CABS: A Case-Based and Graphical Requirements Capture, Formalisation and Verification System.

Peter J. Funk

Ph.D.

University of Edinburgh
March 1998

Abstract

The use of formal specifications based on varieties of mathematical logic is becoming common in the process of designing and implementing safety critical systems and practices for hardware design. Formal methods are usually intended to include in the specification, all the important details of the final system in the specification, with the aim of proving that the specification possesses certain properties and lacks other unwanted properties. In large, complex systems, this task requires sophisticated theorem proving, which can be difficult and complicated. Telecommunications systems are large and complex, making detailed formal specification impractical given current technology. However, formal “sketches” of the behaviours the services provide can be produced, and these can be very helpful in locating which service might be relevant to a given problem.

This thesis describes CABS, a case-based approach that uses coarse-grained graphical requirements specification sketches, to outline the basic behaviour of the system's functional modules (called services), thereby allowing us to identify, re-use and adapt requirements (from cases stored in a library), to construct new cases. The matching algorithm identifies similar behaviour between the input examples and the cases stored in the case library. By using cases that have already been tested, integrated and implemented, less effort is needed to produce requirements specifications on a large scale. Using a hypothetical telecommunications system as an example, it will be shown that a comparatively simple logic can be used to capture coarse-grained behaviour and how a case-based approach benefits from this. The input from the examples is used both to identify the cases whose behaviour corresponds most closely to the designer's intentions, and also in the process of adapting, validating and, finally, verifying the proposed solution against the examples.

Contents

List of Figures

List of Tables

Chapter:

1. Introduction

Requirements play an important role throughout system development and the lack of validated, verified and easily accessible requirements has been suggested to be one of the main areas of focus in requirements engineering [Bubenko 95]. State-based modelling is one of the ways used in practice to tackle this. A conventional use for state-based modelling in telecommunications services is in describing the precise behaviour of those services. Unfortunately this form of detailed modelling is prohibitively expensive for realistically sized problems. This thesis describes a different role for state based models - not as precise behavioural descriptions but as "sketches" of key features required by a client. These features are used by a case-based reasoning (CBR) system to suggest existing services which might be adapted to the clients' needs.

The core of the thesis is in the CBR matching system but, in order to provide this, we need to solve a set of subsidiary problems: how to describe required behaviours at an appropriate level of detail (just sufficient to discriminate cases); how to refine the input examples if (as is likely) the first draft of this isn't sufficient; how to test if the required behaviour is included in the proposed and selected solution (by simulation and automated verification identifying where the behaviour differs).

1.1 Functional Requirements, Problems and Benefits

The application domain that has been chosen is telecommunications services and, in particular, telephone services. Telephone services are a non-trivial domain where hundreds of different services and variants of services have been implemented in telecommunications switches and where the number of services and demand for new services is increasing. Most big telecommunications companies have tried to apply formal methods to the specification of telecommunications services, due to the stringent requirements for reliability in telephone networks and, in particular, the demand that no additional functionality should affect the basic functionality, such as calling an emergency service. The application domain is in fact so complex and large, that formal requirements specifications have not been applied in practice. In the 1970s, research started in earnest on formally specifying systems and, by the late seventies and early eighties, industry assumed that research progress was sufficient to bring the knowledge and research results into practical use [Hsia, Davis, Kung, 93]. A number of large scale projects were initiated to introduce formal requirements specifications. In most areas, formal methods did not deliver on their early promise [Zave 91]; a number of explanations for this are given in [Hall 90].

The size of the application domain (functional requirements of telephone services) used for reference in this research, is large enough to be non-trivial and to confront a number of issues arising from a full scale application. Seventeen behavioural outlines of telecommunications services (the behaviour seen from the point of view of a phone user without describing any of the complex behaviour occurring in the telecommunications network) have been formalised and used in evaluation. Each service contains a number of transition rules[1], representing the behaviour of the service, and a number of term definitions connecting the specification of the system to its environment.

[1] Transition rules and term definitions will be explained in Chapter 5.

Mainstream requirements capture tools in telecommunications are informal and methodology centred and do not require any particular notations of formalisms (Ericssons[2] PROPS method for example). In the state of the art requirements capturing tool Rational Rose *use-cases* are used to capture an initial sketch of the behavioural requirements. Rational Rose will be introduced at Ericsson to be used as their main requirements capturing tool. Use-cases capture examples of behaviour. Different notations can be used in the method depending on the application domain and user preferences. For example the unified modelling language, UML, is recommended for static modelling of objects and their relations. Informal requirements in telecommunications have in a number of cases been shown to be expensive (for an unconfirmed example se Section 2.3.1), leading to legal problems over the exact meaning of the informal requirements once a functionality is delivered that does not meet the customers expectations. Informal requirements have also led to misunderstandings in the design and implementation, causing serious problems, faults and down time in telecommunications systems (an example of this is given in Chapter 2). It has been claimed that poor quality software is costing UK industry £2000 million every year, and that many failures have their roots in informal requirements and specifications [Schofield 92].

These problems are the main reasons for the interest in formal methods from major telecommunications companies. Formal specifications based on varieties of mathematical logic are being used more frequently in the design of safety critical systems. Formal methods are usually intended to include all important details of the final system in the specification, with the aim of proving that it possesses certain properties and does not exhibit other unwanted properties. Fully formalised requirements are today mostly used for well isolated problems where the number of states are less than a few thousand, for example used in protocol specifications. It is believed that a wider use of formal methods would reduce problems caused by textual requirements and formal specifications are successfully used for

[2] Ericsson is one of the largest communications supplier for network operators, service providers, enterprises and customers and employees more than 100,000 people in 140 countries.

many different tasks, but limitations in tools and graphical notations limit their use today [Jensen 97]. Telecommunications services in general include hundreds of thousands of states and have been resistant to such rigorous methods. Isolated parts of the behaviour of services have been formalised but even here the number of states has been exceeding the limit of performance of available tools [Capellmann, Christensen, Herzog 98]. Major telecommunications companies started investigating formal methods thoroughly in the eighties ([Zave 91], [Funk, Reichman 90] [Kelly, Nonnenman 91]) but none use formal methods routinely in service and feature requirements. In large, complex systems, this task requires sophisticated theorem proving, which can be difficult and complicated. Telecommunications systems are large and complex, making their detailed formal specification impractical with current technology. Sometimes, the formalism or combination of formalisms is so complex that even experts in formal methods find it difficult formally to represent some aspects of the system to be specified [Mataga, Zave 93]. Some researchers doubt that existing methods will scale up to such complex systems [Heimdahl, Leveson 95].

1.1.1 Previous Experience and Domain Related Problems

In 1985 Ericsson Research & Development started to explore formal methods in detail. In autumn 1985 I was employed in an industrial project at Ericsson at the department of computer science involved with the task of bringing formal specification into use in industry for the specification of computer based systems. During the following six years, we collaborated with the University of Stockholm, the University of Uppsala, Stanford University and the Swedish Institute of Computer Science (SICS), amongst others. The main task was to develop a formal notation and implement a prototype to explore the use of formal methods in industrial applications such as telephone service requirements. A large coarse grained formal specification of sixteen telephone services[3] was made [Funk, Raichman, 90] where the main behavioural requirements of the services where captured. Most effort was put into exploring and choosing a suitable formal notation expressive enough to capture these requirements but not more expressive than necessary, to enable simulation and analysis of the requirements. The chosen logical notation for this research is based on the results used in the formal methods project at Ericsson (see Appendix A and [Funk 93]). The logical notation was expressive enough to be used in formalising coarse grained telecommunications service specifications on a high abstraction[4] level but, for different reasons (lack of resources being one), we had not addressed sufficiently:

Re-use and modification of previously specified services or parts of services. The most frequent situation in the domain of telecommunications service specifications is the specification of services similar to previous ones.

The issue of iteratively refining and incrementally extending requirements that originally where sketchy, incomplete and contained errors.

End users with background in systems design and programming did not accept the idea of using the formal notation to specify services at Ericsson. Their interest in formal methods was high until they where confronted with logical axioms. Even showing slides with logical or mathematical notations drastically reduced any interest earlier shown.

[3] A telephone *service* (such as divert calls) in Europe is called a *feature* in the United States. *Service* is used here and the word *feature* always refers to features in case-based reasoning (as described in Chapter 6).

[4] At the beginning we had hoped to define a formal notation expressive enough to capture the complete detailed behaviour of telecommunications services (concurrently occurring events, parallelisms, timing constraints, nondeterminism, etc.), but realised that this had to be abandoned if we at the same time wanted to have access to simulation and powerful analysis methods.

These factors contributed to the cancellation of the project in 1992 (started in 1985, about 40 man years where invested). A related project implementing a full scale theorem prover for service requirements specifications with a graphical interface [Ridley, Höök, Engstedt, Lapins, Lindroos 97], started in 1993 and was successfully completed technically but cancelled in 1997. The logical notation and the theorem prover was implemented in C++ and Erlang[5] and proved to be sufficient for full scale use for service specifications. A graphical notation was introduced in parallel with the textual notation (the notation is based on decision trees and bears no similarities to the one used in this research) and required knowledge in logic and formal methods which turned out to be more than any users were prepared to accept. Also, the problem of re-use and refinement of service sketches was not further explored (and was not a defined part of this project). Ericsson is at the moment not actively involved with formal methods for requirements specifications of telecommunications services.

1.2 Capturing and Formalising Requirements

In this research, some of the main features of traditional "strong" use of formal methods are sacrificed in the requirements capture process: *we do not require the specification to be correct and complete from the start.* In many application domains, including the telecommunications domain, original requirements are often sketchy ideas and it is not always justified to force the user to give complete and correct requirements from the start [Cybulski 96]. Requirements capture is seen as an iterative refinement process of some initial requirements that are incomplete (lacking details, missing behaviour for different situations such as odd and unusual situations) and may contain flaws (reflecting a naive or an unclear idea of the functionality that needs refinement).

[5] Erlang is a concurrent functional programming language developed at Ericsson and widely spread both for prototype programming, complex system implementations and in education and for research at universities.

This approach to formal methods has a number of advantages such as: the rapid creation of an outline of the new behaviour which is used for identifying similar behaviour, then simulated and refined until the formalised behaviour reflects a required functionality. This approach is consistent with what has been called a *lightweight approach to formal methods* [Hesketh, Robertson, Fuchs, Bundy 95], where the formal notation has been chosen to be as simple as possible and just expressive enough to outline the main behaviour required[6]. The simplicity of the logical notation enables automated manipulation, translation and comparison between behavioural requirements specifications and formalised input examples. This enables re-use if the requirements of services, previously specified and subsequently implemented, are stored in a case library.

1.2.1 Identifying Similar Behaviour

The main focus for this research is on identifying similar behaviour to enable re-use of previously specified requirements or parts of requirements. In addition to re-use, iterative refinement, enabling the user to sketch out the required behaviour without giving all the details from the start is included, in contrast with the common approach within formal methods where the user is expected to produce complete and correct requirements from the beginning. The aims of this prototype implementation[7] are mainly:

To provide a platform where the identification of similar behaviour can be evaluated (evaluated in Chapter 8).

[6] The notation is purposely not expressive enough to represent the full complexity of telecommunications requirements specifications, such as concurrence, internal communication, etc.

[7] The system has been implemented in LPA-Prolog (Macintosh/Windows) and the non-graphical parts are also compatible with SICSTUS-Prolog.

To put the matching and re-use in context of case-based reasoning where an initial sketch of some wanted behaviour is used for identification of similar behaviour that may be re-used (evaluated in Chapter 8), refined, validated and verified.

1.3 A Scenario Showing how CABS may be Used

To give a framework for understanding CABS (Case Based Requirements Specification System) and to put the different chapters in context, I will give a brief example of how someone might use a full implementation of CABS (including some of the extensions proposed in Chapter 9). I will not dwell in this description on what has been implemented and what is left for further work. By reading the rest of the thesis, it will be clear what has been explored in depth and implemented in this research and what has been left for further improvements. Figure 1.1 gives an overview of how an idea can be taken to a full specification (se Section 1.3.1). At present, the first formal level used in telecommunications requirements is mostly SDL (a programming language with graphical and textual parts often used for telecommunications applications, see Section 2.4), and earlier steps are informal [Eberlein, Halsall, 96a]. CABS acknowledges the need for a tool where the behaviour of a new service can be sketched at an early stage (although this is only one aspect of the requirements). The customer and service designer can, after providing some behavioural examples of the required behaviour, explore the new service by simulation. This is a form of high-level prototyping. CABS is also able to identify similar behaviour in previously specified services and suggest these as solutions, to be re-used in whole or in part.

1.3.1 From Service Idea to Formalised Requirements

Let’s assume that a service provider comes up with the idea that a new telecommunications service is needed to increase their income and to attract new customers. The cloud at the top in Figure 1.1 illustrates such a vague idea of some new functionality. The more focused idea might then be to provide phone users with an *emergency service*, i.e. if something happens, a specific emergency number is automatically dialled. The details have not yet been worked out, but the board meeting assigns a task to one of the telecommunications service sales employees which is to produce a proposal on the functionality, and to acquire an estimate of how much it would cost to order the functionality from a telecommunications company.

