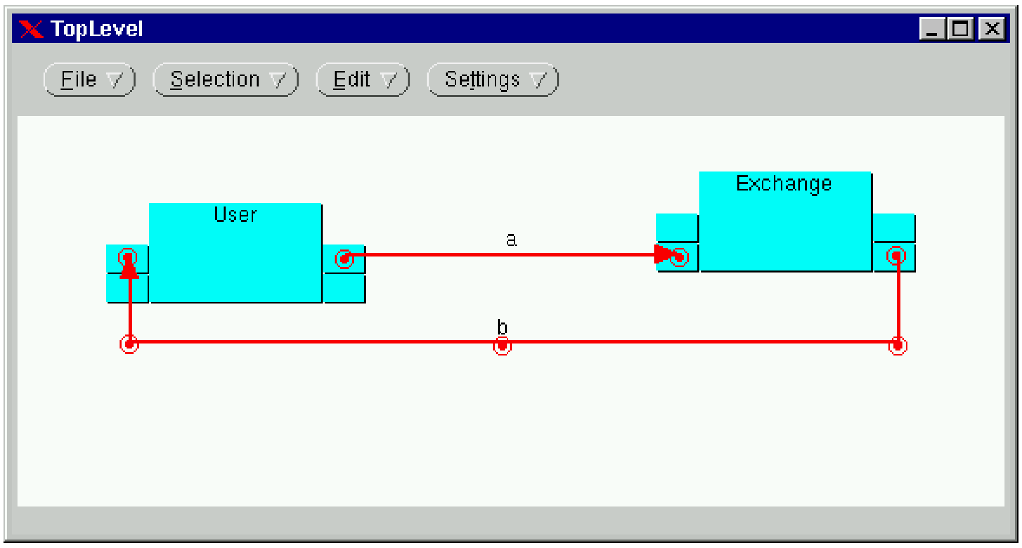
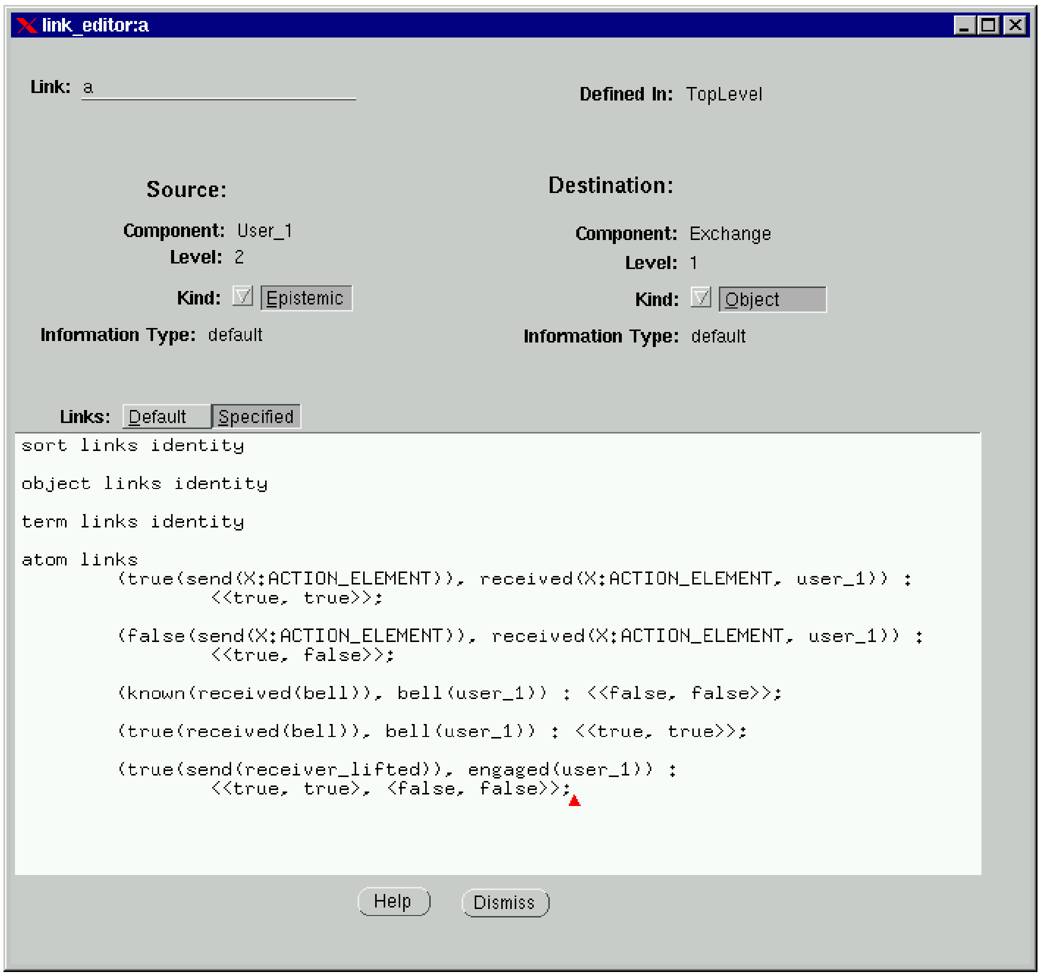


Figure 4.6: Information Exchange within Destool

Link 'a'

Figure 4.7 depicts the specification of the information link from the user component to the exchange component. Beside the fact that an information link forwards information, it is also possible to transform pieces of information. If the user makes an action, then the exchange will receive it, and if the user does not make an action, then this will also be received by the exchange.

Additionally, link a serves three further purposes. First, it provides an initial information of the status of the bell. If it is not known that the bell of a user's phone rings, then it is assumed that the bell of this user's phone does not ring. Second, if the bell of a user's phone rings, then it is explicitly recognised by the exchange. Finally, if a user lifts the receiver, then the exchange knows that the user is now engaged. If the user does not lift the receiver, then it is assumed that the user is not engaged. This gives also in initial information about the engaged status of a user. Especially the information of the bell will be needed when specifying the knowledge base of the exchange.



Link:	a	Defined In:	TopLevel
Source:		Destination:	
Component:	User_1	Component:	Exchange
Level:	2	Level:	1
Kind:	Epistemic	Kind:	Object
Information Type:	default	Information Type:	default

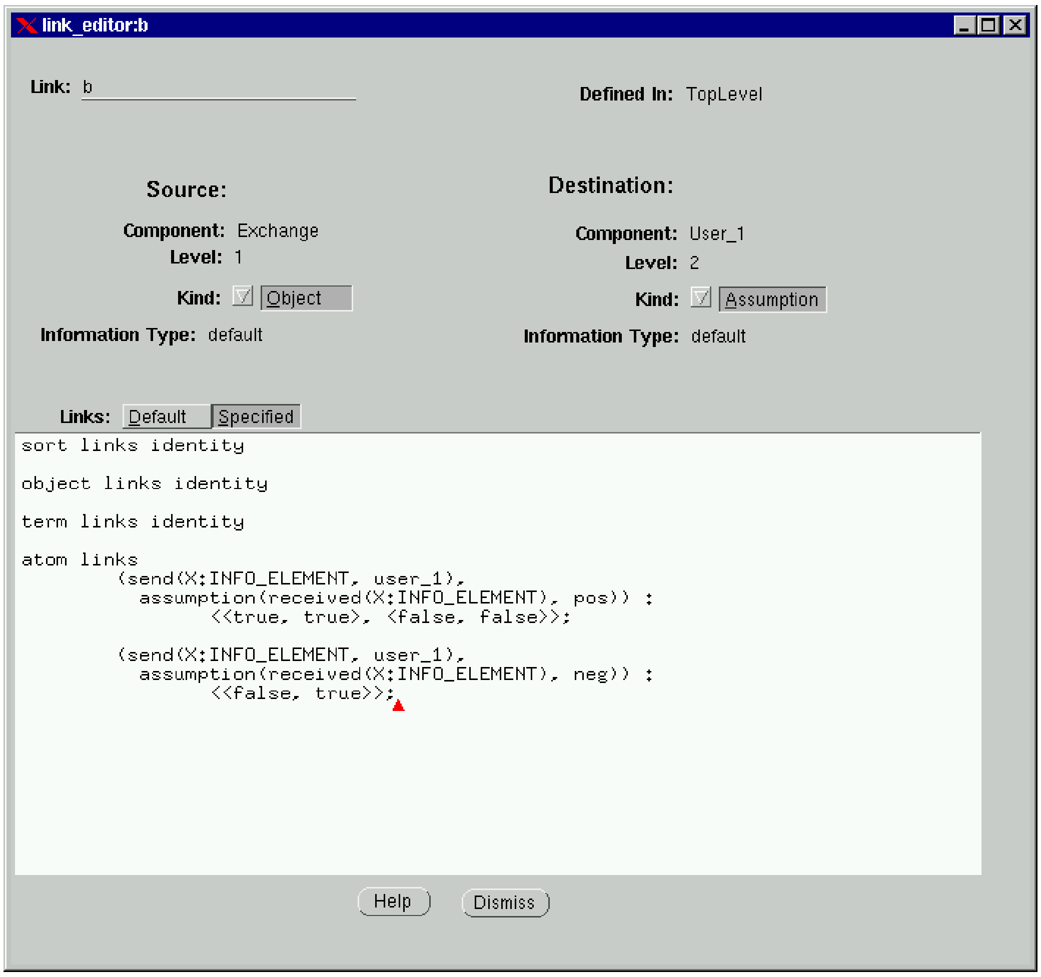
Links: Default [✓ Specified]

```
sort links identity

object links identity

term links identity

atom links
        (true(send(X:ACTION_ELEMENT)), received(X:ACTION_ELEMENT, user_1)) :
                <<true, true>>;

        (false(send(X:ACTION_ELEMENT)), received(X:ACTION_ELEMENT, user_1)) :
                <<true, false>>;

        (known(received(bell)), bell(user_1)) : <<false, false>>;

        (true(received(bell)), bell(user_1)) : <<true, true>>;

        (true(send(receiver_lifted)), engaged(user_1)) :
                <<true, true>, <false, false>>;
```

Figure 4.7: Specification of Link a within Destool

Link ‘b’

The specification of link `b`, connecting the exchange with the user, is much simpler than the specification of link `a`. First, all positive information that are given by the exchange will be transfered to the user, and second, all negative information will be given to the user. The picture of the specification within Destool is given by figure 4.8.



Link: b

Defined In: TopLevel

Source:

Component: Exchange
Level: 1
Kind: Object
Information Type: default

Destination:

Component: User_1
Level: 2
Kind: Assumption
Information Type: default

Links: Default Specified

```
sort links identity

object links identity

term links identity

atom links
        (send(X:INFO_ELEMENT, user_1),
          assumption(received(X:INFO_ELEMENT), pos)) :
                <<true, true>, <false, false>>;

        (send(X:INFO_ELEMENT, user_1),
          assumption(received(X:INFO_ELEMENT), neg)) :
                <<false, true>>;
```

Figure 4.8: Specification of Link `b` within Destool

Task Control

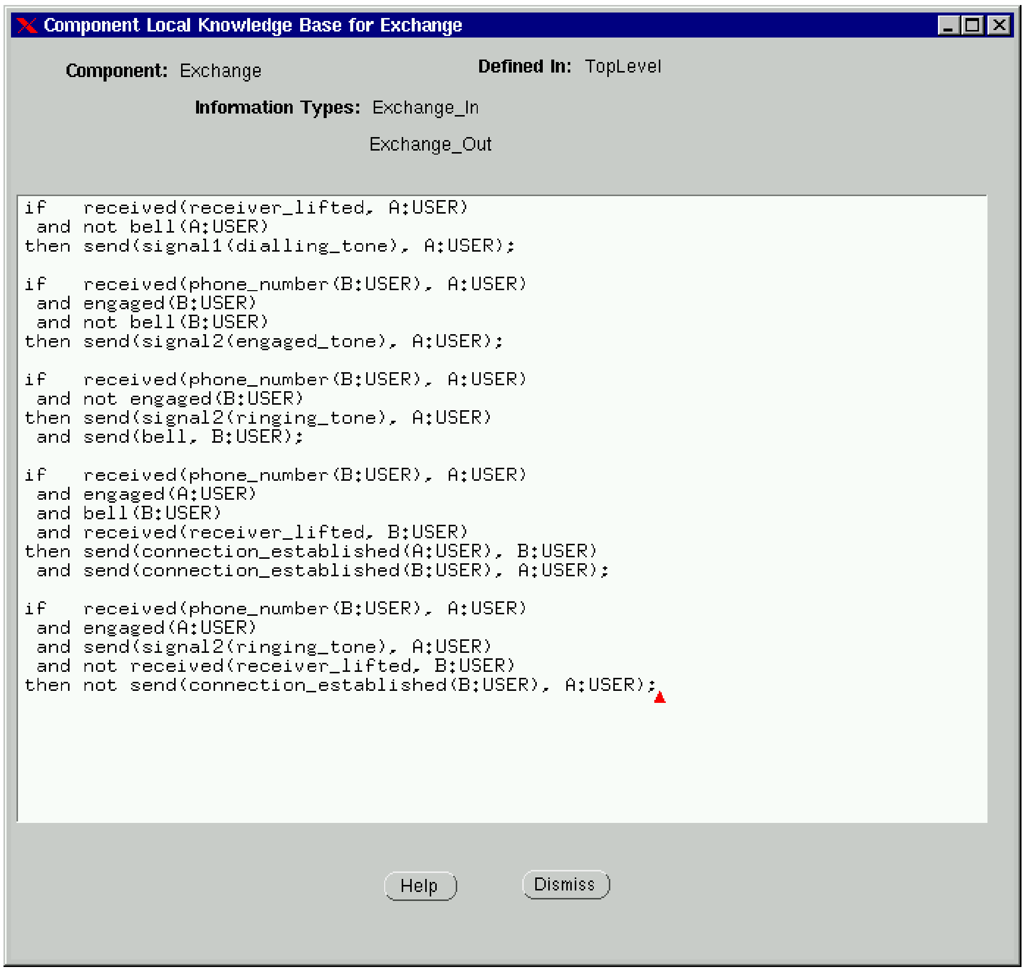
The specification of the task control is fairly simple. All components and links are made awoke after the entire process starts. The whole process stops if there is nothing to do anymore, i.e. if all users put down their receivers.

4.3.4 Knowledge Structures

Finally, we turn to one of the most important parts of the specification, the knowledge bases of the user component and the exchange component. Basically, the knowledge base of a component determines the behaviour of that component. We are most interested in the knowledge bases, since our theory of verification is limited to the static properties of a specification.

The Knowledge Base of the Component 'Exchange'

First, we will look at the specification of the knowledge of the exchange component. Figure 4.9 pictures the knowledge base of the exchange component as it is given within Destool.



Component: Exchange **Defined In:** TopLevel

Information Types: Exchange_In
Exchange_Out

```
if   received(receiver_lifted, A:USER)
 and not bell(A:USER)
then send(signal1(dialling_tone), A:USER);

if   received(phone_number(B:USER), A:USER)
 and engaged(B:USER)
 and not bell(B:USER)
then send(signal2(engaged_tone), A:USER);

if   received(phone_number(B:USER), A:USER)
 and not engaged(B:USER)
then send(signal2(ringing_tone), A:USER)
 and send(bell, B:USER);

if   received(phone_number(B:USER), A:USER)
 and engaged(A:USER)
 and bell(B:USER)
 and received(receiver_lifted, B:USER)
then send(connection_established(A:USER), B:USER)
 and send(connection_established(B:USER), A:USER);

if   received(phone_number(B:USER), A:USER)
 and engaged(A:USER)
 and send(signal2(ringing_tone), A:USER)
 and not received(receiver_lifted, B:USER)
then not send(connection_established(B:USER), A:USER);
```

Figure 4.9: The Knowledge Base of the Component `Exchange` within Destool

The first rule indicates that the exchange should send a dialling tone to the user that lifts the receiver and has no ringing bell, i.e. was not called. Second, if the exchange receives a phone number of a user and the other user is engaged, and therefore, there is no bell ringing, then it sends an engaged tone to the caller.

Third, if the called user is not engaged, then send a signal to the bell of the called user, as well as a ringing tone to the caller. Now, if there is a caller and the user that was called lifts the receiver, a connection between both is established, which will be notified to both users. Finally, if the called user does not lift his receiver, then this will also be transmitted to the caller.

There is a little trick here. If one user receives a bell signal, then the exchange will be informed about this additionally via the information link `a`, and sets `bell(A:USER)` to true. This in turn prevents the exchange for submitting an engaged tone to the caller, if the called user lifts the receiver. In this way, we modelled an exclusive or.

The Knowledge Base of the Component 'User'

Now we turn to the specification of the user. In contrast to the specification of the exchange component, the knowledge base of the user component is given as textual specification.

```
connection_wanted(user_1);
not take_call;

if   received(bell)
 and take_call
then send(receiver_lifted);

if   received(bell)
 and not take_call
then not send(receiver_lifted);

if   talk(A:USER)
then send(receiver_put_down);

if   received(signal1(engaged_tone))
then send(receiver_put_down);

if   received(signal2(engaged_tone))
then send(receiver_put_down);

if   connection_wanted(A:USER)
then send(receiver_lifted);
```

```
if   not received(signal1(X:SIGNAL_ONE_SORT))
then send(receiver_put_down);

if   received(signal1(dialling_tone))
 and connection_wanted(A:USER)
then send(phone_number(A:USER));

if   not received(signal2(X:SIGNAL_TWO_SORT))
then send(receiver_put_down);

if   received(signal2(ringing_tone))
then send(wait_a_moment);

if   received(connection_established(A:USER))
then talk(A:USER);

if   not received(connection_established(A:USER))
then send(receiver_put_down);
```

First, there is the fact `connection_wanted(user_1)`, which states that this user wants to make a call to another user named `User_1`. Second, if this user is called, he will not take the call, i.e. will not lift the receiver. In order to take a call, the `not` needs to be removed from this rule. Further, the receiver will be put down after talking to another user, or if an engaged tone was received.

The next rule states that whenever a user wants to make a call, he has to lift the receiver. Now, the user awaits a signal. If there is none, then just put down the receiver. In case the first signal is a dialling tone, then submit the number of the user the call should be made with, e.g. in our example this will be `User_1`.

Now, a second signal is expected. If there is none, then put down the receiver, and if it is a ringing tone, then wait for an answer. If the other user lifts his receiver, the connection will be established and both can talk. Otherwise, the caller will put down the receiver.

The Entire System

Finally, a picture of the whole telecommunication example is given by figure 4.10. We use four users and one exchange station. `User_2` is called by `User_4` and will not answer the phone call, and `User_3` will call `User_1` and receives an answer. The textual specification of this example can be found in appendix A.

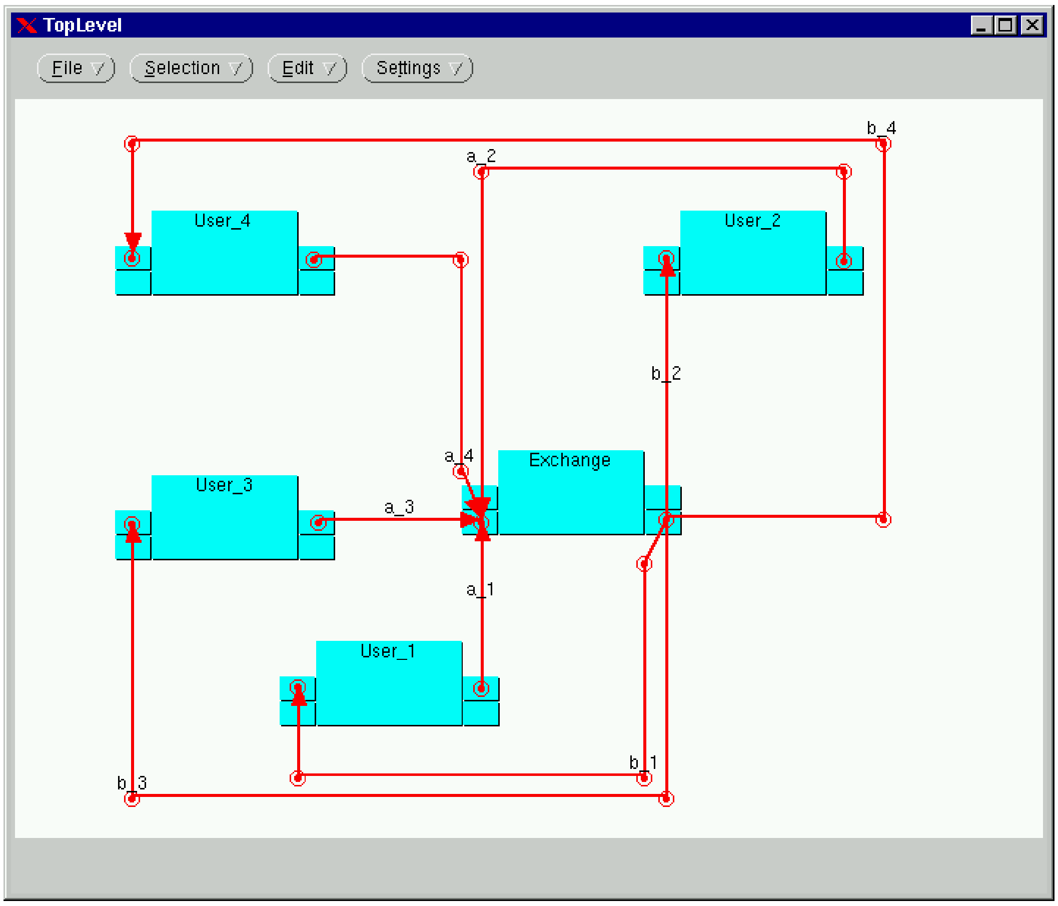


Figure 4.10: The Telecommunication Example within Destool

4.4 Verifying the Specification of the Telecommunication Example

Verification is the process of proving that a specification possesses certain properties. We are going to verify the specification of the telecommunication example using the notions introduced in chapter 3. However, we will not fully verify our specification formally. There are two major reasons for it: First, a complete formal verification is hardly to be done by hand. It is very time consuming and spacious and therefore prone to errors. Second, we would probably miss the point of demonstrating how to use the notions to verify a specification.

The verification will be done for a user component. Often we omit a complete proof and present just some examples how such proofs would have to be performed. Furthermore, for the sake of simplicity and clarity, we are going to use abbreviations for the notions applied in the specification example throughout this verification. Most of these abbreviations should be straightforward, with probably one exception, the use of _. For example, we will use $\mathtt{r(s1(\_))} : 0$ to denote that there is no signal one at all, neither a dialling tone nor an engaged tone.

4.4.1 World Description, Input Models, and Goals

The verification method we developed in chapter 3 is based on a world description, i.e. a set of given situations, and a set of goals that should be verified. Since we often deal with input models, we will give the set of all complete input models related to the world description explicitly.

The World Description for KB_U

We mentioned earlier that the process of creating the world description is a crucial task. For example, it might be the case that our world description has the same biases as the specification. It might also be the case that not all relevant situations will be covered.

Our world description is relevant for the specification of a user component and consists of nine situations. Situations one to six are related to the process of making a call and situations seven, eight, and nine consider the case of an incoming call.

$\mathtt{W_U} = \{\mathtt{N_{U_1}}, \ldots, \mathtt{N_{U_8}}\}$, where

$$\begin{aligned}
\mathtt{N_{U_1}} = \langle &\mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(rt))} : 1, \mathtt{r(ce(A))} : 1, \mathtt{talk(A)} : 1,\\
&\mathtt{s(rl)} : 1, \mathtt{s(pn(A))} : 1, \mathtt{s(wam)} : 1, \mathtt{s(rpd)} : 1\rangle,
\end{aligned}$$

$$\begin{aligned}
\mathtt{N_{U_2}} = \langle &\mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(rt))} : 1, \mathtt{r(ce(A))} : 0,\\
&\mathtt{s(rl)} : 1, \mathtt{s(pn(A))} : 1, \mathtt{s(wam)} : 1, \mathtt{s(rpd)} : 1\rangle,
\end{aligned}$$

$$\mathtt{N_{U_3}} = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(et))} : 1, \mathtt{s(rl)} : 1, \mathtt{s(pn(A))} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_4}} = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(\_))} : 0, \mathtt{s(rl)} : 1, \mathtt{s(pn(A))} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_5}} = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(et))} : 1, \mathtt{s(rl)} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_6}} = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(\_))} : 0, \mathtt{s(rl)} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_7}} = \langle \mathtt{tc} : 1, \mathtt{r(bell)} : 1, \mathtt{r(ce(B))} : 1, \mathtt{talk(B)} : 1, \mathtt{s(rl)} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_8}} = \langle \mathtt{tc} : 1, \mathtt{r(bell)} : 1, \mathtt{r(ce(B))} : 0, \mathtt{s(rl)} : 1, \mathtt{s(rpd)} : 1\rangle,$$

$$\mathtt{N_{U_9}} = \langle \mathtt{tc} : 0, \mathtt{r(bell)} : 1, \mathtt{s(rl)} : 0\rangle.$$

For example, `cw(A):1` stands for `connection_wanted(A)` is true, `s():1` denotes that something was sent, and `r():0` that something was not received.

The Input Models $In(N_{U_1}), \ldots, In(N_{U_9})$

For any model N_{U_i}, $In(N_{U_i})$ denotes the input model that copies the input truth assignment of N_{U_i} but assigns u to all other atoms. For the sake of clarity we omit the atoms which are assigned unknown, hence we only consider the complete input models of W_U.

$\mathtt{In}(\mathtt{W_U}) = \{\mathtt{In}(\mathtt{N_{U_1}}), \ldots, \mathtt{In}(\mathtt{N_{U_9}})\}$, where

$\mathtt{In}(\mathtt{N_{U_1}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(rt))} : 1, \mathtt{r(ce(A))} : 1 \rangle$,

$\mathtt{In}(\mathtt{N_{U_2}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(rt))} : 1, \mathtt{r(ce(A))} : 0 \rangle$,

$\mathtt{In}(\mathtt{N_{U_3}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(et))} : 1 \rangle$,

$\mathtt{In}(\mathtt{N_{U_4}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(dt))} : 1, \mathtt{r(s2(\_))} : 0 \rangle$,

$\mathtt{In}(\mathtt{N_{U_5}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(et))} : 1 \rangle$,

$\mathtt{In}(\mathtt{N_{U_6}}) = \langle \mathtt{cw(A)} : 1, \mathtt{r(s1(\_))} : 0 \rangle$,

$\mathtt{In}(\mathtt{N_{U_7}}) = \langle \mathtt{tc} : 1, \mathtt{r(bell)} : 1, \mathtt{r(ce(B))} : 1 \rangle$,

$\mathtt{In}(\mathtt{N_{U_8}}) = \langle \mathtt{tc} : 1, \mathtt{r(bell)} : 1, \mathtt{r(ce(B))} : 0 \rangle$,

$\mathtt{In}(\mathtt{N_{U_9}}) = \langle \mathtt{tc} : 0, \mathtt{r(bell)} : 1 \rangle$.

Goals for KB_U

We will consider all output literals to be goals, as well as two extra goals: first, whenever the receiver is lifted it has to be put down again, and second, we would like to establish a talk to a user B, in other words:

$\mathbf{Q}_U = \{s(rl), s(pn(A)), s(wam), s(rpd), s(rl) \wedge s(rpd), talk(B)\}$.

4.4.2 Verifying Empirical Foundedness

The idea of the underlying verification method is to check whether the specification in some sense fits the world description. It is therefore important to know whether the world description is detailed enough to make a specification possible at all. It might simply be the case that our world description is underspecified and that no specification that always gives the right answer can be found. This problem was addressed by the notion of empirical foundedness.

The world description W_U is empirically founded if for every goal $\varphi \in \mathbf{Q}_U$ the following holds:

$$\forall\, N_{U_i} \in W_U \quad N_{U_i} \models \varphi \;\Rightarrow\; In(N_{U_i}) \mid\!\models_{W_U} \varphi.$$

Lemma 4.4.1
The world description W_U is empirically founded.

Proof
Outline: By the definition of forcing, we need to prove the truth of each refinement in W_U of $In(N_{U_i})$ for every N_{U_i}. However, for each $In(N_{U_i})$ there exists only one refinement in W_U, namely N_{U_i} itself. Hence, empirical foundedness holds. □

4.4.3 Reasoning Trees for the Users

In order to visualise the reasoning process we will present the reasoning trees for the user component. It is easy to realise that the specification of a user component could be further divided into two subcomponents, one responsible for taking a call and the other for making a call. Therefore, we will present two reasoning trees, where the arches are labelled with input information and nodes denote output information.

Incoming Call

When the user receives a bell signal he can decide whether to take the call or not. If the receiver is lifted, it will also be put down again. It might also be the case that a talk is prevented if no connection is established.

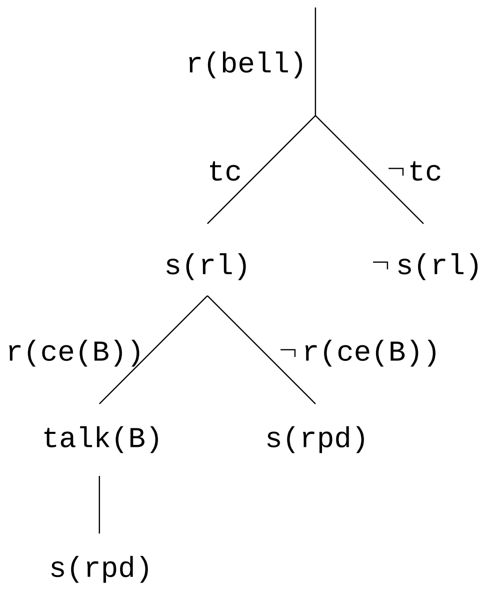


Figure 4.11: Reasoning Tree for KB_U - Incoming Call

Outgoing Call

In order to make a call, the user wants to establish a connection and therefore, he has to lift the receiver. In case the user does not receive the appropriate signals or no signals if they are expected, the receiver will be put down. This also happens if the call is not answered. Otherwise, a talk can be made. Again, whenever the receiver is lifted, it will be put down again.