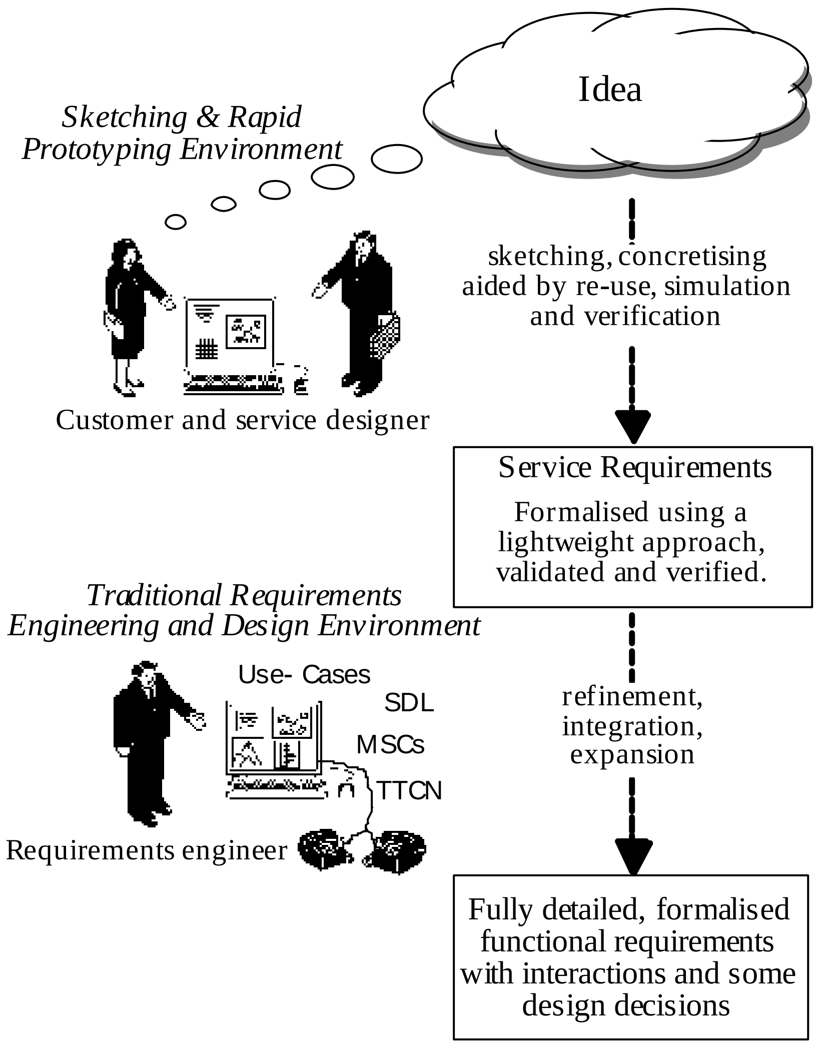

Idea
Sketching & Rapid Prototyping Environment
sketching, concretising aided by re-use, simulation and verification
Customer and service designer
Service Requirements
Formalised using a lightweight approach, validated and verified.
Traditional Requirements Engineering and Design Environment
Use- Cases
SDL
MSCs
TTCN
refinement, integration, expansion
Requirements engineer
Fully detailed, formalised functional requirements with interactions and some design decisions

Figure 1.1: From an idea via formalised requirements sketches to a full specification.

The sales employee makes a *mental picture* of how the new service would work from a phone user's point of view. Traditionally, a large text document containing requirements of the new telecommunications service, interwoven with descriptions of functionality, restrictions, limitations, implementation details etc. would be produced. Once the service is ordered and delivered half a year later, it is hoped that it meets the customers needs and the informal requirements. If not, the company may face legal proceedings on the meaning of the requirements specification documents.

If she was using CABS, the service designer would make a number of sketches of the behaviour of the new functionality (as seen from the telephone user's point of view) in the graphical editor illustrated in the top left picture in Figure 1.1. The service designer would first sketch some examples of the most common use of the service. The most frequent behaviour may be: if a telephone user has an *emergency service* set up and he lifts the phone but is not able to dial a number (for example a diabetic in distress, unable to dial a telephone number but able to lift the receiver), a previously selected number will be dialled after a short delay (to make sure it is not a normal call). The receiver of the call would need to have the existing telephone service *Callers Display* to see who is calling, and can then decide what action to take For example, he might send an ambulance/doctor/nurse or call the neighbours to check the situation). The service designer may also decide to provide examples of the expected behaviour if the called number is busy or if there is no answer.

Once these examples have been given as behavioural example sketches, the sales employee asks the system to propose a solution. A matching algorithm searches a case library where all previously formalised and implemented telephone services are stored, and identifies a number of services that exhibit similar behaviour. The user inspects them, reads some brief textual descriptions of them and may explore some of them in greater depth by simulating their behaviour with the simulator provided[8]. The system also points out where differences exist between the sketches of the behaviour and the formalised behaviour.

The service designer may decide on one proposal that is close in behaviour and already implemented by another company having a large number of residential care homes, where the

[8] Simulating their behaviour involves initialising a number of phones and setting up the different services for the different phones. The user lifts the receiver with a mouse click on the computer screen and tests out the behaviour as if real telephones were involved.

individual guests live in their own apartments but have a reception with a nurse and part-time medical doctor. The service has been in use for 6 months, and after 3 months of use, the customer ordered an extension of the service since the staff quickly found out that they needed three alternative choices of numbers (reception, nurse, doctor). When exploring the service further (using the simulator) she finds that the emergency numbers can only be changed by the receptionist. After considering the customers that her company intends to target, she decides to add the possibility for the telephone user to change the emergency number list themselves. She gives some examples of this behaviour and makes a selective match using only these input examples, and finds that the service *divert call* has a *set-up* functionality that fits the needs well and which only needs minor adaptation of the behaviour. The sales employee calls the technical service support at the telecommunications company they use and also transfers the input examples and selected solutions (middle square box in the Figure 1.1). A requirements engineer receives the formalised requirements, simulates and verifies them together with all other services the customer has to identify interaction and also uses traditional methods to look at how a design of the functionality can be made together with an estimate of the cost. One hour later, the customers sales person gets a proposal back which contains a service which includes the desired behaviour and where all the functional behaviour has been formalised (bottom square box in the Figure 1.1, all packaged into a simulation environment easy to use for the customers sales person). The sales person validates and verifies the service and, at the next board meeting, she demonstrates the functionality of the new service by simulating it on her PC with connection to a number of telephones. The decision is then made to go ahead and order the service which is delivered by re-using parts of the implementation from the similar services.

1.4 Structure of Thesis

Chapter 2 gives a brief background in requirements engineering, formal methods, case-based reasoning and graphical notations, with references to related and relevant literature/research. In Chapter 3, a brief overview and introduction to the problems directly addressed in this research are given. Chapter 4 shows the graphical input examples and defines the syntax and the detailed information that may be added. The case library and everything stored in it is explained in Chapter 5. Definitions of equal and similar behaviour and how these can be translated into a set of features used to identify cases in the case library that have similar behaviour is explained in Chapter 6. In Chapter 7, the design process from an informal idea of a new behaviour to validated and verified formal requirements is explored. Chapter 8 contains an evaluation where the ability to identify similar cases is explored, along with ways in which a solution can be partially evaluated against the input examples. Further work and ideas of improvements are given in Chapter 9. Chapter 10 gives a summary and the conclusions of the research. Appendix A defines the logical notation used by CABS as internal representation. Appendix B contains a glossary of a number of telecommunications terms. Appendix C contains all the formalised telephone services stored in the case library and used for evaluation. Appendix D contains all the input examples used for evaluation in Chapter 8. Appendices E, F and G are reviewed papers, published during the research.

Chapter:

2. Background

This chapter describes interesting areas related to this research project:

- Requirements engineering.
- Formal methods, their benefits and limitations.
- Examples of formal methods in telecommunications.
- Visual notations for state based systems, both telecommunications oriented and generic notations (SDL, MSC, PTNs, Petri nets, etc.).
- Case-Based Reasoning applied to specification and design tasks.

A brief background from the perspective of this research is given for these areas and some references are given to enable the reader to investigate them in greater detail.

2.1 Requirements Engineering

In system development, a major task is to establish in detail what the system is supposed to do. Requirements engineering is concerned with capturing, analysing and defining precisely the tasks the system should perform. This includes formalisation, re-use and evaluation of the system and its requirements. Identifying the requirements is an essential element of system development. Faults/misunderstandings at this level are often very difficult and costly to correct at later stages. Many faults in systems are traced back to requirements capture and specification stages, and are believed to cause a large proportion of industrial costs for poor software (estimated by the UK Department of Trade & Industry to be above £2000 million per year) [Schofield 92]. In addition to this, many systems tackle *wicked problems* [Sommerville 96] where the true nature of the problem first emerges when they are solved during development. Telephone services may be classified as *wicked problems*. Even if their coarse grain characteristic behaviour is simple, interaction and unusual situations can be difficult to identify and predict, and are often first identified when implemented. Prototyping may be useful in identifying and solving *wicked problems*, since these difficulties may be encountered in a prototype and can be solved before a full implementation is made. If prototype development by programming is impractical, too costly, or not feasible for other reasons, simulation of behavioural requirements may be considered (this approach is used in CABS). Simulation and prototyping provide new knowledge, as Herbert Simon elegantly expresses it: Firstly, "*even if we have the correct premises, it may be very difficult to discover what they imply*" and secondly, "*All correct reasoning is a grand system of tautologies, but only God can make direct use of that fact. The rest of us must painstakingly and fallibly tease out the consequences of our assumptions.*" [Simon 81, page 19].

A requirements specification should be open to different implementations as long as the implementation reflects fully the required behaviour, and excludes all unwanted behaviour. Implementation of telephone services has been achieved on a variety of systems (mechanical, electronic and digital), in different programming languages and programming paradigms (centralised, distributed, concurrent).

A lot of research effort is focused on re-use, and it is assumed that the full potential of re-use in system development is far from fully exploited. Re-use by categorisation is one of the main research activities in requirements engineering [Maiden, Mistry, Sutcliffe, 95] and categorisation is essential to the identification of relevant parts for re-use.

In program development, re-use is performed by identifying and using program components or objects from a software library. The amount of code re-used is dependent firstly on the classification and description of the parts so that they can be identified when needed, and secondly on how well re-use is incorporated into the system development process. Automated identification and re-use of software that has not been classified manually is difficult. Most program code is context dependent (the interpretation of a program statement is dependent on the previous and following statements) and allows a lot of freedom to construct a program in a personal style, making automated identification and re-use difficult (although there is ongoing research in this area). Behavioural requirements are sometimes less complex than code because not all the details are included in the requirements. If a formal method restricts the possible ways in which a behaviour solving a particular problem can be described, comparison between different requirements is facilitated, and automated identification of parts that may be relevant for re-use will benefit.

2.2 Formal Methods

Since the 1960s, formal methods have been of growing interest, and have been targeted with increasing research effort. Formal methods are often regarded as a scientific approach to software development [Hall 90]. Formal methods allow precise specification of some aspects of a system; informal specifications are often imprecise, incomplete and ambiguous. A wide variety of formal representations are available which are suited to different tasks in requirements specification and the system development process [Barroca, McDermid, 92]. However, formal notations are not suitable for everything in the requirements and design process, and it is important to carefully select those parts for which they are used [Bowen, Hinchey, 95]. One of the main principles applied when choosing formal representations for requirements engineering is that “a formal representation should be as simple as possible, but no simpler.” [Zave, Jackson, 97, page 106]. Technological advances and increased expressiveness in formal representations are important in order to tackle new and demanding application domains. However, a formal representation with the ability to capture everything would be complicated. Thus, expressiveness has a price in terms of automated reasoning capabilities, executability, proof of consistency, level of mathematical skill needed to understand and use a formalism, etc. Carefully choosing a simple but sufficiently expressive formal notation [Wing 90] is an important task when using formal notations, and limiting expressiveness is a major approach to taming the combinatorial explosion in production systems [Acharya 94]. Sometimes in formal methods, more research effort has been directed towards expressive formalisms that are generic and capture as many aspects and details (such as timing constraints, indeterminism, probabilities, concurrency, etc.) of the system as possible [Johnson, Benner, Harris, Sanders, 93], than into embedding the formalisms in some system development method which facilitates requirements capture and aids the transfer of requirements into a formal notation.

Since the 1980s, formal methods have been used in industry for safety critical applications (avionics, railway signalling systems, power plant control systems, medical electronics, VLSI design), and are often applied by highly skilled mathematicians/logicians using semi-automated theorem provers. Outside these areas, the use of formal methods is less common. Even so, a number of successful individual projects have been reported [Cleland, MacKenzie, 1995]. There is an increasing demand for the use of formal methods in safety-critical systems, for example the UK Ministry of Defence (MoD) strongly recommends formal notations, analysis of consistency and completeness in specifications of safety-critical components and software [Bowen, Hinchey, 95]. The interest in and demand for formal methods for security-sensitive applications such as telecommunications, traffic signalling systems, share dealing systems, banking and finances, is increasing. It is believed that making the use of formal methods easier for non-mathematicians would enable a wider use of formal methods in security-critical/sensitive applications. One factor holding back a wider use of formal methods is "maths scare" amongst designers and programmers [Hall 90]. Furthermore, greater care in identifying which formal methods are suitable for which problem is needed, as the use of an unsuitable formal notation may cause a project to experience difficulties or even fail.

The main issue of this research is to show that it is possible to identify similar behaviour to enable requirements capture and re-use in a case-based reasoning system. Some related issues have been briefly explored and addressed to enable exploration and validation of the main focus of this research, which is the identification and re-use of similar behaviour:

Help users to give more accurate requirements.

Addressed in CABS: Sketching input examples exemplifying the behaviour of some required functionality that are used to identify similar behaviours enables the user to re-use previously formalised and implemented specifications. They can be simulated and verified using a case-based reasoning approach which is hoped to aid the user in identifying problems at an early stage compared with traditional approaches where the first formalised level is program code. Problems with service specifications were identified during evaluation that had not been identified before matching, formalisation, validation and verification of the behaviour which at least shows that these tools under some circumstances are of benefit.

Reduce errors in the final requirements and system implementation.

Addressed in CABS: By re-using a proposed solution from the case library, errors will be reduced since the re-used service has already been integrated with other services and implemented.

Identify and re-use previously specified behaviours that have already been implemented.

Addressed in CABS: The case-based matching is able to identify similar cases in the case library that can be re-used in whole or in part as shown in Chapter 8. Identification and matching is the main focus of this research.

Simplify the task (for non logicians) of creating and modifying formal requirements specifications.

Addressed in CABS: Graphical input sketches combined with transition rules are believed to be more readily accepted than the direct use of a formal logic. Also, an iterative refinement process is proposed and supported by CABS. To confirm this hypothesis, an evaluation with potential users is needed, but this is outside the scope of this research.

Issues relevant to the task of bringing formal methods to industrial use are explored more in depth in the following section (Section 2.2.1). If the readers main interest is the identification and matching similar behaviour reading this section can be omitted.

2.2.1 Issues of Formal Methods and their Relation to this Research

The following are some claims, opinions and critiques about the use of formal methods which are relevant to the application domain of CABS. Not all of the seven issues are within the scope of this research but some of them have been addressed to enable evaluation of CABS's main issues and others are briefly discussed with some ideas or references to potential solutions. *Selected solution:* is a brief description of CABS's specific way of addressing them (independent of whether they are a main issue for this research):

It is commonly believed that formal methods are difficult to scale up since expressive formalisms are often not executable and are only seen as a way of describing requirements more precisely than with natural language [*Hall 90*].

Proposed approach: Choosing a simple logic which is sufficient to formalise the initial requirements, but not necessarily able to capture the full and final behaviour, allows us to specify some basic behavioural requirements for the application domain of telecommunications services and to handle these effectively by simulation of the initial behaviour, re-use, verification and validation.

Selected solution: A simple logic tailored to this particular application domain has been shown to enable re-use by case-based reasoning, simulation and limited verification. Also, translation to and from restricted natural language has been applied for similar notations [Dalianis 95].

Resistance from non-mathematicians and non-logicians to the use of formal methods [Zave, Jackson, 96].

Proposed approach: Bearing in mind the rejection of formal methods by designers and programmers at Ericsson it is hoped that by using graphical notation similar to informal or semi-formal notations already used in the application domain, the acceptance of formal methods will be eased. Textual rules are used in the domain of telecommunications, transition rules bear similarities to these textual rules and transition rules can be translated to and from restricted natural language [Dalianis 95].

Selected solution: A graphical notation is chosen but no effort has been taken to make the notation similar to existing notation since this is beyond the scope of this research and such a notion should be developed in close co-operation with the final users to warrant for an acceptance. The user is not directly confronted with the logical notation used internally. A textual representation of transition rules has not been implemented.

Formal specifications are difficult to re-use [Hall 90].

Proposed approach: By using a case-based reasoning approach and a restricted logical notation, it should be possible to identify parts from a case library that may be re-used. Identification of cases that are similar to the behaviour exemplified in the input examples will enable re-use if the same or a similar case exists in the case library. Also, re-use of individual transition rules may be possible, if the transition rules are context independent.

Selected solution: Matching input cases against a case library enables the identification of similar behaviour (CABS uses an uncomplicated matching algorithm described in Chapter 6) and evaluated in Chapter 8. Results are encouraging and the matching is able to identify the most similar case to sets of input examples. If no matching case exists in the case library, the matching is able to identify similar transition rules that may be re-used. The features used for identifying similar behaviour may need fine-tuning but they have proved to be fairly robust with the case library used for the evaluation

Formal methods are often said to be unsupported by tools which allow the user to iteratively refine and clarify the requirements [Bowen, Hinchey, 96].

Proposed approach: Design and use an approach based on an iterative refinement process where an initial idea of some new behaviour can be refined and modified iteratively until it captures the intended behaviour.

Selected solution: The CABS approach includes a refinement methodology supported by the implementation (see Figure 7.1, page 124). The process was used in the evaluation and no obstacles were encountered. Even if no matching case is available, the input examples can be used to generate a set of transition rules used as an initial proposal for the new service (see Figure 7.1). During the evaluation (Chapter 8), a few unexpected problems were identified both in the input examples and in the case library, which shows the value of using test cases generated from input examples.

Formal specifications are often regarded as difficult to modify [Gotel, Finkelstein, 94].

Proposed approach: 1) Structuring the telecommunications services as cases (sets of transition rules), 2) keeping links to the original input examples, test cases, full specification, etc. (enabling traceability of requirements, from where they originate and where they have been used) and 3) providing a simulator and automated verification so that modifications can be explored in depth.

Selected solution: CABS's approach is to: 1) structure cases as sets of transition rules, 2) store all original input examples, informal comments and test, 3) simulate and verify cases separately or together with other services. When the behaviour of a service needs modification, the input examples aid the understanding and modification process. Test cases identify precisely where the behaviour has been changed.

Formal methods are accused of being difficult to combine and integrate with current system development methods [Bowen, Hinchey, 96].

Proposed approach: By using a formal notation that can be translated into graphs, state machines and natural language, and used for simulation (in the same way as prototypes) and to generate test cases, CABS exhibits desirable features that may integrate into many systems development methods.

Selected solution: CABS focuses on re-use and requirements capturing - a process that is currently hardly supported at all. Nothing in CABS contradicts traditional system development methods and a system which aids system development would benefit from the functionality exemplified by CABS. It may even be possible to translate the output from CABS into the representations used in telecommunications (SDL, Use-Cases, MSCs, etc.) but this has to be investigated. Since the formal notation captures state machines, translation to state based formalisms is possible.

Executable formal methods are often regarded as computationally inefficient.

Proposed approach: This is often true for advanced formalisms handling indeterminism and where the application domain is complex. A restricted logic is proposed for CABS which doesn't aim to capture all the behaviour of the system (only the initial behavioural requirements, leaving out unusual behaviour, error cases, etc.), gives sufficiently fast response times for both simulation and theorem-proving.

Selected solution: The CABS system is implemented in PROLOG with acceptable response time on a desktop computer (response times are below a second for simulation and stepwise verification). Matching times are acceptable even if the case library is considerably larger (see Chapter 8 for details).

Requirements capture is often seen as the main bottleneck in system development [Bubenko 95]. Using a rigorous formal notation in a lightweight formal approach to capture the initial behavioural requirements is shown to have some powerful and desirable features, such as enabling the identification and re-use of previously specified behaviour.

2.3 Telecommunications and Formal Requirements

Telecommunications have, until recently, been mainly technology driven (limits have been set by technical constraints), and less application driven. This has changed rapidly due to the computerisation of telecommunications, which has started replacing technical limits by limits of imagination and innovation. This revolution will change the demands and judgements of telecommunications services. Increasing demands for innovative and creative services with high levels of usefulness, user-friendliness and functionality are emerging, as they are no longer so tightly limited by the difficulties of implementation in hardware and software. Bandwidth is still a limited resource, but the bandwidth available now (and in the near future) is far from fully utilised. One scenario of the future is that bandwidth will be supplied in the same way as petrol/gas/electricity (Norway and Sweden allow customers to change their electricity supplier), and the user will make short term agreements with the supplier offering the best deal on bandwidth. Under this kind of price competition, telecommunications vendors or independent service providers will have to provide services adding value to bandwidth supply, such as more sophisticated telephone services (traditionally call waiting, multi-party calls, re-call, call diversion, levels of availability/privacy, charge advice, banking and also, increasingly, services based around the integration of mobile phones/home phones/computers/video/music, etc.). Changing supplier means, in most circumstances, a changed set of services. Services will be the supplier's best assets in such a scenario, and patenting services may be more relevant than patenting hardware. This puts telecommunications services at the forefront of the basic functionality (a basic telephone call) and providers who cannot provide competitive services to their customers in a short time will see their market share decrease rapidly. Those suppliers who are able to offer services in which the users are interested, will attract more customers. Parts used to design and implement services have been standardised and formalised such as service independent building blocks (SIB's, [ITU Q1203], for formalisation see [Nyström, Jonsson 96]), but telecommunications services themselves cannot be standardised without stifling competition between operators for customers.

Telecommunications services can be classified as security-critical (hence formal methods are of interest and relevance). It is not acceptable that an additional telephone service should inflict problems on basic functionality such as an emergency call, or cause problems for other telephone users, (situations which have in fact occurred in the past[9]). Formal specifications have been explored as ways of identifying and reducing such problems in the system development process but are not routinely used. Pamela Zave at AT&T Bell Laboratories has been active in this area since the late 1970s. PAISLey is an executable specification language developed by Zave and her research team at Bell Laboratories over 8 years (from 1979 onwards) [Zave 91]. Her research is now aimed more at muliparadigmal approaches to requirements specifications, where the underlying notation is based on a simple logic [Zave, Jackson, 97]. There are some similarities to CABS's formal notation; for example, neither system allows internal events, in order to keep the formalism and semantics simple and only allow specification of the system's externally observable behaviour[10]). Using logic as the

[9] *Call diversion* was one of the earliest telephone services provided. The specification and implementation allowed redirection over many steps. Unfortunately, it also allowed redirection to the original number. When a user diverted calls to their holiday home and then diverted calls back from there to their main home, the signalling bandwidth between the two telecommunications switches was, after a while, used up by phone calls diverted back and forth between them in an infinite loop. Worse still, a restart of the telecommunications switch left the diverted number unchanged, causing the same problem all over again. This might have been prevented with formalised requirements, which had been validated and verified (in CABS, such loops cannot be specified and the number of steps that a telephone call can be diverted has to be specified explicitly).

[10] By only specifying the system's interaction with its environment and not the system's inner workings, the specification is kept *implementation independent* (a black box approach since nothing of the inner working of the system is exposed). The inner working of the system is left for design and implementation where hardware and software architecture can be chosen to meet other non-

underlying formalism shifts the focus from the development of a language suitable for a particular application domain to the selection of a suitable subset of logic, which is as restricted as possible, but expressive enough to capture the desired features of the domain.

A different approach to service specification (compared with the PAISLey approach) is the WATSON system [Kelly, Nonnenmann, 92] also developed at AT&T. WATSON takes informal textual examples of telephone services (a graphical notation is also mentioned, but not illustrated), and translates them semi-automatically to a logical notation (similar to the one used in CABS). After the natural language scenarios have been given (WATSON was able to handle scenarios of the size of four sentences (50 words), in 1992), the system tries to identify incomplete parts and problems in the informal description and asks the user yes/no questions (WATSON uses an "off the shelf" theorem prover and domain knowledge mainly encoded in Lisp). WATSON produces control flow skeletons together with attached code for some parts. Control skeletons can then be simulated. Such an approach requires large amounts of knowledge (encoded, stored and kept updated in WATSON) of requirements specification, design, implementation and application domain knowledge, to be able to produce control flow skeletons with attached code from short textual descriptions (such as hardware, network protocols, expected end user etiquette, style of skeleton design, etc.). Capturing a large application domain knowledge base and keeping it up to date is recognised as a problem in the WATSON project. This is a large task even for a narrow application domain (which can be partly bypassed if case-based reasoning can be applied, as discussed in Section 2.5).

A Requirements Assistant for Telecommunications Services tool (RATS) was developed during a PhD project at the University of Wales [Eberlein 97]. RATS enables the user to give information in a structured and layered approach, mostly as informal text but also with links to libraries and in other notations. A high level of tractability is maintained by keeping references

functional requirements (price, size, security, power consumption, distribution, modularity, technology, etc.)

and links between all information objects. The system uses applic ation domain rules to keep track of what information is still missing, guiding the user and ensuring that all the necessary information is given (218 user defined rules and 33 constraints are currently used). RATS can ask questions such as *"How do you intend to achieve the goal 'authentication very important'?"*. Once the user has linked all information with a traditionally produced SDL diagram (production of diagrams is aided by the structured requirements), RATS' task is completed. Compared with using large textual requirements documents (which is the current practice), the structured approach in RATS has some obvious advantages such as tractability and maintainability (for a comprehensive analysis of the tractability problem see [Gotel, Finkelstein, 94]).