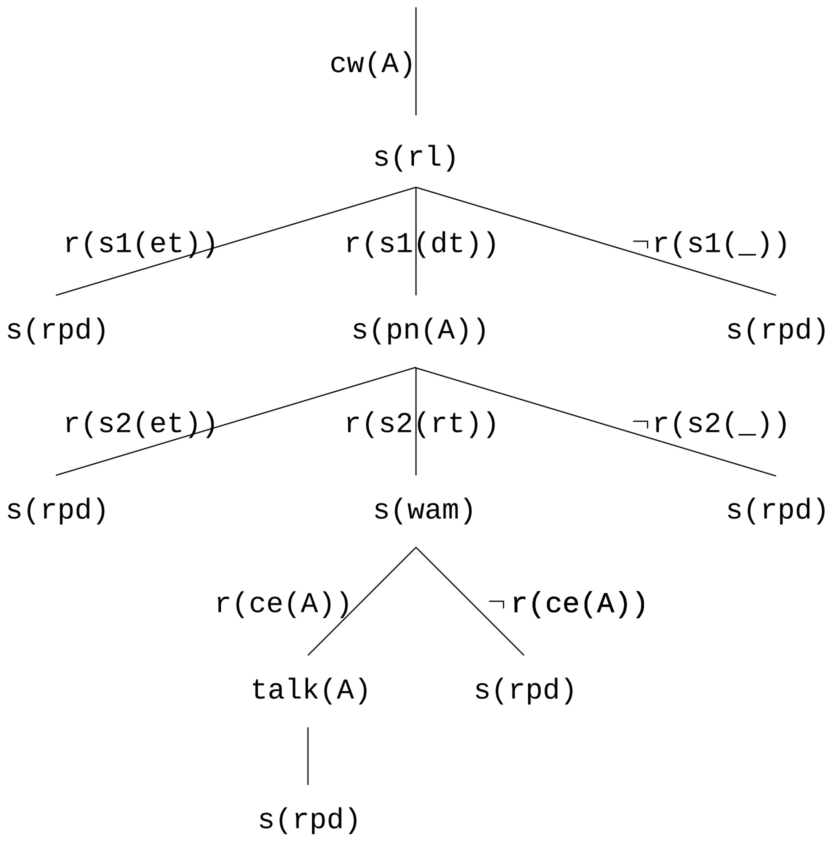


Figure 4.12: Reasoning Tree for KB_U - Outgoing Call

4.4.4 Verifying Correctness and Soundness

The specification of the user component is correct if it holds in the world, i.e. in every situation determined by the world description. In other words, KB_U is correct w.r.t. the world description W_U if for all goals $\varphi \in \mathbf{Q}_U$ it holds:

$$\forall\, N_{U_i} \in W_U \quad In(N_{U_i}) \cup KB_U \vdash_{FC} \varphi \;\Rightarrow\; N_{U_i} \models \varphi.$$

This means, for each input model $In(N_{U_i})$ we have to infer by forward chaining each possible goal using the knowledge base KB_U. Then, for each goal that was inferred we have to determine whether it holds in N_{U_i}. If there is only one goal that was inferred but does not hold, then KB_U is not correct, otherwise correctness w.r.t. W_U holds.

Lemma 4.4.2
KB_U is correct w.r.t. the world description W_U.

Proof
Applying the above procedure will show the correctness of KB_U w.r.t. W_U. Here, we only consider two examples:
First, we take N_{U_1}. From $In(N_{U_1})$ and KB_U it is possible to infer the following literals: $\{s(rl) : 1, s(pn(A)) : 1, s(wam) : 1, talk(A) : 1, s(rpd) : 1\}$. Therefore, all possible conjunctions of these literals can also be inferred, hence $s(rl) \wedge s(rpd)$ is inferable, too. Since each literal as well as $s(rl) \wedge s(rpd)$ hold in N_{U_1}, we can conclude that KB_U is correct for N_{U_1} w.r.t. W_U.
Second, we take N_{U_9}. From $\langle \mathtt{tc} : 0, \mathtt{r}(\mathtt{bell}) : 1\rangle$ we can only infer $\{\neg\ s(rl)\}$, i.e. $\{s(rl) : 0\}$, which also holds by N_{U_9}. Hence, KB_U is also correct for N_{U_9} w.r.t. W_U.
In order to prove the correctness of KB_U w.r.t. W_U we have to show the correctness of KB_U for all $N_{U_i} \in W_U$. □

Because the inference relation $\vdash_{FC}$ is monotonic it holds by theorem 3.3.1 that KB_U is sound w.r.t. W_U.

4.4.5 Verifying Consistency

Consistency is a notion standing for the absence of contradictory information. We introduced consistency as a property that depends on a given set of models. In particular, we will consider the set of partial models that can be obtained from the world description W_U, i.e. $P(W_U)$. Then, KB_U is consistent w.r.t. a set of models $P(W_U)$ if there is no model $M \in P(W_U)$ and no goal $\varphi \in \mathbf{Q}_U$ such that

$$M \cup KB_U \vdash_{FC} \varphi \ \ and \ \ M \cup KB_U \vdash_{FC} \neg\, \varphi.$$

In section 3.4.2 we introduced several cases of inconsistency, like contradictions in facts, rule-pairs, or chains of inference, as well as self-contradicting rules or antecedents. It is easy to find that none of these special cases holds for our specification of a user component.

Lemma 4.4.3
KB_U is consistent w.r.t. $P(W_U)$.

We will not formally prove consistency. Above, we presented the reasoning trees of the user component. It is easy to realise that, given the world description W_U, there is no situation possible in which a goal and its negation can be derived. Furthermore, by lemma 3.4.1 we know that KB_U is at least consistent w.r.t. $In(W_U)$, because KB_U is weakly complete w.r.t. W_U.

4.4.6 Verifying Weak Completeness and Decisiveness

A specification is weakly complete if it does not contain gaps, i.e. for a given input model a conclusion that holds must also be inferable. Formally, KB_U is weakly complete w.r.t. the world description W_U if for all goals $\varphi \in \mathbf{Q}_U$ it holds:

$$\forall N_{U_i} \in W_U \quad In(N_{U_i}) \models_{W_U} \varphi \;\Rightarrow\; In(N_{U_i}) \cup KB_U \vdash_{FC} \varphi.$$

In section 3.5.2 we introduced several cases indicating gaps in a rule-based specification. These cases are: unnecessary literals, illegal literals, dead-end, unreachable rules, and unreachable literals. When inspecting the specification of the user component it is easy to find that none of these examples can be fulfilled.

Lemma 4.4.4
KB_U is weakly complete w.r.t. the world description W_U.

Proof
Outline: For each $In(N_{U_i})$ there exists only one refinement in W_U, namely N_{U_i} itself.Therefore, we only need to prove for all goals $\varphi \in \mathbf{Q}_U$ that the following holds:

$$\forall N_{U_i} \in W_U \quad N_{U_i} \models \varphi \;\Rightarrow\; In(N_{U_i}) \cup KB_U \vdash_{FC} \varphi.$$

For example, for N_{U_1} the following goals hold: $\{s(rl), s(pn(A)), s(wam), s(rpd), talk(A), s(rl) \wedge s(rpd)\}$. By the proof of lemma 4.4.2 we already know that each goal can also be inferred from $In(N_{U_1})$ and KB_U. Therefore, KB_U is weakly complete for N_{U_1} w.r.t W_U. Since it is possible to show that KB_U is weakly complete for all N_{U_i}, it is weakly complete w.r.t. W_U. □

Because KB_U is sound, weakly complete, and W_U is empirically founded, it holds by theorem 3.6.1 that KB_U is decisive w.r.t. the world description W_U.

4.4.7 Verifying Well-Informedness and Strong Completeness

Well-informedness shall insure that a specification does not contain any superfluous information. In section 3.7.1 we introduced well-informedness w.r.t. an arbitrary set of complete input models. In order to show strong completeness it is sufficient enough to consider well-informedness w.r.t. the set of complete input models of the world description.

KB_U is well-informed w.r.t. a set of complete input models $In(W_U)$, if for any partial model $M \in P(In(W_U))$ any goal $\varphi \in \mathbf{Q}_U$ can be derived from the model M if it can be derived from all refinements R in $In(W_U)$ of M. With other words, if for all $M \in P(In(W_U))$ it holds:

$$[\forall\, R \in In(W_U) \quad M \leq R \Rightarrow R \cup KB_U \vdash_{FC} \varphi] \Rightarrow M \cup KB_U \vdash_{FC} \varphi.$$

The following algorithm on checking well-informedness is based on [Leemans et al., 1993]. Let a set of complete input models $In(W_U)$ be given. Take a partial model $M \in P(In(W_U))$. From this M take all complete refinements within $In(W_U)$, and for every refinement determine what can be derived from it via KB_U. Take from these derivations the greatest common information state, i.e. the conclusions that all refinements agree on. This greatest common information state must equal what can be derived from the partial model M via KB_U. If it is equal, then well-informedness holds.

Lemma 4.4.5
KB_U is not well-informed w.r.t. $In(W_U)$.

Proof
For $\mathtt{M} = \langle \mathtt{tc} : \mathtt{u}, \mathtt{r(bell)} : \mathtt{u} \rangle$ it holds $M \in P(In(W_U))$. The only possible refinement of M in $In(W_U)$ is $In(N_{U_9})$. Now $In(N_{U_9}) \cup KB_U \vdash_{FC} \neg\, s(rl)$ but $M \cup KB_U \nvdash_{FC} \neg\, s(rl)$. Hence, we found one M such that well-informedness does not hold. □

Circularity, redundancy, and subsumedness are special cases of well-informedness. When looking for these special cases in the specification of the user component we could not find any. After further investigations we found that our world description W_U is not sufficient. It misses the cases that no bell rings or that no call wants to be made. Therefore, we have to build an extended world description W'_U, where W'_U is at least equal to $\mathtt{W_U} \cup \{\mathtt{N_{U_{10}}}, \mathtt{N_{U_{11}}}\}$, with $\mathtt{N_{U_{10}}} = \langle \mathtt{tc} : \mathtt{1}, \mathtt{r(bell)} : \mathtt{0} \rangle$ and $\mathtt{N_{U_{11}}} = \langle \mathtt{tc} : \mathtt{0}, \mathtt{r(bell)} : \mathtt{0} \rangle$. Now KB_U is well-informed w.r.t. $\{In(N_{U_7}), In(N_{U_8}), In(N_{U_9}), In(N_{U_{10}}), In(N_{U_{11}})\}$, i.e. for the case of an incoming call.

At this point we will not go further. The proof for well-informedness can hardly be done by hand. Therefore, we will assume that KB_U is well-informed w.r.t. a world description W''_U, which is an extension of W'_U. Then the following holds: KB_U is sound and weakly complete w.r.t W_U. Since KB_U is not well-informed w.r.t. $In(W_U)$, it it holds by theorem 3.7.2 that KB_U is not strongly complete w.r.t. W_U. However, KB_U is well-informed w.r.t. $In(W''_U)$ and therefore, KB_U is strongly complete w.r.t. the world description W''_U.

4.5 Conclusion

DESIRE is a valuable specification tool. It offers the possibility to model a problem within a graphical editor and allows the transformation towards an executable specification. However, we provided only a simple specification. For example the specification of the user component only consists of twelve rules and two initialising facts.

When verifying the specification we found the notions introduced in chapter 3 useful. However, we also encountered the problem of creating a suitable world description. Further, an automated verification tool supporting our approach is needed. It would be best if such verification tools can be incorporated within the DESIRE development environment.

Chapter 5

Results and Discussion

The last chapters were devoted to the investigation of knowledge engineering as a source of information for specification engineering. We looked at the relationship of specification and knowledge engineering, we extended a theory of verification, and finally, we made a case study on the formal specification and verification of a telecommunication network. Here, we will first summarise the results of the last chapters and later, we are going to discuss these results, especially considering the value of the framework DESIRE.

We found out that knowledge engineering and specification engineering have much in common. Therefore, it is possible to reuse principles from knowledge engineering. We also extended a work on the verification of knowledge bases such that it can be used for the verification of formal specifications. Further, the complexity of goals that can be proved could be increased. Finally, we performed a specification and verification case study using DESIRE, a system designed for the specification of knowledge bases. DESIRE itself can also be regarded as a knowledge representation system. Hence, we could show that tools and methods from knowledge engineering can be applied in specification engineering.

As a part of this chapter we will now investigate how DESIRE in particular supports specification engineering. Therefore, we will face DESIRE with the demands on tools, the concepts of formal methods, and the lightweight approach, as introduced in chapter 2. We also identified several problems in applying specification engineering. In order to overcome such problems, we have to show the ability to integrate DESIRE with the software development process. We will look at this issue by considering the Standard Siemens Development Methodology. Furthermore, since outsourcing is often used in practice, we will also show the possibility to outsource the specification and verification process to specialised firms.

We will end this chapter with a discussion on DESIRE, presenting what we found to be weak areas of the tool. Finally, some limitations of this work will be named.

5.1 Results

This thesis consists of three main parts: the discussion of the relationships between formal methods and knowledge engineering, the work on the verification of formal specifications and knowledge bases, and finally, the case study on formal specification and verification of a telecommunication example. We will now look at the results of the last chapters and at its contributions.

5.1.1 Relating Formal Methods and Knowledge Engineering

There is strong evidence that formal methods and knowledge engineering overlap in many parts. Unfortunately, we found hardly any articles discussing this issue. Therefore, this work seems to be a first attempt to summarise similarities and differences between knowledge engineering and formal methods.

The main difference between the two approaches lies in their intension: a formal specification is an abstract representation of a program and as such it usually will be refined to become a program. In contrast, knowledge engineering deals with the representation of and reasoning about knowledge in an abstract way, without changing the representation formalism later.

Formal specification and knowledge representation meet at the point of representing facts and relationships in an abstract manner. From a theoretical point of view, this representation is a logical theory. In chapter 2 we identified four basic activities common to knowledge engineering and specification engineering: (1) Choosing a representation formalism; (2) Building a theory; (3) Deciding on a proof theory; and (4) Inferring facts. In particular the task of building a theory is the key point of knowledge engineering and specification engineering, as it was also pointed out by [Turski and Maibaum, 1987] and [Goltz and Herre, 1990].

We also introduced the participants of specification and knowledge engineering. We identified three participants in specification engineering, i.e. the specifier, the programmer, and the customer. In this way we found that the specifier and the knowledge engineer, the programmer and the user, and the customer and the expert have some similarities. For example, both specifier and knowledge engineer have to capture ideas and to create formal representations of it.

Our comparison is by no means complete. It is intended to give a introduction to the relationships of both research fields. We strongly believe that both research areas of computer science can benefit from each other. We especially emphasised the possibilities to use knowledge engineering principles, as well as knowledge representation systems, in specification engineering.

5.1.2 On the Verification of Formal Theories

There is much consensus about the necessity to ascertain correctness of specifications and knowledge bases. However, correctness is often used in an intuitive way. In doing so, the meaning of the attribute 'correct' is, as [Turski and Maibaum, 1987, p. 1] point out, emotionally loaded. This means, correctness is good, lack of it is bad. The main problem is that emotionally loaded terms are often used in different ways. Therefore, such terms have to be analysed more precisely.