A formal specification project at ERICSSON Telecommunications (research phase 1985-1991, implementation phase 1992-1997) was centred more around temporal logic [Echarti, Stålmarck, 88] and theorem proving than PAISLey and WATSON (the logic used is similar to the one used in WATSON). The functional behaviour of telecommunications services is expressed in a logical notation (a graphical notation based on a tree structure is also added in parallel with some logical expressions); generic application domain knowledge (a conceptual model) is given in a graphical notation (directly translated to logical axioms). Simulation enables validation of services, and theorem proving is used to prove consistency (inconsistencies between application domain knowledge and services can be identified). Test suites used in telecommunications for testing implementations can be produced semi-automatically from event traces generated by the theorem prover (all possible behaviours up to a certain length may be generated from the specification) [Ridley 94] [Ahtianen, Chatras, Hornbeck, Kesti, 94]. Event traces share similarities with *Node Usage Cases,* used in telecommunications to guide design and implementation [Ask 94]. The notation used in CABS is based on the notation used in the research project at Ericsson (the logic has been simplified and restricted; see Appendix A).

There are three desirable features for service development:

A prototype/simulation of the new behaviour is needed to explore new services.

Formalisation of the functional requirements, to ensure stable properties and safe integration with other functionality.

Ability to re-use, in order to optimise implementation of new services by re-using previously specified and implemented services.

If formalised requirements can be used as a prototype, the new functionality can be explored on its own as well as with other services and both 1) and 2) are covered. If the formalised requirements can be created by identifying and re-using similar services, then 3) will be solved. Current research explores this approach using a narrower focus than WATSON (CABS does not aim at code production) to capture, refine, re-use and produce requirements in the domain of reactive systems[11], and to enable simulation of the new requirements. CABS shares one main ambition with WATSON, in Kelly and Nonnenmann's own words: "*helping ordinary people (that is conventionally trained telephone engineers) achieve extraordinary results (mathematically precise specifications)*". If the mathematically precise notation can be hidden or encapsulated, it may be possible to relax the limitation to *conventionally trained telephone engineers* with the ambition that telephone users, sales personnel, etc. should be able to specify their requirements themselves, if their aim is to capture only the characteristic requirements (not necessarily consistent and complete, i.e. including all exceptions, odd cases, resolved interactions). Extending, refining and integrating the new behaviour with other telecommunications services would need more experienced requirements designers. The CABS approach takes coarse grained graphical input examples exemplifying the desired behaviour, identifies similar services and parts of services that may

[11] Reactive systems have a direct relation between stimulus and response (input/output) and need external stimuli to produce a response. An example of a trivial reactive system is a light switch having two states (on/off), with the stimulus being: switching it on or off.

be re-used, and enables validation (simulation of the behaviour) and limited verification of requirements. This is a worthy task in itself, and if this can be accomplished and accepted by industry for the specification of reactive systems, the benefits may, for some application domains, be sufficient to make it worthwhile incorporating formal requirements into the system design process. Validation by simulation and verification may be regarded as prototyping combined with the capability to analyse the behaviour in depth.

2.3.1 Specifications in Telecommunications

Customers (public and private telecommunications suppliers, service vendors, institutions, universities or even private customers), order specific telephone services which they hope will meet their needs. One difficulty is that precise informal requirements are difficult to produce and require a high level of skill. It is easy to find examples where misinterpreted informal requirements have caused serious problems[12]. Formal specification aims to provide precise and exact descriptions, independent of stakeholders (customers, engineers, programmers, sales personnel, translators, managers, etc.). Different abstraction levels (with more, or less detail shown) and views (wether only issues relevant for a particular perspective are shown) of the requirements may be useful for different stakeholders [Pohl 94].

---

[12] One story (not officially confirmed) goes that the service *three party call* was informally specified in such a way that it was able to reach a situation where four parties were able to speak with each other. When the three party call service was delivered, the customer insisted on having the four party situation. This could only be implemented by redesigning the hardware, because the exchange only had digital mixers capable of mixing three speech connections. Finally, a solution was found: a trunk line (a connection to another telephone exchange) looping back to the same station, treating the incoming (two party) call as one external caller and able to connect the incoming call with the two other parties. This is an expensive solution, but must, in this case, have been estimated to be less costly than breach of the contract.

Naming something often gives us a false sense of understanding it. It is often surprising how differently words are defined by different domain experts, definitions which sometimes even contradict each other. In telecommunications, the expression “User A is in speech connection with user B” has been defined in the following ways by different persons:

A can hear any sound generated by B.

A can hear B and B can hear A simultaneously.

Either A hears B or B hears A.

None of the three definitions is incorrect. However, speaking about *“being in speech connection”* or *“being connected”* without agreeing on a definition will cause problems during specification or, worse, during design, implementation or product verification.

2.4 Graphical Notations

There are two main types of symbolic representations which both use symbolic expressions: sentential representation (natural language descriptions) and diagrammatic/graphical representations. The latter can explicitly capture topological and geometrical relationships which can only be captured indirectly in a textual representation [Larkin, Simon, 1987]. There is a growing interest in, and promising results from, the use of graphical formalisms for knowledge elicitation, specification and programming (see for example [Hirakawa, Monden, Yoshimoto, Tanaka, Ichikawa, 86], and [Addis, Gooding, Townsend, 93]). It is obvious that the trend in interaction/communication involving computers is becoming more graphical oriented (icons, windows, pictures, animation). For many tasks, graphical notations are claimed to be more readable than textual language [Mataga, Zave, 94]. For the creative and exploratory phases of forming new knowledge, visualisation is often essential and the use of diagrams also aids knowledge elicitation and co-operation between those involved [Addis, Gooding, Townsend, 93]. In formal methods, advanced specification languages have been developed which tackle a wide variety of application domains, but the human aspects of the use of these notations (making them easy to use and understand) have been slower [Robertson 96]. When new formal notations are created, diagrams are often used (see for example [Allen 83], [Kowalski, Sergot, 86]), but the final notations are mostly pure linguistic representations. The role of diagrams is rarely recognised and is, therefore, underestimated in the communication and conceptualisation process [Addis 94].

Recently, more research effort has been focused on giving informal or semi-formal graphical notations clear syntax and semantics, and developing new notations to enable the graphical expression of conceptual models, requirements, dynamic behaviour and programs. Earlier approaches using conventional state machines or state-diagrams encountered difficulties when applied to system design, due to the exponential explosion in the number of states [Harel 87], and were claimed to be hard to read, modify and refine and not suitable for complex specifications [Martin, McClure, 85]. Different approaches to overcome these problems have been explored and graphical languages (often combined with a textual language) are common in system development today; for example:

SDL (<u>S</u>pecification and <u>D</u>escription <u>L</u>anguage, standardised by the International Telecommunications Union, [ITU-Z100]). The SDL language contains both a graphical and textual part. The graphical part is similar to flow charts. The graphical parts together with the textual part of the language enable the user to describe the functionality in such great detail that executable code can be generated directly. Some formalisation efforts have been undertaken, see for example [Leue 95]. With minor alterations in the semantics, a subset of SDL can be translated to Petri nets which has been used for protocol verification at Siemens Telecommunication, Germany [Regensburger, Barnard 98].

Statecharts [Harel et. al. 90]. A graphical notation designed to make it easier to design and implement real time systems. Similar to SDL, it has a graphical part and a textual part and detailed descriptions can be created and used to generate executable code.

Process Transition Networks (PTNs) [Malec 92], [Sandewall 90]. PTNs can be translated to temporal logic and to a subset of Petri nets. The notation aids conceptualisation and knowledge acquisition and its simplicity makes it easy to use for domains in which the expressiveness is sufficient.

Use-Cases [Jacobson, Christerson, Jonsson, Övergaard, 93]. Not a notation in itself, but which allows different notations or even text documents describing specific examples of how the system to be designed will behave. Formalisation and graphical syntax is under development [Regnell, Kimbler, Wesslén, 95].

MSC (Message Sequence Charts describing signalling between objects in a distributed system). A widely used graphical trace language for communicating entities. MSCs may also be used for requirements specifications with a set of suitable tools [Ben-Abdallah, Leue 96].

Petri Net notations [Jensen 97] are a graphical notation enabling behavioural analysis and model checking. The notations are often regarded as complicated for non logicians and this is sometimes overcome by translating to Petri nets from specialist languages. For example some parts of SDL (with slightly altered semantics) can be translated to Petri nets in order to enable model checking [Grahlmann 98]. Since Petri nets are emerging as a common formal notation into which other notations more close to notations used in different application domains can be translated, Petri nets are described in more depth in Section 2.4.1.

These languages are all more expressive than is required for the approach taken in CABS, and include different types of concurrency which is often useful or essential when designing a complex system. In most larger systems, such as telecommunications, the full functionality is difficult to describe with a state transition notation as the number of states will by far exceed the number of states that can be practically handled in available notations. Even so, examples, scenarios and sketches of behaviour for different aspects of a system's functionality can be expressed with state-flavoured style, which is often done informally to complement textual descriptions. An important aspect of CABS is that the graphical notation used is not intended to be a traditional state-based notation capturing a finite state machine: a diagram in the notation used may represent a large set of state machines enabling the user to sketch a behaviour, ignoring details and avoiding confrontation with the so called *state explosion*. The notation used in CABS captures the initial (design independent) sketches of behavioural requirements before design decisions have been taken[13] (the graphical notation for CABS is described in Chapter 4). Little consideration and time has been spent on what graphical formalism is most appropriate for the application domain, bearing in mind that the main research contribution is the identification of similar behaviour. Graphical representation may provide greater benefit if it has been adapted to the application domain and to a specific set of users [Robertson 96], but to do so is beyond the scope of this research.

2.4.1 Petri nets

Petri nets are used as a powerful algebraic graphical notation for communicating automata and are expressive enough to capture systems where concurrent events occur. This is beyond the ability of the chosen notation for CABS but both Petri nets and input examples are state (in CABS a node denotes all states the which the given restriction hold) and transition oriented. Petri nets developed by C. A. Petri in the sixties were the first general theory for discrete parallel systems. Petri nets have proven to be well suited to describe concurrency. A wide variety of Petri Net notations exist which either extend the expressiveness to new classes of problems or make them easier to use. Examples of extensions are high-level Petri nets, timed Petri nets, stochastic Petri nets and Coloured Petri (CPN) nets [Jensen 97]. Petri nets have always had a precise formal definition which

[13] Design decisions are, for example, dividing the system into communicating entities, internal concurrency, communication mechanisms, etc. An example of how deeply design decisions are included in these formalisms would be to use, for example, MSC diagrams with signalling switches to specify a telecommunications service, and implement the functionality using the internet, instead of a network of signalling telecommunications switches (most of the "specification" would be irrelevant").

enables the use of powerful analysis tools (e.g. SPIN [Holzmann, Peled 94]) that can be used to prove different properties of Petri nets. Also, there is n on-going effort to standardise Petri nets.

Lately, Petri nets have emerged as a common notation for different graphical notations adapted to specific application domains. These notations are translated into Petri Boxes, a special kind of low level Petri nets enabling a wide variety of verification techniques such as model checking, verification and application of reduction algorithms [Grahlmann 98]. Both SDL and MSCs have been translated into Petri nets in order to use verification tools developed for Petri nets.

Petri nets look similar to input examples in CABS as shown in Figure 2.1 below (a low-level Place/Transition Net) where the right example is a Petri net and the left example is an input example for CABS as described in Chapter 4. The Petri net has been designed to visually look as similar as possible to the input example for CABS, it has not been explored whether the two examples are semantically equal. Even though the examples look similar, the terminology and way of thinking is different. Petri nets are built with places, input transitions, output transitions, input arcs, output arcs and tokens [Jensen 92]. Places can hold one or more tokens (in the example, there are two telephone tokens), arcs have the capacity to hold 1 or more tokens (the default being one), transitions have no capacity (cannot hold a token). A transition is enabled if the places with arcs leading to the transition have a number of tokens greater than or equal to the capacity of the arc (default capacity being one). During execution of a Petri net, the tokens will move around in the net and the number of tokens may vary. When using a Petri net, terms such as synchronisation, concurrency and merging are difficult to avoid. The Petri net example in Figure 2.1 contains the primitive constructions: synchronisation (e.g. the processes “ring tone a” and “ring signal b” are synchronised by starting the transition “dialling idle b”), concurrency (e.g. “ring tone a” and “ring signal b” are two concurrent processes started by the transition “dialling idle b”) and merging which are not used in CABS when sketching the behaviour of telephone services. In high level Petri nets, a token can contain complex data and may describe the entire state of the process or data base. For the input example in the notation for CABS, each node has facts that are expected to be valid, and all states in which these facts are true are denoted by the node. For more details see Chapter 4, and for details on facts for the nodes in the CABS input example see Appendix C.3. The additional facts for nodes in CABS notation may indicate that high level Petri nets are the closest of these dialects to CABS (tokens in low-level Petri nets cannot carry any data). On the other hand, high level Petri nets have a larger vocabulary such as functions (ML is used in CPN), markings, initialisation expressions, guards and are able to express process invocation, different types of loops and procedure calls. Kurt Jensen states: “Making a CPN model is very similar to the construction of a program” [Jensen 92]. This may be very useful when specifying and designing a complex concurrent system but is much more than CABS needs for initial sketches of required behaviour.