Actually, there exist many notions for the verification of specifications and knowledge bases. [Preece et al., 1992] and [Treur and Willems, 1994] discuss some of them. The work by [Leemans et al., 1993] and [Treur and Willems, 1994] was especially developed for verifying knowledge bases expressed in propositional logic. Further, the authors restricted themselves to the derivation of literals.

We were able to extend this work in three ways: first, we not only considered knowledge bases but universal theories. This improved the generality of our work. Second, we used a set of goals, containing open formulas and existentially quantified open formulas. Therefore, we are able to verify more complex structures. Finally, we developed our theory using the truth relation instead of the inference relation. However, we did show that the 3-valued truth relation can be approximated by the inference relation of the calculus of Forward Chaining. This means, it is possible to build a proof tool to check theories for the introduced properties.

As a result of chapter 2 we know that formal specifications and knowledge bases are formal theories. Therefore, the results of chapter 3 can be used for the verification of specifications and knowledge bases. In providing clear definitions of the notions of verification, we are laying foundations for the development of methods and tools to verify formal specifications and knowledge bases.

5.1.3 Specification and Verification of a Multi-Agent-System

In order to put this work into a practical context, we provided a case study of formal specification and verification in chapter 4. We decided to use a simple telecommunication example, because most people would be familiar with it.

In order to develop the specification, we used DESIRE, a system originally developed for the specification of knowledge based systems. Further, DESIRE together with its supporting tools Destool and tgm can be considered as a knowledge representation system as introduced by figure 2.1. Therefore, we showed the ability of a knowledge representation systems to be a valuable specification tool. Finally, we could demonstrate a verification process using the notions of verification introduced in chapter 3.

5.2 Discussion

In this section we will relate our work, and especially DESIRE, to the specification engineering process as it was introduced in section 2.1. Furthermore, we discuss the approach of formal specification and verification in the context of the software development method used at the Siemens AG Austria. Since outsourcing is a commonly applied process in development, we will also look at the possibilities to outsource the formal specification and verification process. Afterwards, we will discuss two alternatives to formal methods which are currently used in practice, tests and reviews. Some remarks on DESIRE and a discussion of the limitations of this thesis will conclude this section.

5.2.1 DESIRE and the Demands and Concepts of Formal Specification

In section 2.1 we introduced the process of specification engineering. We identified several properties a good specification has to fulfil. Further, we introduced fundamental concepts, as well as demands on tools and methods, in order to build good specifications. Finally, a lightweight approach to formal specification was introduced. Now, we show that the framework DESIRE is a valuable tool in specification engineering by relating it to the introduced concepts and demands.

Properties of a Specification

DESIRE specifications are unambiguous, since DESIRE is a formal language. Further, DESIRE specifications are easily modifiable. This is due to the compositional framework of DESIRE. Changes within a component can be done without changing the system, as long as the input and output signatures are not modified.

In order to prove completeness and consistency of DESIRE specifications, it has to be distinguished between static and dynamic properties. Static properties of the components can be proved using the notions introduced in chapter 3. Notions and methods for the verification of the dynamics were introduced by [Treur and Willems, 1995] and applied by [Cornelissen et al., 1997].

Fundamental Concepts

Abstraction is the process of removing details from a representation. Within the framework DESIRE abstraction is especially supported by information hiding. On the top level, only components and links, representing the information flow, are depicted. In that way it is possible to capture a general overview of the system. For more information it is possible to zoom into composed components, or to obtain signatures and targets from primitive components. In our opinion, the pictorial view on DESIRE specifications is very clear.

The framework DESIRE is aimed at the specification of compositional architectures. Basically, the notion of a compositional architecture arises from a strong relationship between the notions of declarative semantics and functional task decomposition. The semantics of the whole system can be obtained as a composition of the semantics of each of the components, by means of generic, predefined and standardised construction principles. Further, the functionality of a compositional architecture is determined by the functionalities of its components and by the way these functionalities interact. Hence, the concepts of composition and decomposition are natural approaches within DESIRE.

Further, combination of languages is also supported by DESIRE. Within DESIRE it is possible to use so-called alternative specifications. For example, such alternative specifications can be databases or any kind of algorithms written in any programming language. The communication with the execution manager is done via files including input and output information in a predefined format. The replacement of components with conventional components also allows a step by step refinement of the specification towards an executable program.

While developing a DESIRE specification a number of generic tasks can be identified. A generic model can be used for a wide variety of more specific tasks through refinement and composition. For example, in chapter 4 the specification of a user component is in principle generic. We only had to adjust the behaviour by providing additional information, like who should be called and should the receiver be lifted if the bell rings. In that way reuse is possible, which in turn reduces time, costs, and effort needed to design and maintain system designs.

Finally, we turn to data structures and algorithms. Unfortunately, we are not so familiar with the internal development of DESIRE. Therefore, we do not know whether new data structures and algorithms were used or not.

Demands on Tools and Methods

In section 2.1 we introduced several criteria for tools and methods to be valuable in specification engineering. Of course, DESIRE has to be measured against them.

- Early payback. We did not apply DESIRE in a real development process, we just specified a simple telecommunication example. Therefore, we are not able to comment on this point. However, since formal methods help to structure ideas, it can be assumed that they are always beneficial as soon as they are used.

- Incremental gain for incremental effort. This might be the case, since it is possible to specify systems at different levels of detail. It is possible to simply specify the components and the information exchange for demonstration purpose but also to add signatures and knowledge bases in order to create an executable prototype.

- Multiple use. Due to the compositional method and the specification of generic tasks, it is possible to reuse earlier specified parts. However, in general, DESIRE is best used for the specification of multi-agent systems which are usually distributed systems.

- Integrated use. The integration of DESIRE with a software development method will be discussed in the next subsection.

- Ease of use and ease of learning. It is a bit difficult to comment on this issue. On the one hand, DESIRE is easy to use. It is very simple to construct the pictorial representation of a system. On the other hand, to fully understand the syntax and semantics of DESIRE is more difficult. Unfortunately, the documentation to DESIRE is not very good. There only exists a manual (in Dutch) and a short description on how to use the development environment. Further, there exists course material from the annually offered courses on DESIRE which is handed out to each participant. In our opinion, it is quite recommendable to attend such a course.

- Efficiency. This is probably a weak site of DESIRE. First, the prototype created by DESIRE is in PROLOG, which is not an execution efficient language. Further, there are no verification tools incorporated into the development framework. However, syntax checking and limited semantic checking (comparable to type-checking) is provided.

- Error detection oriented. DESIRE does not include a verification tool. However, since it is possible to generate a prototype and to test it within the DESIRE framework we would say that DESIRE is error detection oriented.

- Focused analysis. This is possible due to the debugger included in the execution manager. In that way it can be focused on special components while executing the prototype. Further, it is possible to distinguish between static and dynamic properties of the system. Thus, both properties can be verified independently.

- Evolutionary development. DESIRE strongly supports the development of partial specifications, basically due to compositional character of DESIRE specifications. Further, the verification method presented in chapter 3 supports partial verification of components. Therefore, the presented approach is strongly recommendable for evolutionary development.

A Lightweight Approach to Formal Specification

In section 2.1 we also introduced a lightweight approach to formal specification. The key points were: partiality in language, partiality in modelling, partial analysis, and partiality in composition. Here, we are going to show that DESIRE and the compositional verification method fit well into this lightweight approach.

The language used in DESIRE is based on many-sorted 3-valued predicate logic of first order. Modelling within this language is simple and fairly intuitive. Further, only a production rule style is used to represent relationships, which serves clarity and easy analysis. Therefore, DESIRE fulfils the point of partiality in language. On the other hand, the mathematical toolkit within DESIRE is fairly poor. This makes specifying mathematical relationships a bit difficult.

Partiality in modelling and partiality in analysis are well supported. Both follow straight from the compositional framework of DESIRE. Once the interfaces are specified it is up to the modeller to decide which components should be specified in more detail. Further, verification can be limited to selected components. Finally, partiality in composition also follows from the compositional framework. Components can be specified separately and later be composed using links. This means, DESIRE is worth using in the scope of a lightweight approach to formal specification.

5.2.2 Formal Specification and Verification within stdSEM

The Standard Siemens Development Methodology

The Standard Siemens Development Methodology, [stdSEM, 1997], defines a set of activities that should be followed in order to develop a software product. In particular stdSEM uses the waterfall approach of software development (see [Sommerville, 1992]). Figure 5.1 displays the stages of the software development process according to stdSEM.

Within the waterfall model there are a number of stages that follow each other in sequence (the arrows downwards). In practice, however, the development stages overlap and often information from one stage will be handed back to the previous stage (the arrows upwards).

The advantage of the waterfall model is its simplicity. Each stage of the development process can be defined independently and therefore, it is possible to define the outcome of each stage easily. The process of formal software specification is part of the definition phase.

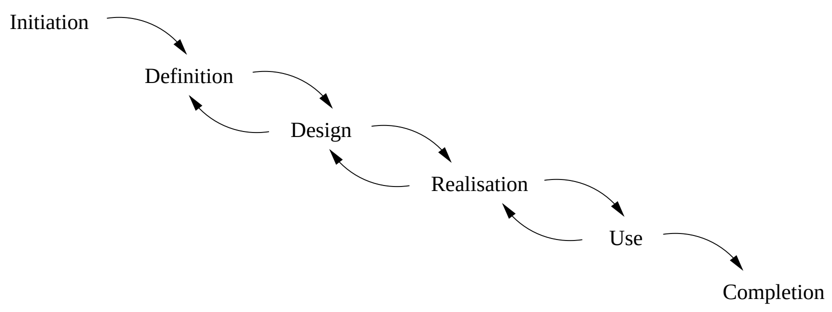


Figure 5.1: The Software Development Process according to stdSEM

The Phase “Definition”

The goal of the definition phase is to collect, to develop, and to analyse the requirements of the software product. It aims at getting well defined and validated requirements of the product. The authors of stdSEM strongly advise to use methods and tools to support requirements engineering. Further, the goal of the project has to be defined, as well as an evaluation of the feasibility has to be created. Finally, the course of the project has to be set up.

The outcome of the definition phase are the following documents: requirement specification, functional specification, several planning documents, and, if necessary, an offer to the customer. In addition to these documents, domain models, interface specifications, as well as prototypes might be developed.

Prototyping

Prototyping is actually another software development method but it can also be incorporated into other development methods. A prototype is a program for user experiment. However, the object of prototyping is to establish the system requirements. Later, the prototype is reimplemented to produce a production-quality system. Normally, the prototype is thrown away or will be kept for reuse in other projects.

Formal Methods and DESIRE within stdSEM

The approach of formal methods, as well as DESIRE, the framework for the design and specification of compositional architectures (see chapter 4), fit well into stdSEM. Both can be used throughout the entire development process but especially in the definition phase. DESIRE, in particular, can also be used for the construction of prototypes.

Formal specifications are precise and unambiguous. Thus, they remove areas of doubt in specification. The principle value of using formal methods is that it forces an analysis of the system requirements at an early stage. The costs of specification increase, however, correcting errors at this stage is cheaper than modifying a delivered system. Further, using formal methods increases the confidence in a system. It is not yet entirely clear, if formal methods increase or decrease the costs of development.

The best place to use formal specification methods within stdSEM is the definition phase, when the requirements are defined and validated. DESIRE can be used for the functional specification of a system. Due to the necessity to specify input and output signatures within DESIRE, it is perfectly suited for interface specifications. Finally, since an implementation generator is incorporated into the DESIRE development environment, it can be considered as an executable specification language, well suited for prototyping.

Executable specification languages, like DESIRE, are often proposed for the use in prototyping. [Sommerville, 1992] recommends [Hekmatpour and Ince, 1988] for further reading on this issue. Especially the framework DESIRE has the potential to be a good prototyping tool. It offers an easy to use graphical editor for fast design. We already argued that DESIRE supports compositional development, a method frequently used in prototyping. Furthermore, in prototyping it is most important to develop appropriate user interfaces. DESIRE allows to include such specifications in form of conventional components. Finally, the execution trace created by the execution manager is a good basis for further analysis.

Formal methods in general and DESIRE in particular can in principle be used within the software development method used at the Siemens AG Austria. We showed some ways of using these methods and tools. Of course, a study investigating the impact of formal methods and DESIRE within stdSEM needs to be carried out.

5.2.3 Outsourcing of the Formal Specification and Verification Process

According to the Collins English Dictionary [col, 1994], to outsource has two meanings:

1. to subcontract (work) to another company; or
2. to buy in (components for a product) rather than manufacture them.

In the scope of this thesis, we will mainly consider formal methods with respect to the first meaning. We already mentioned the work by [Easterbrook, 1996] and

[Wing, 1985]. Both discuss the advantages and disadvantages of an independent process of specification and verification.

In [NASA, 1989](as in [Easterbrook, 1996]) the process of independent verification and validation (IV& V) is characterised as follows:

> [IV& V] is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software.

The main value of such view is the fresh perspective it offers on questions of software safety and correctness. It is like a second opinion on a problem to counterbalance that of the developer. In that way the benefit is a significantly reduced risk of software errors. Further, since some errors are detected early in the software development process, they are cheaper to fix.

In chapter 2 we identified several reasons for the rare use of formal methods in practice, like lack of training, lack of high-level specification languages, and lack of support tools. As a possible solution, [Wing, 1985] suggests:

> Hiring trained specialists would be a feasible way of overcoming these difficulties and increase the use of specifications in software development. As these specialists gain recognition for their expertise, specification firms may arise ...

[Wing, 1985] compares specifiers with lawyers and architects. The analogy to lawyers is based on the use of a specification as part of a contract, and the analogy to architects is based on the use of a specification as a design. Just as law firms and architectural firms, specification firms can be specialised.

Once there exist specialised firms for specification, it is possible to outsource the work of creating and proving formal specifications. This view corresponds to partial outsourcing, where only specific and well-defined tasks are handed to a subcontractor. In contrast, complete outsourcing might also be possible. Thus, the entire software development process is given to a specialised firm, which in turn might hire other subcontractors.

For large companies, like Siemens AG Austria, it might even be advantageous to have specification groups within the company. For example, the American space agency NASA, the National Aeronautics and Space Administration, has its own formal methods program, which not only conducts research on formal methods but also applies them to real world problems. Actually, many NASA projects have been cited as successful applications of formal methods, e.g. [NASA, 1995].

5.2.4 Current Alternatives to Formal Methods

There might still be some formal methods advocates arguing that there are no alternatives to formal methods. However, since formal methods are not used frequently in industry, there must exist other methods to ensure software quality. Further, it is recognised that by the use of formal methods in the software development process, traditional development methods should not be abandoned.

Tests

Formal verification is not often used today. Therefore, testing is still the predominant technique of evaluation. Verification and test are two complementary notions. While verification is successful, if it shows the absence of errors, testing is successful, if it shows the presence of errors. An introduction to testing is given by [Appelrath and Ludewig, 1992, pp. 223–235] as well as by [Sommerville, 1992, pp. 373–387].