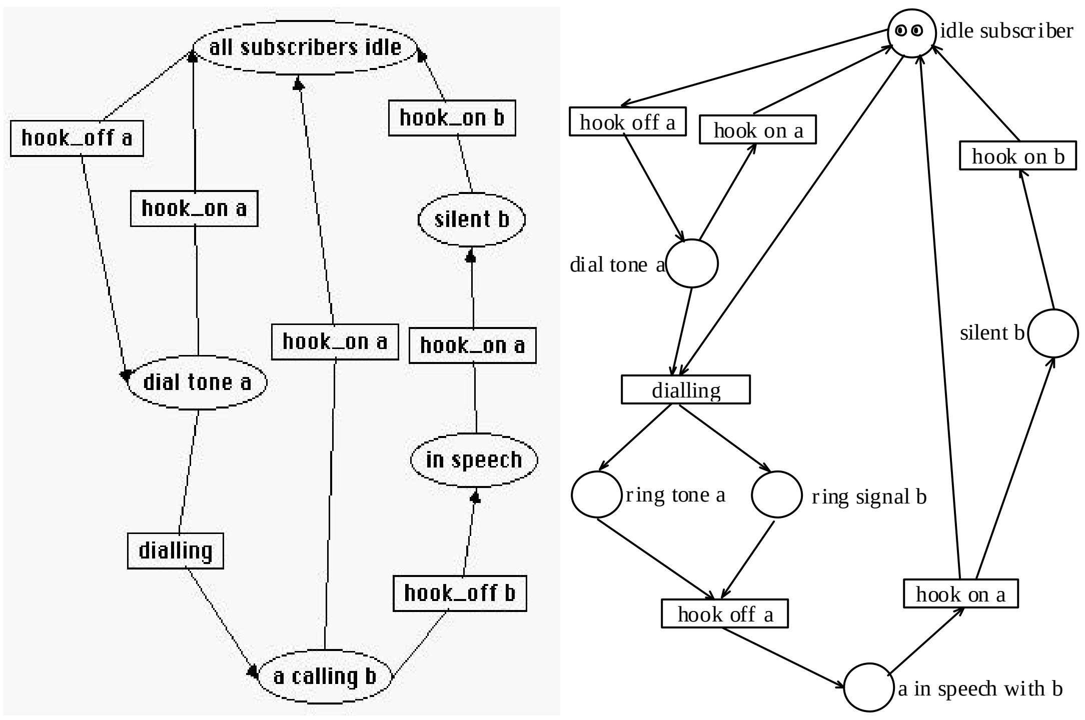


Figure 2.1: Input example in CABS and Petri net example

2.5 Case-Based Reasoning

The central concept of case-based reasoning is expressed by Riesbeck and Schank as: "... the essence of how human reasoning works. People reason from experience. They use their own experience if they have a relevant one, or they make use of the experience of others ..." [Riesbeck, Schank, 1989, page 7]. Aamodt and Plaza's picture, Figure 2.2, illustrates the main ideas of case-based reasoning: a problem is given in the top left corner, similar cases are retrieved from a case library and the most suitable case is selected and re-used. The most suitable case may need to be revised to solve the problem. If the solution is approved, the problem and its solution are stored in the case library. Next time a similar problem is encountered, less adaptation of the retrieved case may be needed and the performance will increase if similar problems are often encountered and the features identifying similar cases are good enough.

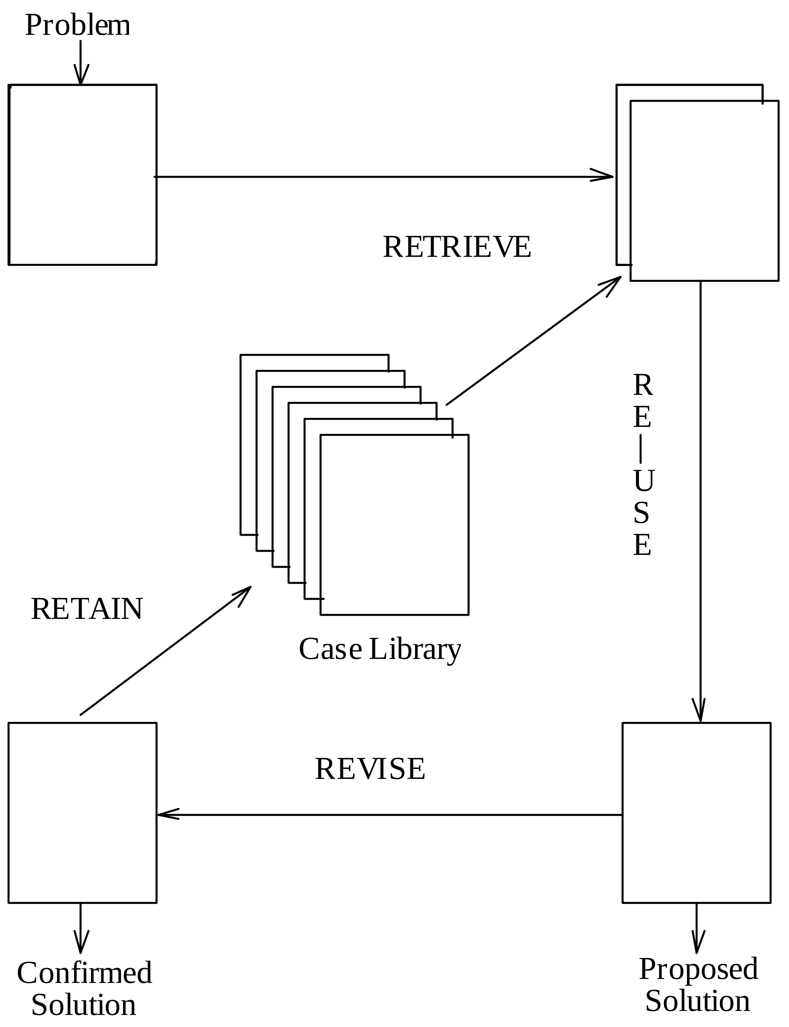


Figure 2.2: General architecture of a case-based reasoning system. Adapted from [Aamodt, Plaza 94].

If a rule based system produces a particular solution, or fails to do so, it may not always make sense to look at individual rules that produced the result [Jackson 90]. Looking at a previous case that has solved a similar problem may, for some situations, be easier to understand because cases provide a context for understanding [Kolodner 93]. A case-based system may also adapt to changing demands, for example, if a new type of problem not previously encountered is solved (if no similar cases are available, a solution to the problem is most likely to be produced manually). The solved problem and its solution are stored in the case library as a new case, with the aim of expanding its competence [Aamodt 93]. The next time the system encounters the same or a similar problem, the system will have increased its potential to produce a solution. It is more likely that, in a rule based system, the rules would need to be updated to include this new class of problems.

Case-based reasoning may be suitable for problem areas in which the knowledge of how a solution is created is poorly understood [Watson 97], e.g. the creation of formal requirements of telecommunications services from a set of behavioural examples. The WATSON system, described in Section 2.3, is one of the few research projects taking on the task of formally capturing knowledge about how telecommunications services are formalised from natural language in a semi-automatic approach. In technical domains, case-based reasoning has been applied to a variety of application domains such as: architectural design support [Pearce, Goel, Kolodner, Zimring, Sentosa, Billington, 92]; qualitative reasoning in engineering design [Sycara, Navinchandra, 89], [Nakatani, Tsukiyama, Fukuda, 92], software specification re-use [Maiden, Sutcliffe, 90], software re-use [Fouqué, Matwine, 93], re-use of mechanical designs [Mostow, Barley, Weinrich, 89], [Bardasz, Zeid, 92], telecommunications network management [Brandau, Lemmon, Lafond, 91], fault correction in help desk applications [Watson 97], building regulations [Yang, Robertson, Lee], fault diagnosis and repair of software [Hunt 97].

In conclusion, case-based reasoning may be applied to application domains that are not sufficiently well understood to create a consistent and complete rule-base, on condition that:

problems and their solutions have similarities.

a case library with past problems and their solutions is available or can be created.

there are good ways for identifying relevant cases in the case library.

solutions can be adapted and re-used for similar problems.

Chapter:

3. Introduction to CABS

In this chapter, an overview is given of the case-based specification approach, and an introduction to the problems addressed in this work. In application domains like telecommunications, formal methods are still not used for requirements specification. Even so, a number of logical formalisms seem to be ready for large scale commercial use in real applications and have been explored in the domain of telecommunications services (see for example [Armstrong, Elshiewy, Virding, 86] and [Echarti, Stålmarck, 88]). As explored in the previous chapter, there are a number of different reasons why formal methods are still rarely used for requirements specification in industry.

In the CABS methodology, the task of producing a requirements specification is not just handled as a simple task of transferring the requirements from the user to the chosen formalism. It is a much more involved intellectual process, and when parts of the requirements are captured, the user often modifies and changes his requirements, i.e. requirements change and evolve until the user is satisfied. This iterative refinement process is often acknowledged in software production and experimental development, but less often supported by formal methods. Formal methods practitioners sometimes give the impression that they are expecting the clients to have their requirements all ready, and the main task is to get them into some formal notation (not necessarily executable).

Using CABS, we view the process of producing formal requirements, in particular, behavioural requirements, as more of an experimental development task, where we start with sketches of required behaviour and use these sketches to rapidly produce something which can be evaluated in a variety of ways (simulation, automatic verification, simulation involving end users, etc.). We then refine the sketch, compare them with similar requirements, re-use parts of similar requirements, modify the original sketches, all this in a tightly integrated environment where no unnecessary demands on order or sequence are put on the requirements engineer. This will aid the user of CABS to refine and extend the requirements until she is convinced that the formalised requirements capture what the user/customer requires.

3.1 Outline of the CABS System

CABS attempts to ease or overcome some of the obstacles encountered when producing formal requirements specifications for telecommunications services. The approach is based on the combination of formal methods, case-based reasoning, example based input and the use of an executable logic. By using this combination, CABS aims to make formal requirements specifications more acceptable and to bring formal requirements specifications to practical use for telecommunications services (and similar application domains).

The CABS system is illustrated in Figure 3.1. In the top left-hand corner, the requirements process starts with a number of graphical input examples provided by the user and produced with the graphical editor implemented in CABS (see Figure 4.1 for an example input and the editor). These graphical input examples use nodes and links (explained in Chapter 4) to sketch the behavioural requirements. When the behaviour of some examples has been drawn, they can be refined and extended by selecting a node or link to obtain a window where details can be added.

The matching algorithm (the second box from the top on the left in Figure 3.1), uses the input examples to identify cases from the case library (top right in Figure 3.1) which capture similar behaviour. The cases are previously formalised requirements that have been validated, verified and integrated with other cases (as described in Chapter 5). An analysis of the differences and similarities between links and transition rules is used to identify transition rules that are similar (the analysis measures a number of features and is described in Chapter 6). It is always possible to determine whether the rules capture exactly the same behaviour (but this is less likely to occur). When a set of similar transition rules have been identified, each case is ranked on the basis of its transition rules and how well they match links in the input examples.

The user has a number of different options (shown in the third box from the top on the left in Figure 3.1) to choose from when confronted with the result from the matching. The user may select one of the proposed previously specified services (solid line from the re-use box) that have been identified as capturing similar behaviour to the exemplified behaviour. If a close enough case is not present in the case library, then a new service has to be constructed based on input examples, matching cases and transition rules. Alternatively, the input examples can be refined (this choice is shown with the broken line from the re-use box) in order to improve the match. If there is no suitable match in the case library, the input examples can be used as a starting point to specify a new case (explored in Chapter 7).

When there is a proposed case that the user believes may be an acceptable solution, she can verify and validate the proposed solution (the *Revise* box in Figure 3.1). From the input examples, test cases are generated which, if successful, verify that the proposed solution captures the behaviour exemplified in the input examples. The user can also simulate the dynamic behaviour of the proposed solution in order to validate that her intentions are captured (these simulations may also be added as test cases). A theorem prover analyses the solution with respect to known domain restrictions (this is not fully implemented in CABS: simple checks of restrictions have been implemented, but not fully integrated, in the CABS prototype). The user may also decide to undertake some adaptation of the proposed solution in order to make the behaviour conform to the input examples. At any stage, the user may decide to add more (or refine) input examples and re-do the match in part or in full (the broken line from the Revise box in Figure 3.1). When the solution has been validated and verified, it is added to the case library.