Testing should not be replaced by verification. For example, [Bowen and Hinchey, 1995b] report that even though a formal specification were proved to be correct, testing still showed the presence of an error. This means that formal methods are no guarantee of correctness. Formal methods can only increase the confidence in a system but errors can still exist. Hence, the ninth commandment by [Bowen and Hinchey, 1995b] still holds, which says:

> Thou shalt test, test, and test again.

However, as [Bowen and Hinchey, 1995b] point out, formal methods offer another alternative to traditional testing techniques, namely specification-based testing. The formal specification might be used as a guide for determining functional tests for the system. Further, the specification itself can be used to derive expected results of test data. Once more, formal methods showed its ability to be beneficial in the traditional development process.

Reviews

Another method to ensure software quality are reviews. A review involves a group of people examining parts or all of a software system or its associated documentation with the aim of finding system problems. The conclusions of the review are recorded and passed to whoever is responsible for correcting the discovered problems. Reviews are not limited to specifications or code, all documents created in a development process should be reviewed.

Reviews share a very important property with the independent verification and validation process [Easterbrook, 1996]. In both cases individuals not previously engaged in the development process have to investigate the documents. This fresh perspective often leads to a more rigorous search for errors. Further, documents, like specifications, might be interpreted in ways not originally in the mind of the developer. In this way reviews can lead to more unambiguity of the documents. Reviews are necessary, since it is usually not sufficient to fully verify all documentations.

Another advantage of reviews is their value for training purposes. They offer a good opportunity for designers to explain their design to other project members, newcomers, or designers who must interface with the system. More information on the review process can be found in [Sommerville, 1992, pp. 595–598].

5.2.5 A Discussion on DESIRE

The framework for Design and Specification of Interacting Reasoning components, DESIRE, is a specification system providing a graphical user interface, syntax and type checking facilities, an implementation generator, and an execution manager including a debugger. In our opinion, DESIRE is a very valuable specification tool. However, we also found some minor weaknesses.

First, we missed a verification tool within the DESIRE software development environment. This is especially disturbing, since it is mentioned in [Brazier et al., 1997, p. 50] that there exists one. However, the environment includes a syntax and semantics checker. In our opinion, at least better consistency checking facilities need to be included.

Second, the graphical user interface is very valuable. However, from time to time the user has to open so many editor windows that the clarity gets lost. In our opinion, it should also be possible to write and edit the textual specification directly.

Finally, we turn to the user manual and the online help. Both are most concerned with the use of Destool, the development environment. It is hardly possible to find any hints on the syntax definition or on how to specify certain functionality, like a closed world assumption. The user surely needs the course material [Brazier et al., 1997] from Amsterdam for a better understanding. On the other hand, for many editors default templates are available, which helps a lot in developing specifications.

Fortunately, the framework DESIRE is still evolving. It strongly profits from the feedback that is generated by its use in several projects. As far as we know, it is planned to develop a version of DESIRE for parallel platforms. Further, syntax and semantic checking facilities will be improved. Finally, research results on the reasoning process will also be incorporated.

5.2.6 Limitations

This thesis gives a good overview of formal specification and knowledge engineering. It introduces a formal verification theory and shows the applicability of formal specification and verification. It also puts the presented approaches into the context of a practical used development method. However, this thesis has also some limitations which are mostly due to the complexity of each of the problems we dealt with.

In chapter 2 we compared specification engineering and knowledge engineering by considering aims, methods, concepts, tools, and participants. Of course, the discussion is by no means complete. Each topic could be investigated in more detail, especially the relationship of knowledge representation languages and specification languages.

Then we developed a theory of verification. This theory is limited to static properties of universal theories. Further, the work is restricted to universal theories, Herbrand interpretations, and $\Sigma_1(Q)$-sentences. Also we used a world description which we considered to be given explicitly. The following problem arises: Under which conditions has a finite set W of Herbrand models an $\forall$-axiomatizable theory? What about this question if all models of W are assumed to be finite?

In chapter 4 we presented a simple specification example. This example is far too primitive to be a real world case. We also did not develop an implementation from this specification. This means that DESIRE needs first to be evaluated on more complex examples, preferably real world problems. Second, starting from the specification, code has to be developed. Ideally, this would be done like a scientific experiment with control cases, where we can look at the results of developing the same piece of code with and without formal methods.

During the work on the example specification we also had the idea of developing a generic task model for verification within DESIRE. Since DESIRE possesses an inference relation it should be possible to use it as a theorem prover. The question is how such a generic task model should look like.

Finally, we provided a short investigation on the use of formal methods within the Standard Siemens Development Methodology. We already noted that the integration of formal methods within traditional development methods is necessary for their success. Therefore, an investigation only concentrating on the problem of integrating formal methods into stdSEM has to be carried out.

Chapter 6

Conclusion and Future Work

Software development is a rather crucial and complex process. Therefore, software errors were usually excused in the past. However, nowadays many technical systems depend on the reliability of software, hence it must be possible to find ways to ensure the correct functioning of programs. It is now widely accepted that, if correctly applied, formal methods can improve software reliability.

Formal specification and verification methods are now on the edge of industrial applicability. For example, [Clarke and Wing, 1996], [Craigen et al., 1993] [Easterbrook et al., 1998], [Miller and Srivas, 1995], and [NASA, 1995] report the successful use of formal specification and verification within industrial settings. However, such examples are still exceptions rather than the rule in modern software development. It is therefore necessary to investigate and to overcome the impediments to industrial use of formal methods.

In order to establish formal methods in industry, tools and methods have to fulfil certain criteria. For example, tools and methods should be easy to learn and easy to use, tools should increase the efficiency in development, it should be possible to work error detection orientated, and an evolutionary development should be possible. However, the development of tools and methods is a very expensive and time consuming task. Therefore, it might be worthwhile to investigate existing methods and tools from other disciplines in computer science.

This thesis offers a new perspective on specification engineering. Specification engineering is the process of building and evaluating a description of a problem, i.e. a specification. Such problem description is a collection of knowledge about a domain of interest, and therefore, specification engineering deals with the collection, representation, and evaluation of knowledge. This in turn is the task of knowledge engineering, hence specification engineering can be considered to be a subtask of knowledge engineering. This conclusion leads to the view that methods and tools from knowledge engineering, as well as knowledge based systems themselves, can be used for specifying programs.

6.1 Summary and Conclusions

Specification engineering and knowledge engineering have much in common. Basically, both deal with the abstract representation of knowledge. On the one hand, it is the knowledge about what a program should do, and on the other hand, it is the knowledge about a domain of discourse. The outcome of both approaches is a formal description, i.e. a specification or a knowledge base respectively.

While developing specifications or knowledge bases both approaches face similar problems, rely on similar concepts, and have similar demands on tools and methods. For example, we identified the problem of knowledge acquisition, education, and technology transfer. Further, common concepts used are those of abstraction, composition, decomposition, and reusability. Finally, there are the demands on tools and methods, which were already summarised on the last page.

Another important problem is the quality of the representation. We named the following quality factors for specifications and knowledge bases: unambiguity, completeness, consistency, and modifiability. The process of evaluation, i.e. the process of verification and validation, shall ensure these properties.

We concentrated on verification as a formal and internal view related to concepts like completeness and consistency. Following the work by [Preece et al., 1992], [Leemans et al., 1993], and [Treur and Willems, 1994] we introduced the notions of correctness and soundness, consistency and ambivalence, weak and strong completeness, decisiveness, and well-informedness. Originally, these notions were introduced in the context of verifying knowledge bases. Due to the generalisation towards the use of universal theories, we are also able to use these notions for the verification of formal specifications. Thus, we provided an example of adjusting knowledge engineering methods for specification engineering.

The presented theory on verification strongly depends on a world description, i.e. a set of situations that should or should not occur. We showed that empirical foundedness is a desirable property of a world description. Additionally, we provided a set of goals that have to be verified. The advantage of such an approach, which was also propagated by [Yue, 1987], is the orientation towards an early error detection. However, since the whole verification process depends on the world description and on the goals both have to be chosen very carefully.

The notion of forcing was essential for the theory of verification. Forcing ensures that a goal holds in all possible situation that follow from a particular partial situation, i.e. it determines which goals should follow from a partial model. Throughout the development of our theory we used the 3-valued truth relation for defining the above properties. We also showed that this truth relation can be approximated by an inference relation like forward chaining. In that way it is possible to construct an automated theorem prover for checking knowledge bases and formal specifications. This certainly improves the efficiency of our approach.

Often, it is not feasible to prove all properties of theories formally. Therefore, we introduced special cases in order to verify consistency, weak completeness, and well-informedness. For example, a contradiction in rules shows inconsistency, gaps indicate the violation of weak completeness, and circularity and redundancy the breaking of well-informedness. [Leemans et al., 1993] introduced a proof tool for verifying these special cases.

We, however, did not only look for such special cases but used the definitions of the presented notions for a verification process. In order to do so, we developed a specification of a simple telecommunication network using the framework DESIRE. Due to its structure, DESIRE together with its supporting tools, including the graphical editor and the implementation generator, can be regarded as a knowledge representation system. Therefore, the use of DESIRE as a specification tool underpins our general idea.

To actually perform the verification we introduced a set of situations we considered to be relevant. This world description was shown to be empirically founded. Correctness and soundness, as well as weak completeness and decisiveness could be verified for the knowledge base of a user component. When verifying well-informedness, we realised that our world description had to be extended. This, however, showed that it is a crucial step to obtain an appropriate set of situations.

Earlier we demanded certain properties of tools and methods in order to be valuable in specification engineering. We were able to show that DESIRE possesses some of these properties. Hence, it was left to investigate the impact of formal methods, and especially the possibility to use DESIRE within the Standard Siemens Development Methodology (stdSEM). We realised, formal methods are best applied within the definition phase of stdSEM and for prototyping. DESIRE, in particular, turned out to be a valuable specification and prototyping tool.

Further, we asked whether it was possible to outsource, i.e. to subcontract, the work on formal specification and verification. We argued that outsourcing can be very effective. Not only that specialisation increases efficiency but often a fresh perspective raises questions not previously asked by the developers. In particular, larger companies should establish specification and verification groups within their company.

In this thesis we argued much in favour of formal methods for the development of software. However, we also pointed out that traditional development methods should not be abandoned. For example, although formal methods are propagated for developing reliable software, one should be aware that testing and reviews are still necessary evaluation methods.

We showed that formal specification and verification can benefit from achievements made in knowledge engineering. Therefore, we built a bridge between both disciplines of computer science. However, we are aware that there still remains much work on this issue.

6.2 Future Work

This thesis can be a starting point for investigations in many directions. First, further work has to be done on the relation of specification and knowledge engineering. Since [Guarino, 1995] related ontology and knowledge engineering, it would also be possible to investigate the relationship of formal ontology and specification engineering. An introduction to the terminology of ontology engineering is given by [Guarino and Giaretta, 1995].

The investigation of the verification of knowledge bases and specifications was embedded in a compositional specification and verification framework. We restricted ourself to static properties of a component. In order to fully verify compositional specifications, interactions of components, i.e. dynamic properties, have to be considered as well. Some preliminary work on this issue was already done by [Treur and Willems, 1995].

Another aspect of future work lies in the further development of DESIRE. First, some more syntax and semantic checks could be incorporated, like a 'defined but not used' analysis. Second, a verification tool should be incorporated into the development environment. This would much help specification developers to evaluate their work. Furthermore, it might be possible to allow two kinds of negation within a knowledge base. [Herre et al., 1995] propose to use partial logics with two kinds of negation for knowledge representation. Finally, more published case studies on the use of DESIRE are necessary. Ideally, such case studies will be performed in industrial environments in order to provide more insight into the value of DESIRE as a general specification tool.

Finally, investigations on the relation of DESIRE to other formal specification methods, in particular the translation of DESIRE specifications into other specification languages, might be a topic of research. Translations from one specification language into another play an important role in the field of viewpoint specifications [Bowman et al., 1995]. For example, the compositional approach within DESIRE seems to be comparable to schemas in the specification language Z [Spivey, 1992]. Further, DESIRE is designed for specifying parallel systems. This suggests some relationship to CSP, a specification language for concurrent systems [Hoare, 1985].

> There are two ways of constructing a software design. One way is to make it so simple that there are no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> [Hoare, 1981]

We hope that this work contributes to the first way.

Bibliography

Abadi, M. and Lamport, L. (1993). Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132.

Ainsworth, M., Cruickshank, A. H., Wallis, P. J. L., and Groves, L. J. (1994). Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51.

Appelrath, H.-J. and Ludewig, J. (1992). *Skriptum Informatik - eine konventionelle Einführung*. Teubner Verlag, Stuttgart, 2nd edition.

Boehm, B. W. (1981). *Software Engineering Economics*. Advances in Computing Science and Technology. Prentice-Hall, Englewood Cliffs New Jersey.

Boiten, E., Derrick, J., Bowman, H., and Steen, M. (1997). Constructive consistency checking for partial specification in Z. *Submitted for publication.* Online available: `http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency/cccfpsiZ.html` (last access 30/04/1998).

Bowen, J. and Stavridou, V. (1993). The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective. In Woodcock, J. C. P. and Larsen, P. G., editors, *FME'93: Industrial-Strength Formal Methods, Lecture Notes in Computer Science 670*, pages 183–195. Formal Methods Europe, Springer-Verlag. Online available: `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Jonathan.Bowen/fme93.ps.Z` (last access 27/05/1998).

Bowen, J. P. and Hinchey, M. G. (1995a). Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41. Online available as Technical Report #PRG-TR-7-94: `http://www.cl.cam.ac.uk/users/mgh1001/TECHREPORTS/7myths.ps.Z` (last access 14/07/1998).

Bowen, J. P. and Hinchey, M. G. (1995b). Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63. Online available as Technical Report No. 350: `http://www.cl.cam.ac.uk/users/mgh1001/TECHREPORTS/10cs.ps.Z` (last access 14/07/1998).

Bowman, H., Derrick, J., Linington, P., and Steen, M. (1995). FDTs for ODP. *Computer Standards and Interfaces*, 17(5–6):457–479. Online available: `http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency/fdtodp.html` (last access 01/05/1998).

Bratko, I. (1990). *PROLOG - Programming for Artificial Intelligence.* International Computer Science Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition.

Brazier, F. M. T., Treur, J., Wijngaards, N. J. E., and Willems, M. (1994). Temporal semantics and specification of complex tasks. Technical Report IR-375, Artificial Intelligence Group, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Online available: `http://www.cs.vu.nl/~wai/pub/reports/IR-375.ps.Z` (last access 01/06/1998).

Brazier, F. M. T., Treur, J., Wijngaards, N. J. E., and Willems, W. (1995). Formal Specification of Hierarchically (De)Composed Tasks. In Gaines, B. R. and Musen, M., editors, *Proc. of the 9th Banff Knowledge Acquisition for Knowledge Based Systems Workshop.* University of Calgary. Online available: `http://www.cs.vu.nl/~wai/pub/1995/Brazier_etal02.ps.Z` (last access 01/06/1998).

Brazier, F., Jonker, C., and Treur, J. (1996a). *Design of Multi-Agent Systems (Part 1).* Artificial Intelligence Group, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.

Brazier, F., Jonker, C., and Treur, J. (1996b). *Syllabus - Ontwerp van Kennissystemen.* Artificial Intelligence Group, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.