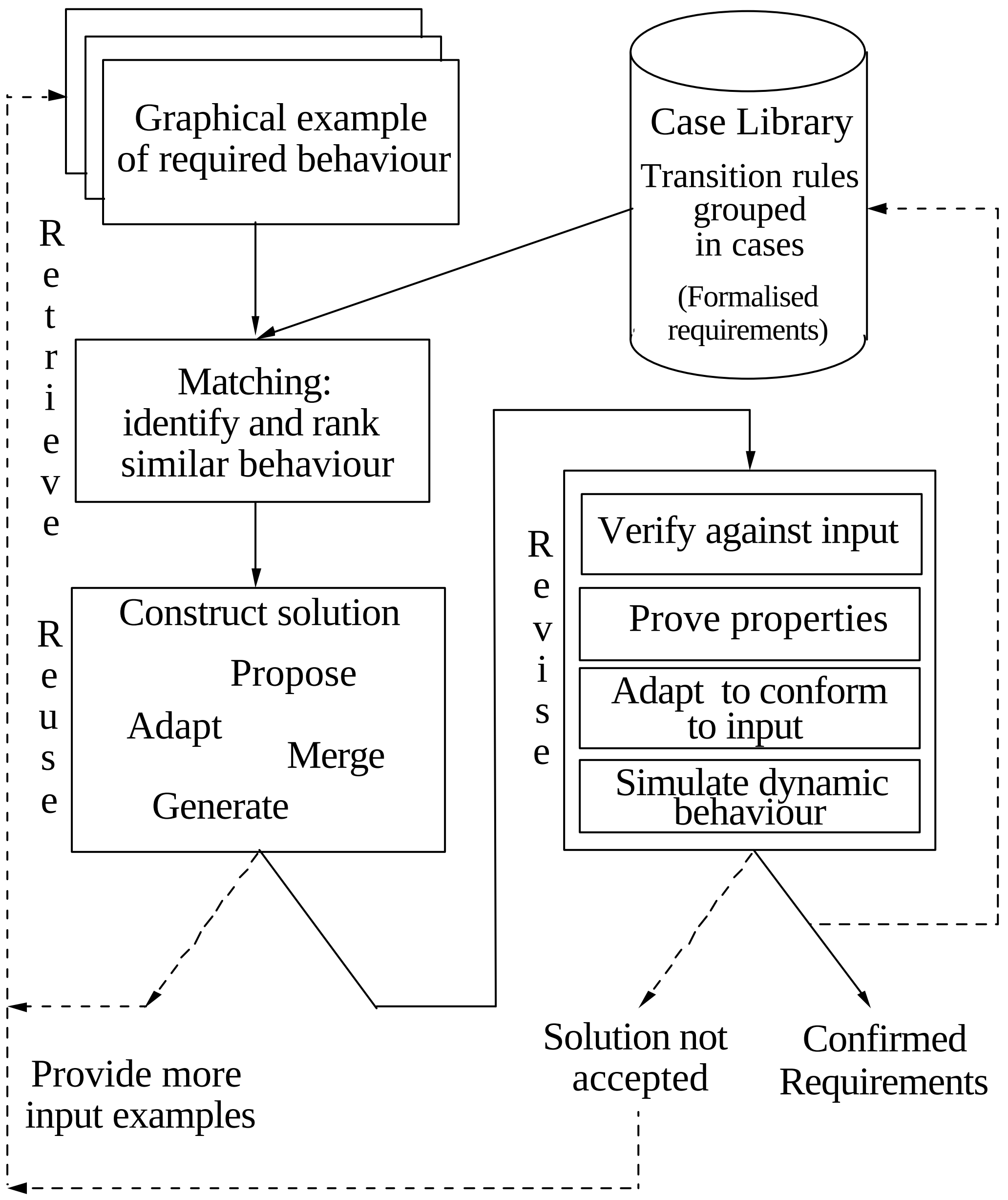

Graphical example of required behaviour
Case Library
Transition rules grouped in cases
(Formalised requirements)
Retrieve
Matching: identify and rank similar behaviour
Reuse
Construct solution
Propose
Adapt
Merge
Generate
Revise
Verify against input
Prove properties
Adapt to conform to input
Simulate dynamic behaviour
Store
Provide more input examples
Solution not accepted
Confirmed Requirements

Figure 3.1: Outline of the CABS approach

For some application domains, the ultimate goal may be to use the formalised and confirmed requirements directly as an implementation. This is possible for a very narrow class of application domains, where the interface to the environment (stimuli/response) of the requirements specification of the system is expressed on the same abstraction level as the final system itself and where the final system has to be implemented on a computer (which is not the case for telecommunications services where stimuli/response are commonly expressed on higher abstraction levels). If so, a requirements specification including all the desired behaviour and excluding all unwanted behaviour might be used as the final implementation. For the application domain of telecommunications services there are high demands of efficiency on the final code. The requirements could be seen as the tip of the iceberg and the final implementation is a highly optimised and integrated system of software and hardware in a global network of co-operating telecommunications switches. In these circumstances, the requirements specification is used as input to the design process and for generating test sequences for verification.

In conclusion, CABS is aimed at providing a closely integrated approach to requirements design and supporting iterative refinement, re-use and revision to produce formalised, validated and verified requirements specifications capturing the required behaviour of the system to be constructed.

Chapter:

4. Graphical Input Examples Exemplifying Behaviour

It is common to apply graphical notations to a number of different tasks in specification and design processes. In telecommunications, graphical notations are widely used, examples of which are SDL (a graphical Specification and Description Language, standardised by the International Telecommunications Union [ITU-Z100]), MSC (Message Sequence Charts), traditional flow charts, etc. Most notations used in specification have been formalised to a greater or lesser extent and are mostly used for design reflecting the chosen implementation structure (MSCs capture signalling between nodes assuming the services are implemented with communicating entities). CABS uses a graphical notation to capture behavioural examples (see Figure 4.1), which outlines different parts of some required behaviour, but does not aim to compete with the large area of ongoing research on graphical formalisms. The graphical notation used is only intended to capture some of the externally visible behaviour (any requirements specification should not put demands on how the behaviour is implemented internally [Wieringa 96]) and internal signalling or communicating entities can purposely not be expressed in the formalism.

Graphical formalisms for behaviour can mostly be classified as state based, transition based, transaction based or any combination of these. The full behaviour of a telecommunications system contains too many states to be handled graphically (even if there are only a few telephones involved), without introducing levels of abstraction for states. Therefore, it is difficult to base telecommunications requirements specifications directly on state transition diagrams: state transition based formalisms are mainly used in domains with less then a few thousand states, preferably less than a few hundred states if they are produced and maintained by humans. If there is no abstraction of states, the number of different states in the telecommunications domain will be so large that it will be difficult for a user to handle. From a computational point of view, there would be no problems with this application domain since the specified behaviour for telecommunications services is simply that they should be finite and deterministic. The purpose of the graphical notation is simply to outline the main characteristics of the behaviour (and not to describe all possible behaviour) and it therefore bypasses the need to handle large numbers of states; the graphical notation is a starting point for the production of formal requirements.

For CABS, a graphical transition based formalism has been chosen. The graphical examples in the CABS system are used in the initial stage of rapidly putting together a draft specification, and arriving at an executable specification, so that initial ideas about the required behaviour and their corresponding examples can be refined and validated. The graphical input examples are also used together with the information added during the refinement of the input examples to provide automated assistance in verification. It contains nodes (ovals) and directed links (arrows) which will be explained in detail in sections 4.1 and 4.2 respectively. Nodes and links are given names (links have their stimulus name in a square box, where a stimulus is the external event that triggers a transition from one node to another, if all other conditions are met) and pairs of nodes can be connected by links in any way. A new node is created by selecting the *create node* tool (the first tool in the tool list in Figure 4.1) and a new link is created by selecting the *create link* tool (the second tool in the tool list). For nodes and links, an additional window with details about the node or link can be shown. This window is shown when the *details* tool (third tool in tool list) is chosen and the node or link is selected by clicking on it. A node can be moved by choosing the *move* tool (the fourth tool) and dragging the node to the new position (all links to/from the node will automatically be updated). A node can be renamed/replaced and a node or link can be deleted by selecting the corresponding tool (fifth, sixth respective seventh tool), and then selecting the node or link (any links to/from a deleted node will automatically be deleted). The graphical representation and editor are designed to be uncomplicated, general and deliberately unlike other graphical formalisms used in telecommunications since their aim is different and similarities may confuse matters. Graphical input examples also have a non-graphical representation (with some additional information about the input example), which can be examined by the user by selecting the information tool (eight tool from the top in Figure 4.1) which results in the display of a window with details of the input example as shown in Figure 4.2. The ninth tool is used to redraw the window and the last tool matches the input example against the case library.

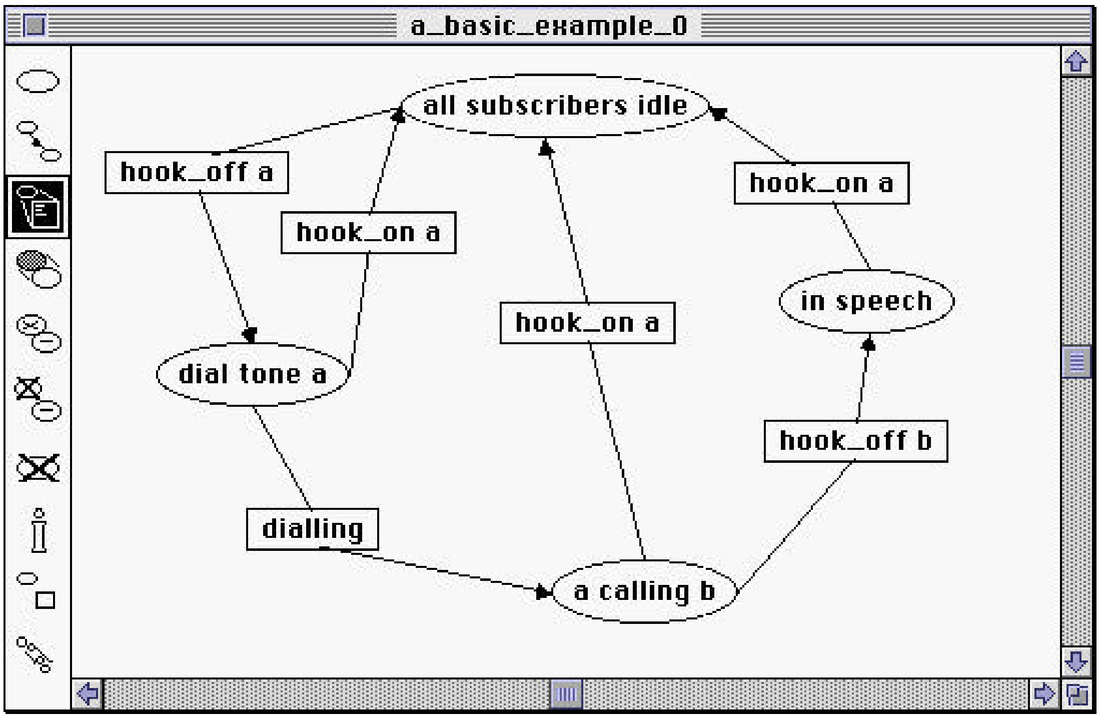
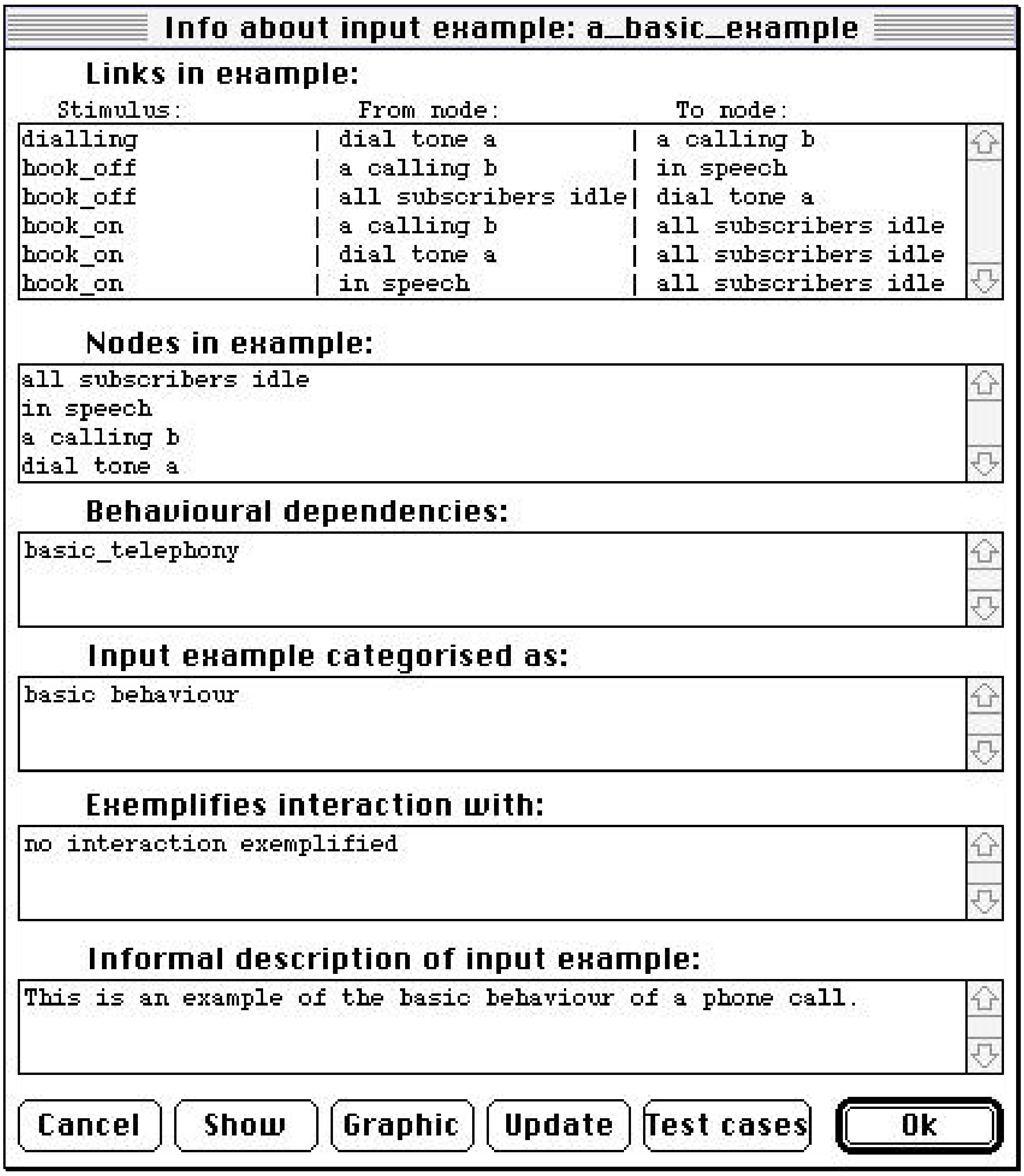


Figure 4.1: A graphical input example exemplifying a basic behaviour for the service *basic call*

The non-graphical window for the input example (Figure 4.2) contains a scrollable list, *Links in example*, with all the links in the input example and information of triggering stimulus, start node and end node. A scrollable list, *Nodes in example:*, contains all the nodes in the input example. These two lists capture all the information shown graphically in Figure 4.1. Selecting a node or link in these lists and then pressing the *Show* button will show a window describing the node or link in detail, as described in Sections 4.1 and 4.2 (this window is also accessible through the *detail* tool in Figure 4.1).

Some of the functionality may be dependent of the functionality of some previously specified service. When creating a new input example, the user states the services on which the new behaviour is obviously dependent: for example, the *three way call* service is often defined as an extension of the *call waiting* service, and if *call waiting* is not available, *three way call* cannot be used on its own. These services are listed under *Known behavioural dependencies:* and are called behavioural dependencies to distinguish them from more subtle dependencies (see Section 5.1) which, in some cases, can be identified automatically in CABS. Structuring services as being dependent on other services is common practice for telephone services. In CABS, this information is used in the matching process where cases on which the behaviour is dependent should be included as proposed solutions.



Info about input example: a_basic_example

Links in example:

```
  Stimulus:                From node:              To node:
dialling             | dial tone a            | a calling b
hook_off             | a calling b            | in speech
hook_off             | all subscribers idle   | dial tone a
hook_on              | a calling b            | all subscribers idle
hook_on              | dial tone a            | all subscribers idle
hook_on              | in speech              | all subscribers idle
```

Nodes in example:

```
all subscribers idle
in speech
a calling b
dial tone a
```

Behavioural dependencies:

```
basic_telephony
```

Input example categorised as:

```
basic behaviour
```

Exemplifies interaction with:

```
no interaction exemplified
```

Informal description of input example:

```
This is an example of the basic behaviour of a phone call.
```

Figure 4.2: Textual representation of input example

Informal examples of behaviour given in textual requirements specifications of a telecommunications service are often categorised in some way for convenience of reference. We have not investigated which categories are most commonly used, but have implemented a facility for defining categories. Five different categories have been selected (categories can be added/removed to suit the application domain): *basic behaviour; odd case; error case;*

unsuccessful behaviour; excluded behaviour. An input example may be classified as being in more than one category. The user selects the categories when creating a new input example and the categorisation is shown in the text list field *Input example categorised as*. In Figure 4.2, the input example *a_basic_example* is categorised as *basic behaviour*. Categories may aid the user in the process of structuring behavioural requirements. The classification may also be used to assess whether the user has given sufficient input examples, or if the system should request more input examples. If an input example exemplifies excluded behaviour, it should be handled differently in the matching, validation and verification process. Excluded behaviour (negative examples) has not been fully implemented in CABS (see the discussion in Chapter 10).

Interaction between behaviours is of central concern in telecommunications and is often claimed to be the most severe problem in developing and managing telecommunications systems [Zave 93][14]. If the behaviour of a telecommunications service is modified when some other service is active/inactive, or if it modifies the behaviour of some other service when it is active/inactive, we say then that the two interact. Interaction between services is not "*a problem that can be solved*" since it is part of the required behaviour, therefore decisions on how services interact have to be made before or during implementation. When the user adds a new input example, she can select what interaction the input example exemplifies, and the selected services are shown in the text list *Exemplifies interaction with:* in Figure 4.2. In input examples, it is more likely that the desired interaction is exemplified, leaving unwanted interaction to be handled when the full specification is produced (including all the desired behaviour and excluding all the unwanted behaviour). If the unwanted behaviour is exemplified as an input example, it is categorised as negated input examples. A negated input example can be used if there is some specific behaviour that

[14] Some interaction between services may be introduced by architectural/implementation choices such as dividing the system into communicating processes [Cameron, Velthuijsen 93], and is not relevant on a requirements specification level.

should not be allowed (this may be needed in the telecommunications domain when services interact, but may also be useful in other situations). Negated input examples are considered a useful extension, and may, in some situations, further improve matching/verification results, but are not classified as essential to the approach and have not been implemented in the prototype.

An informal textual description of the input example together with relevant links can be provided by the user in the text field *Informal description of input example*. This information is used for the convenience of the users and for documentation. The *Graphic* button shows the window with the graphical representation. The *Update* button is used to update any changes (the graphical window is updated dynamically).

4.1 A Node

Each node has a unique name that is a mnemonic name for a situation, such as two telephone users being in speech connection with each other (the oval *in speech* in Figure 4.1).

A situation can encompass many different states[15], for example the node *dial tone a* (details for this node are shown in Figure 4.4) may intuitively mean that the user *a* has a dial tone, which may be true for many different states. In a telecommunications system, there may be millions of different states where the user *a* has a dial tone, but most of the differences will be irrelevant for any particular example.

[15] A state is defined as a unique description of a system's current status, as used in state based approaches, where each state is often given a unique number. A finite state machine is an example of a frequently used state based formalism.

4.1.1 Creating Nodes

When the user creates a new input example, the first step is to create some nodes. The user selects the first tool from the list of tools on the left in Figure 4.1. The user then clicks on the position in the graphical area where the node is to be placed. A window where the user can select the node name appears (Figure 4.3). If the user chooses to use a node that has been specified earlier in some other input example, she clicks on the selected node and presses the *OK* button. If in doubt, the *Details* button can be pressed in order to inspect the selected node. Ideally there is always a suitable node to select. If a new node name is given, the details for this new node can be specified as explained in section 4.1.2 when the *Details* button is pressed.

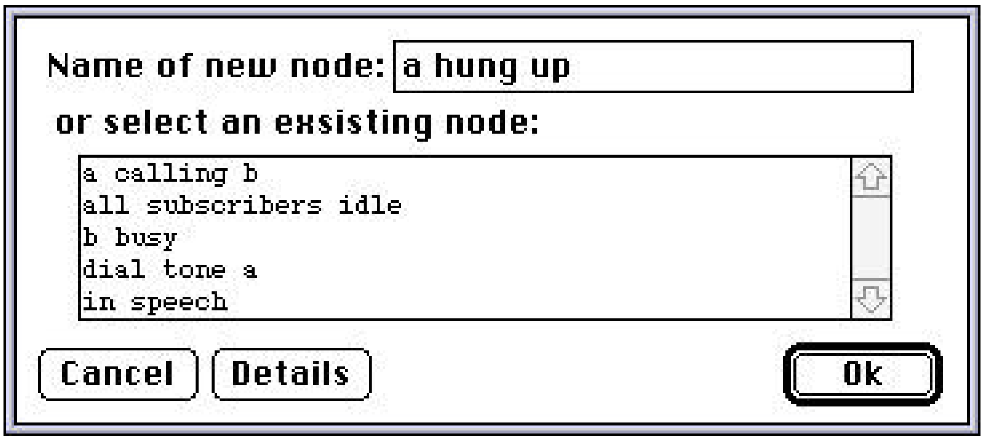

Name of new node: a hung up
or select an exsisting node:
a calling b
all subscribers idle
b busy
dial tone a
in speech

Figure 4.3: Select node name for input example

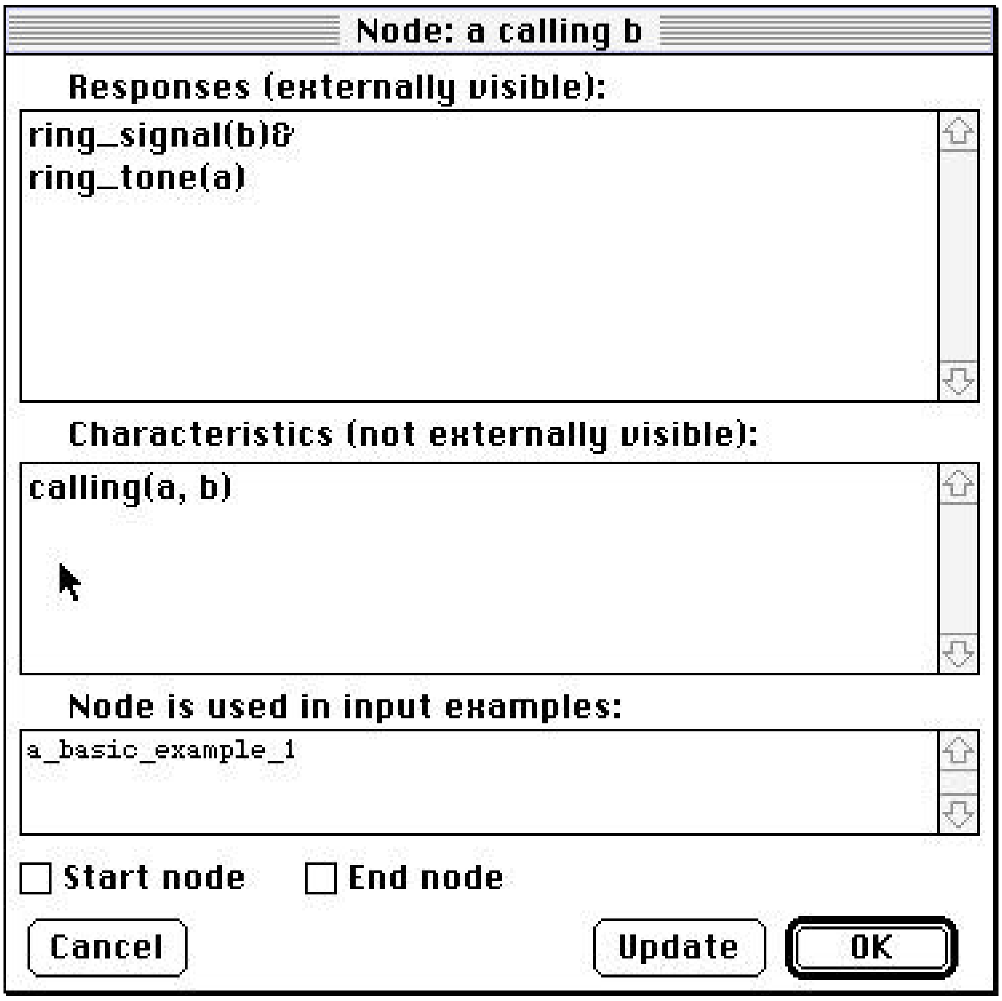
After the user has pressed the *OK* button, the node is drawn as a circle with the name in the graphical area (Figure 4.1).

4.1.2 Details for Nodes

When a telecommunications expert talks about a specific situation such as two subscribers being in speech connection (represented by a node in CABS), the user normally has a comparatively well defined meaning in her mind. Unfortunately, it often happens that different telecommunications experts do not necessarily have the same meaning in their minds; hence, a more precise description of a situation is needed. In CABS, a more fine grained definition of a node is given as a conjunction of terms. Terms are explained in detail in Section 5.1 (the following example may be sufficient to provide a basic understanding). To add to or modify the details of a node, the user selects the *detail* icon in the graphical window (the third icon on the left in Figure 4.1) and then clicks on the chosen node in the graphical window. This appears in a node window, as shown in Figure 4.4. If no details have been given for this node, all fields will be empty. The user can now select the terms (by selecting them from a menu or by typing them into the field) that are expected to hold for this node, and add them in the corresponding field. For example, for the node *a calling b,* the terms *calling(a, b)* and *ring_tone(a)* and *ring_signal(b)* are expected to hold (terms may also be negated). The first predicate term, *calling(a, b),* is a relation between user *a* and user *b,* stating that user *a* is calling user *b;* the second term states that user *a* has a ring tone and the third term states the fact that user *b*'s telephone is ringing. A *relation* term is by definition not externally visible and is therefore added in the field *Characteristics (not externally visible).* The two terms *ring_tone* and *ring_signal* are defined as response terms and are therefore, by definition, externally visible and added in the field *Response (externally visible).* In telecommunications systems, externally visible effects are so central that response terms (externally visible terms) are often treated separately, even on a requirements specification level.

The same node may be used in different input examples, and the input examples in which the node is used will be shown in the list *Node is used in input example.* If a node has to be modified, the user must make sure that the change is valid for all other input examples using the same node or, if not, they must choose a different name for the node and define this new node.

When giving input examples, it is obvious to the user in most cases which node is the start node and which is the end node (there may be more than one). Intermediate nodes are nodes that are temporarily passed through in order to achieve some required result. The user can specify whether a node is a start node, an end node, both or neither (if a node is neither a start node nor an end node, it is an intermediate node). In CABS, this selection is made by ticking the corresponding box in Figure 4.4. This information is useful in the verification process and in order to automatically generate test cases which will capture all behaviour between the start and end nodes (this narrows down the number of test cases considerably and in fact, in a large system, the number of test cases would be difficult to handle without this information; for more on this, see Chapter 7).