Brazier, F., Jonker, C., and Treur, J. (1997). *Design of Intelligent Multi-Agent Systems.* Artificial Intelligence Group, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.

Chang, C. C. and Keisler, H. J. (1990). *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 3rd edition.

Clarke, E. M. and Wing, J. M. (1996). Formal Methods: State of the Art and Future Directions. Technical Report CMU-CS-96-178, CMU Computer Science. Online available: `http://www.cs.cmu.edu/afs/cs/usr/wing/www/`

mit/paper/paper.ps (last access 14/07/1998) and http://www.cs.cmu.edu/Reports/1996.html (last access 14/07/1998).

Collins English Dictionary, 3rd Edition Updated (1994). Published by HarperCollins Publishers, PO Box, Glasgow G40NB. (ISBN 0 00 470678-1).

Cornelissen, F., Jonker, C. M., and Treur, J. (1997). Compositional Verification of Knowledge-Based Systems: A Case Study for Diagnostic Reasoning. In Plaza, E. and Benjamins, R., editors, *Proceedings of the 10th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW-97)*, Lecture Notes in Artificial Intelligence 1319, pages 65–80. Springer Verlag. Berlin. Online available: http://www.cs.vu.nl/~jonker/Papers/ekaw97.ps (last access 03/06/1998) and http://www.cs.vu.nl/~wai/Papers/ekaw97.ps (extended version) (last access 01/06/1998).

Craigen, D., Gerhart, S., and Ralston, T. (1993). An International Survey of Industrial Applications of Formal Methods (Volume 1: Purpose, Approach, Analysis and Conclusions, Volume 2: Case Studies). Technical Report NIST GCR 93/626-V1 & NIST GCR 93-626-V2 (Order numbers: PB93-178556/AS & PB93-178564/AS), Atomic Energy Control Board of Canada, U.S. National Institute of Standards and Technology, and U.S. Naval Research Laboratories, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA. Online available: http://hissa.ncsl.nist.gov/sw_develop/form_meth.html (last access 14/07/1998).

ct (1998). Editorial. *c't - Magazin für Computer Technik.* Heft 10/98, p. 3, Verlag Heinz Heise. Online available: http://www.heise.de/ct/98/10/003/ (last access 13/06/1998).

Dignum, F. and van de Riet, R. P. (1991). Knowledge Base Modelling Based on Linguistics and Founded in Logic. *Data & Knowledge Engineering*, 7:1–34. Online available: ftp://ftp.cs.vu.nl/pub/lics/kbm_ling_logic.ps.gz (last access 14/07/1998).

Easterbrook, S. (1996). The Role of Independent V & V in Upstream Software Development Process. In *Proceedings of the 2nd World Conference on Integrated Design and Process Technology (IDPT), Austin, Texas.* Online available as Technical Report #NASA-IVV-96-015: http://research.ivv.nasa.gov/docs/techreports/1996/NASA-IVV-96-015.ps (last access 30/04/1998).

Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D. (1998). Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Transactions on Software Engineering*, 24(1).

Online available: `http://eis.jpl.nasa.gov/quality/Formal_Methods/document/ieee1-98.ps` (last access 30/04/1998).

Engelfriet, J., Jonker, C. M., and Treur, J. (1997). Compositional Verification of Knowledge-Based Systems in Temporal Epistemic Logic. In Bossi, A., Marchiori, E., et al., editors, *Proceedings of the ILPS97 Workshop on Verification*. Online available: `http://www.dsi.unive.it/~bossi/ILPSWORKS/jonker.ps.gz` (last access 14/07/1998).

Engelfriet, J. and Treur, J. (1997). A Compositional Reasoning System for Executing Nonmonotonic Theories of Reasoning. In Gabbay, D. M., Kruse, R., Nonnengart, A., and Ohlbach, H. J., editors, *Qualitative and Quantitative Practical Reasoning, Proceedings of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97*, Lecture Notes in Artificial Intelligence 1244, pages 252–266. Springer Verlag. Online available: `ftp://ftp.cs.vu.nl/pub/joeri/FAPR97.ps.Z` (last access 12/07/1998).

Fensel, D. and van Harmelen, F. (1994). A Comparison of Languages which Operationalize and Formalise KADS Models of Expertise. *The Knowledge Engineering Review*, 9:105–146. Online available: `http://www.cs.vu.nl/~frankh/abstracts/KER94.html` (last access 14/07/1998).

Friedrich, G., Gottlob, G., and Stumptner, M. (1990). *Wissensrepräsentation*, chapter 2, pages 21–60, In [Gottlob et al., 1990].

Fuchs, N. E. (1992). Specifications Are (Preferably) Executable. *IEE Software Engineering Journal*, 7(5):323–334. Online available: `http://www.ifi.unizh.ch/staff/fuchs/` (last access 14/07/1998).

Fütty, E. (1997). A Pseudo-Natural Language For Verifying Specifications of Algorithms by Means of an Automaton. Diploma Thesis, Technical University of Vienna, Institute for Computer Languages, Department of Formal Logic Applications.

Goltz, H.-J. and Herre, H. (1990). *Grundlagen der logischen Programmierung.* Akademie-Verlag, Berlin.

Gottlob, G., Frühwirth, T., and Horn, W., editors (1990). *Expertensysteme.* Springers Angewandte Informatik. Springer-Verlag, Wien - New York.

Gottwald, S. (1989). *Mehrwertige Logik - Eine Einführung in Theorie und Anwendungen.* Akademie-Verlag, Berlin.

Guarino, N. (1995). Formal Ontology, Conceptual Analysis and Knowledge Representation. *International Journal of Human and Computer Studies, special issue on The Role of Formal Ontology in the Information Technology*, 43(5–6):625–640. Online available: `http://www.ladseb.pd.cnr.it/Infor/Ontology/Papers/OntologyPapers.html` (last access 14/07/1998).

Guarino, N. and Giaretta, P. (1995). Ontologies and Knowledge Bases: Towards a Terminological Clarification. In Mars, N. J. I., editor, *Towards Very Large Knowledge Bases.* IOS Press. Online available: `http://www.ladseb.pd.cnr.it/Infor/Ontology/Papers/OntologyPapers.html` (last access 03/05/1998).

Hayes, I. J. and Jones, C. B. (1989). Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338. Online available as Technical Report: `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-89-12-1.ps.Z` (last access 28/07/1998).

Hekmatpour, S. and Ince, D. (1988). *Software Prototyping, Formal Methods and VDM.* Addison Wesley Publishing Company, Wokingham, England.

Herre, H. (1993a). Constructive Proof Systems. Report, Institute of Computer Science, Leipzig University, Augustusplatz 10-11, 04109 Leipzig, Germany. Online available: `http://www.informatik.uni-leipzig.de/~herre/deduct/cproofs.ps` (last access 25/05/1998).

Herre, H. (1993b). Semantical Completeness of Model-Based Diagnosis. In *EUROVAV'93 - Proceedings of the European Symposium on the Validation and Verification of Knowledge Based Systems*, pages 217–229. 24-25 March 1993, Palma de Mallorca, Spain.

Herre, H., Jaspars, J., and Wagner, G. (1995). Partial Logics with Two Kinds of Negation as a Foundation for Knowledge-Based Reasoning. Report No. 12, Institute of Computer Science, Leipzig University, Augustusplatz 10-11, 04109 Leipzig, Germany.

Herre, H. and Pearce, D. (1992). Disjunctive Logic Programming, Constructivity and Strong Negation. In [Pearce and Wagner, 1992], pages 391–410.

Hoare, C. (1985). *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd.

Hoare, C. A. R. (1981). The Emperor's Old Clothes. Quoted as in [Bowen and Hinchey, 1995b].

Holloway, C. M. (1997). Why Engineers Should Consider Formal Methods. In *Proceedings of the 16th AIAA/IEEE Digital Avionics Systems Conference*, volume 1, pages 1.3–16 – 1.3.–22. Irvine CA. Online available: `http://shemesh.larc.nasa.gov/cMh/cmh-bio-publications.html/` (last access 14/07/1998).

Holloway, C. M. and Butler, R. W. (1996). Industrial Practice: Impediments to Industrial Use of Formal Methods. *IEEE Computer*, 29(4):25–26. Online available: `http://shemesh.larc.nasa.gov/cMh/cmh-bio-publications.html` (last access 14/07/1998).

Holyer, I. (1991). *Functional Programming with Miranda.* UCL Press, 2nd edition.

Horn, W. (1990). *Knowledge Engineering*, chapter 3, pages 73–89, In [Gottlob et al., 1990].

Hu, D. (1987). *Programmer's Reference Guide to Expert Systems.* Howard W. Sams & Company.

IEEE (1984). IEEE Guide to Software Requirements Specification. ANSI / IEEE Std 830, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA.

Jackson, D. and Wing, J. M. (1996). Lightweight formal methods. *IEEE Computer*, 29(4):22–23. Online available: `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/venari/www/ieee96-roundtable.html` (last access 12/06/1998).

Jonker, C. M. and Treur, J. (1998). Compositional Verification of Multi-Agent Systems: A Formal Analysis of Pro-activeness and Reactiveness. In Langmaack, H., Pnueli, A., and De Roever, W. P., editors, *Proceedings of the International Symposium on Compositionality, COMPOS'97.* Springer Verlag. to appear. Online available: `http://www.cs.vu.nl/~jonker/Papers/compos97.ps` (last access 14/03/1998).

Kleuker, S. (1995). A Gentle Introduction to Specification Engineering using a Case Study in Telecommunications. In Mosses, P. D., Nielsen, M., and Schwartzbach, M. I., editors, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95)*, pages 636–650. Lecture Notes in Computer Science 915, Springer-Verlag. Online available: `http://theoretica.informatik.uni-oldenburg.de/personal/Kleuker.Stephan.html` (last access 26/04/1998).

Kowalski, R. (1979). *Logic for Problem Solving.* North-Holland, Amsterdam.

Leemans, P., Treur, J., and Willems, M. (1993). On the Verification of Knowledge-based Reasoning Modules. Technical Report IR 346, Department of Mathematics and Computer Science, AI Group, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.

Miller, S. P. and Srivas, M. (1995). Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL. Online available: `http://www.csl.sri.com/aamp5.html` (last access 26/07/1998).

NASA (1989). Software Assurance Guidebook. Report SMAP-GB-A201, NASA Goddard Space Flight Center.

NASA (1995). Formal Methods Specifcation and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion. Report NASA-GB-002-95 (Release 1.0), National Aeronautics and Space Administration, Office of Safety and Mission Assurance, Washington, DC 20546, USA. Online available: `http://eis.jpl.nasa.gov/quality/Formal_Methods/` (last access 14/07/1998).

Pearce, D. and Wagner, G., editors (1992). *Logics in AI - European Workshop JELIA'92, Berlin, Germany, September 1992*, Lecture Notes in Artificial Intelligence 633. Springer Verlag, Berlin, Heidelberg.

Preece, A., Batarekh, A., and Shinghal, R. (1992). Verifying Rule-Based Systems. Technical report, Department of Computing Science, King's College, University of Aberdeen, Aberdeen AB24 3UE, Scotland, UK. Online available: `ftp://ftp.csd.abdn.ac.uk/pub/apreece/KER92pt1.ps.Z` (last access 20/03/98).

Reichgelt, H. (1991). *Knowledge Representation: An AI Perspective.* Tutorial Monographs in Cognitive Science. Ablex Publishing Corporation, Norwood, New Jersey.

Rothmaler, P. (1995). *Einführung in die Modelltheorie: Vorlesungen.* Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, Oxford.

Russel, S. and Norvig, P. (1995). *Artificial Intelligence - A Modern Approach.* Prentice-Hall International, Inc.

Sannella, D. (1988). A Survey of Formal Software Development Methods. Technical Report ECS-LFCS-88-56, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh. Online available: `http://www.dcs.ed.ac.uk/home/dts/pub/SannellaDT.html` (last access 14/07/1998).

Sommerville, I. (1992). *Software Engineering.* Addison-Wesley Publishing Company, Wokingham, England, 4th edition.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd., 2nd edition. Online available: `http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html`(last access 26/07/1998).

stdSEM (1997). *stdSEM.* Siemens AG Austria. Draft Version.

Treur, J. (1988). Completeness and definability in diagnostic expert systems. In *Proceedings of ECAI-88*, pages 619–624. Munich, Germany, Pitman Publishing.

Treur, J. and Wetter, T., editors (1993). *Formal Specification of Complex Reasoning Systems.* Ellis Horwood Workshop Series. Ellis Horwood Limited.

Treur, J. and Willems, M. (1994). A Logical Foundation for Verification. In Cohn, A. G., editor, *ECAI 94. Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 745–749, Chichester. John Wiley & Sons Ltd. Online available: `http://www.cs.vu.nl/~wai/pub/1994/Treur_Willems01.ps.Z` (last access 01/06/1998).

Treur, J. and Willems, M. (1995). Formal Notions for Verification of Dynamics of Knowledge-Based Systems. In *Proceedings of the European Symposium on the Validation and Verification of Knowledge-Based Systems, EUROVAV'95*, pages 189–199. Chambery. Online available: `http://www.cs.vu.nl/~wai/pub/1995/Treur_Willems01.ps.Z` (last access 01/06/1998).

Turner, D. (1986). Functional programming as executable specifications. In Hoare, C. A. R. and Shepherdson, J. C., editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice Hall.

Turski, W. M. and Maibaum, T. S. E. (1987). *The Specification of Computer Programs.* International Computer Science Series. Addison-Wesley Publishing Company, Reading, Mass.

Tyrell, H. (1911). Quotation online available: `http://atb-www.larc.nasa.gov/all-quotes-to-date.html` (last access 15/07/1998).

Welt (1996). Am falschen Ende gespart. Ariane-Absturz: Programmierfehler schuld am Milliarden-Debakel? DIE WELT, Ausgabe vom 5. Juli 1996, Axel Springer Verlag, Berlin. Online available: `http://www.welt.de/archiv/1996/07/05/0705s104.htm` (last access 13/06/1998).

Welt (1998). Das bittersüße Lächeln des Bill Gates. Panne bei der Premiere von "Windows 98" in Chicago - Plötzlich stürzte der Computer ab. DIE WELT, Ausgabe vom 22. April 1998, Axel Springer Verlag, Berlin. Online available: `http://www.welt.de/archiv/1998/04/22/0422vm02.htm` (last access 13/06/1998).

Wing, J. M. (1985). Specification Firms: A Vision for the Future. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 241–243. Gloucester Hotel, London (UK).

Wing, J. M. (1995). Hints to Specifiers. Technical Report CMU-CS-95-118R, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213. Online available: `http:/www.cs.cmu.edu/afs/cs.cmu.edu/project/venari/papers/education/paper.ps` (last access 12/06/1998).

Witteveen, C. (1992). Expanding Logic Programs. In [Pearce and Wagner, 1992], pages 373–390.

Yue, K. (1987). What does it mean to say that a specification is complete? In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 42–49. Monterey, California, USA.

Finally, we recommend the following URLs:

`http://www.comlab.ox.ac.uk/archive/formal-methods.html`,

which provides hyperlinks to many on-line repositories of information relevant to formal methods, including some freely available tools, and

`http://www.cs.vu.nl/vakgroepen/ai/projects/desire/`,

the DESIRE homepage, including hyperlinks to on-line available publications.

Index of Symbols

Index

Appendix A

The Telecommunication Example Textual Specification[1]

```
component TopLevel
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         components
            User_2,
            Exchange,
            User_1,
            User_3,
            User_4;
         information links
            a_2,
            b_2,
            a_1,
            b_1,
            a_3,
            b_3,
            a_4,
            b_4;
         initial task information
```

[1]We would have liked to present the complete syntax definition of DESIRE in order to enable the reader to understand the specification more easily. Unfortunately, the publication of the syntax definition is controlled by copyright. However, the syntax definition is part of the course material on the design of multi-agent systems [Brazier et al., 1997], which is handed to each participant of this course.

```
         extent all_p;
      task control contents
   knowledge base TopLevel_task_control
      information types TopLevel_task_control_sig
      contents
         if start
         then next_link_state(a_1, awake)
         and  next_link_state(b_1, awake)
         and  next_link_state(a_2, awake)
         and  next_link_state(b_2, awake)
         and  next_link_state(a_3, awake)
         and  next_link_state(b_3, awake)
         and  next_link_state(a_4, awake)
         and  next_link_state(b_4, awake)
         and  next_component_state(Exchange, awake)
         and  next_component_state(User_1, awake)
         and  next_component_state(User_2, awake)
         and  next_component_state(User_3, awake)
         and  next_component_state(User_4, awake);
   end knowledge base /* TopLevel_task_control */
kernel information
   public kernel information
      knowledge structures
         information type UserIT
            sorts
               USER


            objects
               user_1,
               user_2,
               user_3,
               user_4:  USER

         end information type

         information type SignalIT
            sorts
               SIGNAL_ONE_SORT,
               SIGNAL_TWO_SORT


            objects
               dialling_tone,
               engaged_tone:  SIGNAL_ONE_SORT
      ringing_tone,
      engaged_tone:  SIGNAL_TWO_SORT

end information type

information type InfoIT
   information types
      UserIT,
      SignalIT;
   sorts
      INFO_ELEMENT


   objects
      bell:  INFO_ELEMENT

   functions
      connection_established: USER -> INFO_ELEMENT;
      signal1: SIGNAL_ONE_SORT -> INFO_ELEMENT;
      signal2: SIGNAL_TWO_SORT -> INFO_ELEMENT;

end information type

information type ActionIT
   information types
      UserIT;
   sorts
      ACTION_ELEMENT


   objects
      receiver_lifted,
      receiver_put_down,
      wait_a_moment:  ACTION_ELEMENT

   functions
      phone_number: USER -> ACTION_ELEMENT;

end information type

information type User_Input_Info
   information types
      UserIT;
   relations
      connection_wanted: USER ;
```

```
                  take_call;

            end information type

            information type User_In
               information types
                  InfoIT;
               relations
                  received: INFO_ELEMENT ;

            end information type

            information type User_Out
               information types
                  ActionIT;
               relations
                  send: ACTION_ELEMENT ;

            end information type

            information type User_Internal
               information types
                  UserIT,
                  User_Input_Info;
               relations
                  talk: USER ;

            end information type

      private kernel information
component User_2
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         initial task information
            task control focus tcf;
            extent all_p;
         task control contents
            standard
   kernel information
      public kernel information
      public levels
         level_1 , level_2;
      public level chain
         level_1 < level_2;
      input interface
         level level_1
             information type User_In;
      output interface
         level level_1
             information type User_Out;
   private kernel information
      initial kernel information
      level level_2
         target(tcf, X:OA, determine);
      kernel contents
      knowledge base User_2_local_kbs

         information types
            User_Internal,
            User_Out,
            User_In;
         contents
            /* connection_wanted(user_1); */
            not take_call;

            if   received(bell)
             and take_call
            then send(receiver_lifted);

            if   received(bell)
             and not take_call
            then not send(receiver_lifted);

            if   talk(A:USER)
            then send(receiver_put_down);

            if   received(signal1(engaged_tone))
            then send(receiver_put_down);

            if   received(signal2(engaged_tone))
            then send(receiver_put_down);

            if   connection_wanted(A:USER)
            then send(receiver_lifted);
```

```
               if   not received(signal1(X:SIGNAL_ONE_SORT))
               then send(receiver_put_down);

               if   received(signal1(dialling_tone))
                and connection_wanted(A:USER)
               then send(phone_number(A:USER));

               if   not received(signal2(X:SIGNAL_TWO_SORT))
               then send(receiver_put_down);

               if   received(signal2(ringing_tone))
               then send(wait_a_moment);

               if   received(connection_established(A:USER))
               then talk(A:USER);

               if   not received(connection_established(A:USER))
               then send(receiver_put_down);
         end knowledge base /* User_2_local_kbs */
end component /* User_2 */
component Exchange
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         initial task information
            task control focus tcf;
            extent all_p;
         task control contents
            standard
   kernel information
      public kernel information
         public levels
            level_1 , level_2;
         public level chain
            level_1 < level_2;
         input interface
            level level_1
                information type Exchange_In
                  information types
                     ActionIT;
                  relations
```

```
                    received: ACTION_ELEMENT * USER ;
                    engaged: USER ;
                    bell: USER ;

                end information type

            output interface
               level level_1
                   information type Exchange_Out
                     information types
                        InfoIT;
                     relations
                        send: INFO_ELEMENT * USER ;

                end information type

         private kernel information
            initial kernel information
            level level_2
               target(tcf, X:OA, determine);
            kernel contents
            knowledge base Exchange_local_kbs

               information types
                  Exchange_Out,
                  Exchange_In;
               contents
                  if   received(receiver_lifted, A:USER)
                   and not bell(A:USER)
                  then send(signal1(dialling_tone), A:USER);

                  if   received(phone_number(B:USER), A:USER)
                   and engaged(B:USER)
                   and not bell(B:USER)
                  then send(signal2(engaged_tone), A:USER);

                  if   received(phone_number(B:USER), A:USER)
                   and not engaged(B:USER)
                  then send(signal2(ringing_tone), A:USER)
                   and send(bell, B:USER);

                  if   received(phone_number(B:USER), A:USER)
                   and engaged(A:USER)
                   and bell(B:USER)
                   and received(receiver_lifted, B:USER)
               then send(connection_established(A:USER), B:USER)
                and send(connection_established(B:USER), A:USER);

               if   received(phone_number(B:USER), A:USER)
                and engaged(A:USER)
                and send(signal2(ringing_tone), A:USER)
                and not received(receiver_lifted, B:USER)
               then not send(connection_established(B:USER), A:USER);
         end knowledge base /* Exchange_local_kbs */
end component /* Exchange */
component User_1
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         initial task information
            task control focus tcf;
            extent all_p;
         task control contents
            standard
   kernel information
      public kernel information
         public levels
            level_1 , level_2;
         public level chain
            level_1 < level_2;
         input interface
            level level_1
                information type User_In;
         output interface
            level level_1
                information type User_Out;
      private kernel information
         initial kernel information
         level level_2
            target(tcf, X:OA, determine);
         kernel contents
         knowledge base User_1_local_kbs

            information types
               User_Internal,
               User_Out,
            User_In;
         contents
            connection_wanted(user_3);
            take_call;

            if   received(bell)
             and take_call
            then send(receiver_lifted);

            if   received(bell)
             and not take_call
            then not send(receiver_lifted);

            if   talk(A:USER)
            then send(receiver_put_down);

            if   received(signal1(engaged_tone))
            then send(receiver_put_down);

            if   received(signal2(engaged_tone))
            then send(receiver_put_down);

            if   connection_wanted(A:USER)
            then send(receiver_lifted);

            if   not received(signal1(X:SIGNAL_ONE_SORT))
            then send(receiver_put_down);

            if   received(signal1(dialling_tone))
             and connection_wanted(A:USER)
            then send(phone_number(A:USER));

            if   not received(signal2(X:SIGNAL_TWO_SORT))
            then send(receiver_put_down);

            if   received(signal2(ringing_tone))
            then send(wait_a_moment);

            if   received(connection_established(A:USER))
            then talk(A:USER);

            if   not received(connection_established(A:USER))
            then send(receiver_put_down);
      end knowledge base /* User_1_local_kbs */
end component /* User_1 */
```

```
component User_3
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         initial task information
            task control focus tcf;
            extent all_p;
         task control contents
            standard
   kernel information
      public kernel information
         public levels
            level_1 , level_2;
         public level chain
            level_1 < level_2;
         input interface
            level level_1
                information type User_In;
         output interface
            level level_1
                information type User_Out;
      private kernel information
         initial kernel information
         level level_2
            target(tcf, X:OA, determine);
         kernel contents
         knowledge base User_3_local_kbs

            information types
               User_Internal,
               User_Out,
               User_In;
            contents
               /* connection_wanted(user_2); */
               take_call;

               if   received(bell)
                and take_call
               then send(receiver_lifted);

               if   received(bell)
```

```
                 and not take_call
                then not send(receiver_lifted);

                if   talk(A:USER)
                then send(receiver_put_down);

                if   received(signal1(engaged_tone))
                then send(receiver_put_down);

                if   received(signal2(engaged_tone))
                then send(receiver_put_down);

                if   connection_wanted(A:USER)
                then send(receiver_lifted);

                if   not received(signal1(X:SIGNAL_ONE_SORT))
                then send(receiver_put_down);

                if   received(signal1(dialling_tone))
                 and connection_wanted(A:USER)
                then send(phone_number(A:USER));

                if   not received(signal2(X:SIGNAL_TWO_SORT))
                then send(receiver_put_down);

                if   received(signal2(ringing_tone))
                then send(wait_a_moment);

                if   received(connection_established(A:USER))
                then talk(A:USER);

                if   not received(connection_established(A:USER))
                then send(receiver_put_down);
         end knowledge base /* User_3_local_kbs */
end component /* User_3 */
component User_4
   task information
      public task information
         task control foci
            tcf;
         evaluation criteria
            tcf;
      private task information
         initial task information
            task control focus tcf;
         extent all_p;
      task control contents
         standard
kernel information
   public kernel information
      public levels
         level_1 , level_2;
      public level chain
         level_1 < level_2;
      input interface
         level level_1
             information type User_In;
      output interface
         level level_1
             information type User_Out;
   private kernel information
      initial kernel information
      level level_2
         target(tcf, X:OA, determine);
      kernel contents
      knowledge base User_4_local_kbs

         information types
            User_Internal,
            User_Out,
            User_In;
         contents
            connection_wanted(user_2);
            take_call;

            if   received(bell)
             and take_call
            then send(receiver_lifted);

            if   received(bell)
             and not take_call
            then not send(receiver_lifted);

            if   talk(A:USER)
            then send(receiver_put_down);

            if   received(signal1(engaged_tone))
            then send(receiver_put_down);

            if   received(signal2(engaged_tone))
```

```
               then send(receiver_put_down);

               if   connection_wanted(A:USER)
               then send(receiver_lifted);

               if   not received(signal1(X:SIGNAL_ONE_SORT))
               then send(receiver_put_down);

               if   received(signal1(dialling_tone))
                and connection_wanted(A:USER)
               then send(phone_number(A:USER));

               if   not received(signal2(X:SIGNAL_TWO_SORT))
               then send(receiver_put_down);

               if   received(signal2(ringing_tone))
               then send(wait_a_moment);

               if   received(connection_established(A:USER))
               then talk(A:USER);

               if   not received(connection_established(A:USER))
               then send(receiver_put_down);
         end knowledge base /* User_4_local_kbs */
end component /* User_4 */
private link a_2  : epistemic - object
   domain User_2
      level level_2
   co-domain Exchange
         level level_1
   sort links identity

   object links identity

   term links identity

   atom links
    (true(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_2)):
    <<true, true>>;

    (false(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_2)):
    <<true, false>>;

    (known(received(bell)), bell(user_2)) : <<false, false>>;
    (true(received(bell)), bell(user_2)) : <<true, true>>;

    (true(send(receiver_lifted)), engaged(user_2)) :
    <<true, true>, <false, false>>;

   end link /* a_2 */
private link b_2  : object - assumption
   domain Exchange
      level level_1
   co-domain User_2
         level level_2
   sort links identity

   object links identity

   term links identity

   atom links
    (send(X:INFO_ELEMENT, user_2),
      assumption(received(X:INFO_ELEMENT), pos)) :
    <<true, true>>;

    (send(X:INFO_ELEMENT, user_2),
      assumption(received(X:INFO_ELEMENT), neg)) :
    <<false, true>>;

   end link /* b_2 */
private link a_1  : epistemic - object
   domain User_1
      level level_2
   co-domain Exchange
         level level_1
   sort links identity

   object links identity

   term links identity

   atom links
    (true(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_1)):
    <<true, true>>;

    (false(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_1)):
    <<true, false>>;
    (known(received(bell)), bell(user_1)) : <<false, false>>;

    (true(received(bell)), bell(user_1)) : <<true, true>>;

    (true(send(receiver_lifted)), engaged(user_1)) :
    <<true, true>, <false, false>>;
   end link /* a_1 */
private link b_1  : object - assumption
   domain Exchange
      level level_1
   co-domain User_1
         level level_2
   sort links identity

   object links identity

   term links identity

   atom links
    (send(X:INFO_ELEMENT, user_1),
      assumption(received(X:INFO_ELEMENT), pos)) :
    <<true, true>, <false, false>>;

    (send(X:INFO_ELEMENT, user_1),
      assumption(received(X:INFO_ELEMENT), neg)) :
    <<false, true>>;
   end link /* b_1 */
private link a_3  : epistemic - object
   domain User_3
      level level_2
   co-domain Exchange
         level level_1
   sort links identity

   object links identity

   term links identity

   atom links
    (true(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_3)):
    <<true, true>>;

    (false(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_3)):
    <<true, false>>;
    (known(received(bell)), bell(user_3)) : <<false, false>>;

    (true(received(bell)), bell(user_3)) : <<true, true>>;

    (true(send(receiver_lifted)), engaged(user_3)) :
    <<true, true>, <false, false>>;
   end link /* a_3 */
private link b_3  : object - assumption
   domain Exchange
      level level_1
   co-domain User_3
         level level_2
   sort links identity

   object links identity

   term links identity

   atom links
    (send(X:INFO_ELEMENT, user_3),
      assumption(received(X:INFO_ELEMENT), pos)) :
    <<true, true>, <false, false>>;

    (send(X:INFO_ELEMENT, user_3),
      assumption(received(X:INFO_ELEMENT), neg)) :
    <<false, true>>;
   end link /* b_3 */
private link a_4  : epistemic - object
   domain User_4
      level level_2
   co-domain Exchange
         level level_1
   sort links identity

   object links identity

   term links identity

   atom links
    (true(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_4)):
    <<true, true>>;

    (false(send(X:ACTION_ELEMENT)),received(X:ACTION_ELEMENT,user_4)):
    <<true, false>>;
    (known(received(bell)), bell(user_4)) : <<false, false>>;

    (true(received(bell)), bell(user_4)) : <<true, true>>;

    (true(send(receiver_lifted)), engaged(user_4)) :
    <<true, true>, <false, false>>;
   end link /* a_4 */
private link b_4  : object - assumption
   domain Exchange
      level level_1
   co-domain User_4
         level level_2
   sort links identity

   object links identity

   term links identity

   atom links
    (send(X:INFO_ELEMENT, user_4),
      assumption(received(X:INFO_ELEMENT), pos)) :
    <<true, true>>;

    (send(X:INFO_ELEMENT, user_4),
      assumption(received(X:INFO_ELEMENT), neg)) :
    <<false, true>>;
   end link /* b_4 */
end component /* TopLevel */
```

Appendix B

The Telecommunication Example Reasoning Trace

```
        Evaluation mode:fail_contra


DESIRE General Manager System in prolog,   version 2.92a  (CVSDATE)
      options: stack user update special_sorts


Read configuration from "/usr/local/ai/hd/lib/prolog/config.gm"



HD // Part version 1.26

desimpl.pl compiled into dbdefs, 0.25 sec, 87,096 bytes.


partial_evaluation(TopLevel,determine_all_output_atoms,all_p,succeeded)
partial_evaluation(TopLevel,determine_all_output_atoms,all_p,succeeded)
        awake composed component "TopLevel" -> BUSY

 TASK CONTROL STEP of "TopLevel"

  UPDATING TASK CONTROL INPUT
    ESTABLISHED previous_own_component_state(idle) by TCloop
    ESTABLISHED not own_component_state(idle) by TCloop
    ESTABLISHED start by TCloop

  ACTIVATING TASK CONTROL KBS of "TopLevel"
  REASONING TopLevel    (step,all-p)
    target(X:OA,determine)
    derived(next_link_state(a_1, awake))       by rule r41
    derived(next_link_state(b_1, awake))       by rule r41
    derived(next_link_state(a_2, awake))       by rule r41
    derived(next_link_state(b_2, awake))       by rule r41
    derived(next_link_state(a_3, awake))       by rule r41
    derived(next_link_state(b_3, awake))       by rule r41
    derived(next_link_state(a_4, awake))       by rule r41
    derived(next_link_state(b_4, awake))       by rule r41
    derived(next_component_state(Exchange, awake))       by rule r41
    derived(next_component_state(User_1, awake))       by rule r41
    derived(next_component_state(User_2, awake))       by rule r41
    derived(next_component_state(User_3, awake))       by rule r41
    derived(next_component_state(User_4, awake))       by rule r41
        cannot be established stop
        cannot be established not stop

main menu

    continue  :c
    quit      :q
    debug     :d
    pr lastm  :p
    pr modx   :m
    output    :o
    pr file   :f
    stepping  :s
enter choice:s

control menu

    leap      :l
    spy modx  :s
    spy mod#  :p
    spy atom  :a
    unspy m   :u
    unspy at  :n
    leap tm.  :t
    quit      :q
    mainmenu  :m
enter choice:l

  LINK a_1
    BUFFERED
```

```
      not bell(user_1)
      not engaged(user_1)

 LINK b_1

 LINK a_2
   BUFFERED
      not bell(user_2)
      not engaged(user_2)

 LINK b_2

 LINK a_3
   BUFFERED
      not bell(user_3)
      not engaged(user_3)

 LINK b_3

 LINK a_4
   BUFFERED
      not bell(user_4)
      not engaged(user_4)

 LINK b_4

END TASK CONTROL STEP of "TopLevel"


TASK CONTROL STEP of "TopLevel"

 UPDATING TASK CONTROL INPUT
   ESTABLISHED not previous_own_component_state(idle) by TCloop
   RETRACTED next_component_state(User_4, awake)
            (revision)
   RETRACTED next_component_state(User_3, awake)
            (revision)
   RETRACTED next_component_state(User_2, awake)
            (revision)
   RETRACTED next_component_state(User_1, awake)
            (revision)
   RETRACTED next_component_state(Exchange, awake)
            (revision)
   RETRACTED next_link_state(b_4, awake)
            (revision)
```

```
    RETRACTED next_link_state(a_4, awake)
             (revision)
    RETRACTED next_link_state(b_3, awake)
             (revision)
    RETRACTED next_link_state(a_3, awake)
             (revision)
    RETRACTED next_link_state(b_2, awake)
             (revision)
    RETRACTED next_link_state(a_2, awake)
             (revision)
    RETRACTED next_link_state(b_1, awake)
             (revision)
    RETRACTED next_link_state(a_1, awake)
             (revision)
    ESTABLISHED not start by TCloop

  ACTIVATING TASK CONTROL KBS of "TopLevel"

  REASONING TopLevel    (step,all-p)
    target(X:OA,determine)
        cannot be established stop
        cannot be established not stop

 END TASK CONTROL STEP of "TopLevel"


  ACTIVATING "Exchange" (tcf,all-p)

    UPDATING INTERFACE (of) Exchange
    ESTABLISHED
       not bell(user_1)
       not engaged(user_1)
       not bell(user_2)
       not engaged(user_2)
       not bell(user_3)
       not engaged(user_3)
       not bell(user_4)
       not engaged(user_4)
  REASONING Exchange    (tcf,all-p)
    target(X:OA,determine)

termination(Exchange, tcf, succeeded)


  LINK b_1
  LINK b_2

  LINK b_3

  LINK b_4

  ACTIVATING "User_1" (tcf,all-p)

  REASONING User_1    (tcf,all-p)
    target(X:OA,determine)
    derived(connection_wanted(user_3))       by rule r60
    derived(send(receiver_lifted))       by rule r67

termination(User_1, tcf, succeeded)


  LINK a_1
    BUFFERED
       received(receiver_lifted, user_1)
       engaged(user_1)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_2" (tcf,all-p)

  REASONING User_2    (tcf,all-p)
    target(X:OA,determine)

termination(User_2, tcf, succeeded)


  LINK a_2
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_3" (tcf,all-p)

  REASONING User_3    (tcf,all-p)
    target(X:OA,determine)

termination(User_3, tcf, succeeded)


  LINK a_3
```

```
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_4" (tcf,all-p)

  REASONING User_4    (tcf,all-p)
    target(X:OA,determine)
    derived(connection_wanted(user_2))       by rule r87
    derived(send(receiver_lifted))       by rule r94

termination(User_4, tcf, succeeded)


  LINK a_4
    BUFFERED
       received(receiver_lifted, user_4)
       engaged(user_4)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "Exchange" (tcf,all-p)

    UPDATING INTERFACE (of) Exchange
       received(receiver_lifted, user_1)
       engaged(user_1)
       received(receiver_lifted, user_4)
       engaged(user_4)
  REASONING Exchange    (tcf,all-p)
    target(X:OA,determine)
    derived(send(signal1(dialling_tone), user_4))       by rule r55
    derived(send(signal1(dialling_tone), user_1))       by rule r55

termination(Exchange, tcf, succeeded)


  LINK b_1
    BUFFERED
      assumption(received(signal1(dialling_tone)),pos)   true
       received(signal1(dialling_tone))
primitive component "User_1" -> BUSY
rescheduled awake link destination "User_1"

  LINK b_2

  LINK b_3
  LINK b_4
    BUFFERED
      assumption(received(signal1(dialling_tone)),pos)   true
       received(signal1(dialling_tone))
primitive component "User_4" -> BUSY
rescheduled awake link destination "User_4"

  ACTIVATING "User_1" (tcf,all-p)

    UPDATING INTERFACE (of) User_1
       received(signal1(dialling_tone))
  REASONING User_1    (tcf,all-p)
    target(X:OA,determine)
    derived(send(phone_number(user_3)))       by rule r69

termination(User_1, tcf, succeeded)


  LINK a_1
    BUFFERED
       received(phone_number(user_3), user_1)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_4" (tcf,all-p)

    UPDATING INTERFACE (of) User_4
       received(signal1(dialling_tone))
  REASONING User_4    (tcf,all-p)
    target(X:OA,determine)
    derived(send(phone_number(user_2)))       by rule r96

termination(User_4, tcf, succeeded)


  LINK a_4
    BUFFERED
       received(phone_number(user_2), user_4)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "Exchange" (tcf,all-p)

    UPDATING INTERFACE (of) Exchange
```

```
       received(phone_number(user_3), user_1)
       received(phone_number(user_2), user_4)
  REASONING Exchange    (tcf,all-p)
    target(X:OA,determine)
    derived(send(signal2(ringing_tone), user_4))       by rule r57
    derived(send(bell, user_2))       by rule r57
    derived(send(signal2(ringing_tone), user_1))       by rule r57
    derived(send(bell, user_3))       by rule r57

termination(Exchange, tcf, succeeded)



  LINK b_1
    BUFFERED
      assumption(received(signal2(ringing_tone)),pos)   true
       received(signal2(ringing_tone))
primitive component "User_1" -> BUSY
rescheduled awake link destination "User_1"

  LINK b_2
    BUFFERED
      assumption(received(bell),pos)   true
       received(bell)
primitive component "User_2" -> BUSY
rescheduled awake link destination "User_2"

  LINK b_3
    BUFFERED
      assumption(received(bell),pos)   true
       received(bell)
primitive component "User_3" -> BUSY
rescheduled awake link destination "User_3"

  LINK b_4
    BUFFERED
      assumption(received(signal2(ringing_tone)),pos)   true
       received(signal2(ringing_tone))
primitive component "User_4" -> BUSY
rescheduled awake link destination "User_4"

  ACTIVATING "User_1" (tcf,all-p)

    UPDATING INTERFACE (of) User_1
       received(signal2(ringing_tone))
  REASONING User_1    (tcf,all-p)
```

```
    target(X:OA,determine)
    derived(send(wait_a_moment))       by rule r71

termination(User_1, tcf, succeeded)


  LINK a_1
    BUFFERED
       received(wait_a_moment, user_1)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_2" (tcf,all-p)

    UPDATING INTERFACE (of) User_2
       received(bell)
  REASONING User_2    (tcf,all-p)
    target(X:OA,determine)
    derived(not take_call)       by rule r42
    derived(not send(receiver_lifted))       by rule r44

termination(User_2, tcf, succeeded)


  LINK a_2
    BUFFERED
       not received(receiver_lifted, user_2)
       bell(user_2)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_3" (tcf,all-p)

    UPDATING INTERFACE (of) User_3
       received(bell)
  REASONING User_3    (tcf,all-p)
    target(X:OA,determine)
    derived(take_call)       by rule r74
    derived(send(receiver_lifted))       by rule r75

termination(User_3, tcf, succeeded)


  LINK a_3
    BUFFERED
```

```
       received(receiver_lifted, user_3)
       bell(user_3)
       engaged(user_3)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_4" (tcf,all-p)

    UPDATING INTERFACE (of) User_4
       received(signal2(ringing_tone))
  REASONING User_4    (tcf,all-p)
    target(X:OA,determine)
    derived(send(wait_a_moment))       by rule r98

termination(User_4, tcf, succeeded)


  LINK a_4
    BUFFERED
       received(wait_a_moment, user_4)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "Exchange" (tcf,all-p)

    UPDATING INTERFACE (of) Exchange
       received(wait_a_moment, user_1)
       not received(receiver_lifted, user_2)
       bell(user_2)
       received(receiver_lifted, user_3)
       bell(user_3)
    RETRACTED send(bell, user_3)
             (revision)
    RETRACTED send(signal2(ringing_tone), user_1)
             (revision)
       engaged(user_3)
       received(wait_a_moment, user_4)
  REASONING Exchange    (tcf,all-p)
    target(X:OA,determine)
    derived(send(connection_established(user_1), user_3)) by rule r58
    derived(send(connection_established(user_3), user_1)) by rule r58
    derived(not send(connection_established(user_2), user_4))
                                                          by rule r59

termination(Exchange, tcf, succeeded)
```

```
  LINK b_1
    BUFFERED
      assumption(received(connection_established(user_3)),pos)   true
       received(connection_established(user_3))
primitive component "User_1" -> BUSY
rescheduled awake link destination "User_1"

  LINK b_2

  LINK b_3
    BUFFERED
      assumption(received(connection_established(user_1)),pos)   true
       received(connection_established(user_1))
primitive component "User_3" -> BUSY
rescheduled awake link destination "User_3"

  LINK b_4
    BUFFERED
      assumption(received(connection_established(user_2)),neg)   true
       not received(connection_established(user_2))
primitive component "User_4" -> BUSY
rescheduled awake link destination "User_4"

  ACTIVATING "User_1" (tcf,all-p)

    UPDATING INTERFACE (of) User_1
       received(connection_established(user_3))
  REASONING User_1    (tcf,all-p)
    target(X:OA,determine)
    derived(talk(user_3))      by rule r72
    derived(send(receiver_put_down))      by rule r64

termination(User_1, tcf, succeeded)


  LINK a_1
    BUFFERED
       received(receiver_put_down, user_1)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_3" (tcf,all-p)
```

```
    UPDATING INTERFACE (of) User_3
       received(connection_established(user_1))
  REASONING User_3    (tcf,all-p)
    target(X:OA,determine)
    derived(talk(user_1))       by rule r85
    derived(send(receiver_put_down))       by rule r77

termination(User_3, tcf, succeeded)


  LINK a_3
    BUFFERED
       received(receiver_put_down, user_3)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "User_4" (tcf,all-p)

    UPDATING INTERFACE (of) User_4
       not received(connection_established(user_2))
  REASONING User_4    (tcf,all-p)
    target(X:OA,determine)
    derived(send(receiver_put_down))       by rule r100

termination(User_4, tcf, succeeded)


  LINK a_4
    BUFFERED
       received(receiver_put_down, user_4)
primitive component "Exchange" -> BUSY
rescheduled awake link destination "Exchange"

  ACTIVATING "Exchange" (tcf,all-p)

    UPDATING INTERFACE (of) Exchange
       received(receiver_put_down, user_1)
       received(receiver_put_down, user_3)
       received(receiver_put_down, user_4)
  REASONING Exchange    (tcf,all-p)
    target(X:OA,determine)

termination(Exchange, tcf, succeeded)
```

```
  LINK b_1

  LINK b_2

  LINK b_3

  LINK b_4
                awake composed component "TopLevel" -> NON BUSY

Nothing to do, exiting
```

Appendix C

System Configuration

The specification of the telecommunication network was developed using

- DESTOOL version 1.50,
- DESTOOL version 1.61,
- DESTOOL version 1.79, and finally
- DESTOOL version 1.84

on an IBM-compatible PC with a 6x86 P150+ CPU and 64MB RAM running

- S.u.S.E. Linux 5.0 - Kernel 2.0.30 (`www.suse.de`),
- SWI-Prolog and xpce version 4.9.7. (Online available: `http://swi.psy.uva.nl/projects/xpce/home.html` (last access 27/07/1998)), and
- gcc version 2.7.2.1.

The DESIRE software environment can be installed for non-commercial applications only. Please contact Lourens van der Meij `<lourens@vlet.cs.vu.nl>` for more information.

This thesis was typeset using LaTeX2ε, as it comes along with teTeX (version 0.4), and the `oz.sty` package (Online available: `http://svrc.it.uq.edu.au/Object-Z/pages/latex.html` (last access 27/07/1998)).

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 29. Juli 1998 Ralph Miarka