Node: a calling b

Responses (externally visible):

ring_signal(b)&
ring_tone(a)

Characteristics (not externally visible):

calling(a, b)

Node is used in input examples:

a_basic_example_1

☐ Start node ☐ End node

Figure 4.4: Example of a detailed node description in CABS

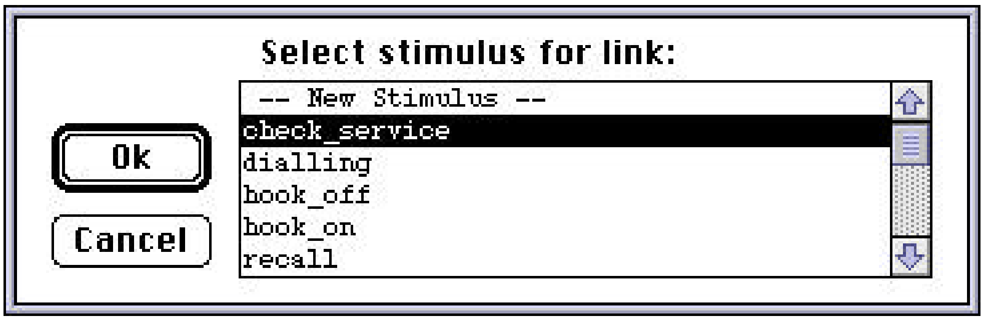
As mentioned, the user is expected to give the main characteristics of a node (by choosing from a list with all terms that have been defined in the case library), excluding facts of less relevance for the node. In most cases, such a brief description of the main characteristics will be sufficient, since the input is used primarily for identifying similar behaviour in the case library and for the final verification after the requirements have been formalised. In cases where there is no good match (a new type of behaviour with no similar case in the case library), the input examples are used as a starting point for generating a new case. However, in these situations, the input may need refinement. From this point, whenever we mention input examples, or graphical input examples, we mean both the diagram itself and the details given on nodes and links.

If all terms have a natural language phrase declared, the user could choose to use natural language (NL) phrases instead of terms. For example, if *calling(A, B)* has the NL phrase *A is calling B,* this phrase could be shown in Figure 4.4 in the field *Characteristics (not externally visible).* An NL translation would be useful for users less familiar with formal notations and if the examples were shown to customers, they may not wish to see brackets at all. The way in which formulae of terms can be translated into natural language phrases has been explored in depth [Dalianis 96]. In CABS, NL phrases have not been implemented but this is proposed as an extension (adding a prototype implementation of NL phrases would require little effort).

4.2 A Link

A link in the input example describes a transition from one node to another. The main condition for the transition to take place is that the stimulus term in the link occurs. A stimulus in the telecommunications domain may, for example, be an action performed by a phone user, such as lifting the receiver (*hook_off*) or dialling a number (*dialling*) as shown in Figure 4.1. In the graphical notation it is optional to show arguments for a link. When looking at the details for a link, all arguments to a stimulus are shown (for example in *dialling(A,Nr,T),* the first argument is the phone user dialling, the second argument is the number dialled and the third argument is the time this occurred). See Section 4.2.1, Figure 4.6 and Section 5.1 for more on arguments.

When the user wishes to add a new link between two nodes, she selects the second tool from the list of tools on the left in Figure 4.1 and then clicks on the node from which the link will originate. Then, she clicks on the terminating node (a broken line is shown between the originating node and the cursor until the terminating node is selected). When the terminating node has been selected, a window for selecting the stimulus term for the link is shown (Figure 4.5). The user can select a stimulus term from the list showing all stimuli terms defined in the case library. If the item -- *New Stimulus* -- is selected, the user can add the name of a new stimulus term. The user may define the stimulus term in detail, as described in Section 5.1 (this should be done before the input examples containing the new term are used in the matching).


Select stimulus for link:

-- New Stimulus --
check_service
dialling
hook_off
hook_on
recall

Figure 4.5: Select stimulus name for new link for input example

When the stimulus term has been selected, the new link will be drawn between the two nodes and the name of the stimulus term will be shown in a box in the middle of the link. When all nodes and links have been put in place in the input example (as shown in Figure 4.1), the input example gives all stakeholders a graphical sketch of the required behaviour.

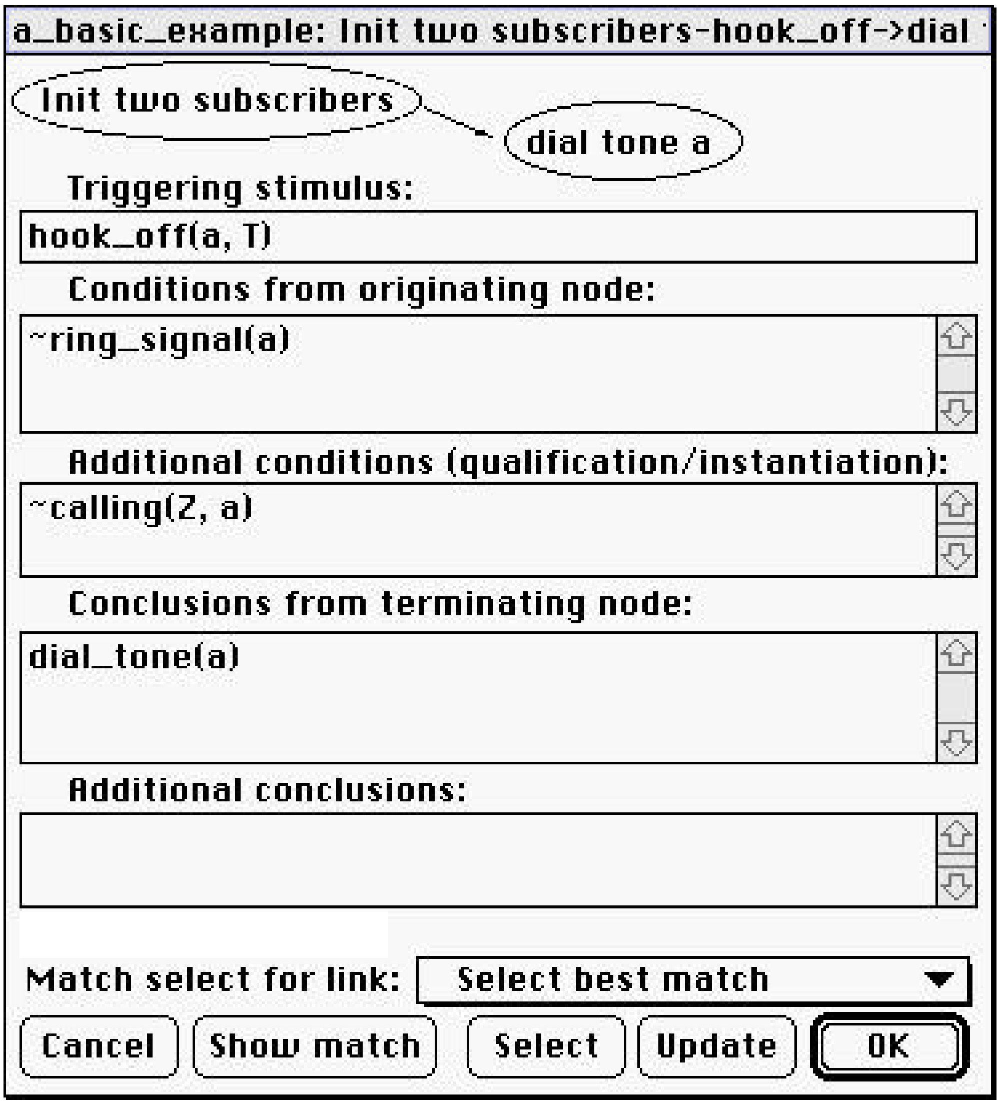
4.2.1 Defining or Refining Links

A link is identified by its originating node, its terminating node, its stimulus term and the input example in which it is used. In our examples, the triggering stimulus name is always used as the name of the link. We refer to a link by the name of its input example followed by the originating node name, the triggering stimulus name and the terminating node name and, therefore, there is no practical need to introduce unique names for links. In some situations, a link needs some added details in order to reflect the user's intention for the transition between the originating and terminating nodes. The details for a link are added in the same way as for nodes (by selecting the detail-tool and clicking on the link in order to get a link window as shown in Figure 4.6). In the link window, we draw the originating node and terminating node. The first edit field is the stimulus term, with its arguments extracted automatically from the definition of the term.

In CABS, the terms of the originating and terminating nodes are put, by default, into the corresponding scroll edit fields in Figure 4.6 (*Conditions from originating node:* and *Conclusions from terminating node:*) when a new link is created. The user deletes the condition and conclusion terms that seem to be irrelevant or of low significance, bearing in mind that the link will be used to identify a matching case in the case library.

Additional conditions in Figure 4.6 (field *Additional conditions (qualifications/ instantiation):*) are there to allow the user to add some specific conditions not explicitly given by the originating node. In some cases, additional conditions may be added to discriminate between two links with the same stimulus term leaving the same node. For example, if user *a* lifts the receiver and receives a dial tone, she should not currently be called by some other user (if she lifts the receiver when called by another used she would answer the incoming call, this can be exemplified with another link). This additional condition *~calling(Z,a),* not explicitly stated in the field *Conditions from originating node,* is put in the field *Additional conditions (qualifications/instantiation).*

Additional conclusions in Figure 4.6 (field *Additional conclusions:*) are there to allow the user to add some specific conclusions not explicitly given by the terminating node (no additional conclusions are given in Figure 4.6). Added conclusions may be facts to be carried forward in time and used at a later stage in the telecommunications service or used by some other telecommunications service such as *Charge Advice*. An example of a fact needed at a later stage is *which user originated a three way call* (the service *three way call* is specified such that if the person who originated the call hangs up, the other two connections are cancelled so that no confusion about who is paying for the call may arise). This fact can be added as an *Additional conclusions: three_way_call_originator(User)* when the three way call is initiated.



a_basic_example: Init two subscribers-hook_off->dial

Triggering stimulus:

hook_off(a, T)

Conditions from originating node:

~ring_signal(a)

Additional conditions (qualification/instantiation):

~calling(Z, a)

Conclusions from terminating node:

dial_tone(a)

Additional conclusions:

Match select for link: Select best match

Figure 4.6: An example of a detailed transition link description in CABS

The pop up menu *Match select for link:* and the buttons *Show match* and *Select* are first relevant during and following matching as is explained in Chapter 7. If a link does not generate any good matches, the user may decide to refine an input case by revising/refining the links (by adding/removing appropriate terms), which hopefully results in a better match. Other ways of improving the matching results are explored in Chapter 7. The *Update* button confirms any changes made in the edit fields and the previous definition of the link is replaced.

4.3 The Use of Graphical Input Examples in CABS

Initially, every case (the required behaviour for a telecommunications service) originates from a number of graphical input examples. These input examples represent the original behavioural requirements for the case even if the case itself captures more behaviour than exemplified in the input examples (the case may have been refined during validation and integration). We store the input examples for each case in the case library for a number of reasons:

Input examples are used to automatically generate test cases and verify that the final solution (the formalised requirements) captures the behaviour exemplified in the input examples (explained in Chapter 7).

Generated test cases are also used to verify the interaction with other cases (explained in Chapter 7).

If the behavioural requirements for a case are changed, this change will be made by altering the graphical input examples.

We may re-use input examples as a starting point when we specify the behavioural requirements of a new case (input examples can be copied and renamed in CABS).

The input examples may be used for understanding, learning and documentation of the telecommunications system produced.

In Chapter 5, a detailed description is given of exactly what is stored in the case library, and how relevant information is defined, updated and shown to the requirements designer.

Chapter:

5. Case Library

The case library is a central part of CABS. It is intended to contain everything that is needed for the process of formalising the required behaviour (a subset of the total behaviour of the system when it has been fully implemented) such as initial requirements, informal and formalised definitions, test cases used for verification and relations between these objects. To make CABS's internal representation easy to extend and modify, the case library is organised in an object-oriented fashion where each instance can be uniquely identified and has a number of attributes and methods assigned to it (for example see [Bose 94]). Figure 5.1 shows an overview of the case library and the relationships between the main parts within it. The relationships shown as broken lines have not been implemented in the CABS system (beyond the scope of the research) and are only shown to give the context. All the main objects in the case library have attributes such as creation and modification dates, informal description, etc. These organisational issues and design decisions are all hidden behind the user interface and the case library will be described as seen through the user interface. Everything in the case library can be saved and loaded between sessions.

The case library comprises six main sections: case definitions, transition rule definitions, term definitions, test cases, graphical input examples and system definitions. A system definition (top left corner of Figure 5.1) is basically a set of cases capturing all the required behaviour the system is expected to exhibit when it has been implemented, including the more common interactions between these cases. An add-on system is a set of cases that adds some particular functionality to a system, for example the system *mobile_telephony* or *ISDN_telephony* (Integrated Service Digital Network) adds behaviour to the system *basic_telephony* (see glossary in Appendix D). A case (a telecommunications service) captures the required behaviour of some particular functionality in a system and is shown in the centre of Figure 5.1. The behaviour of a case is represented by a set of transition rule definitions (middle left of Figure 5.1) and definitions of terms (below centre) that are considered to belong to that case. Graphical input examples (top right of Figure 5.1) exemplify the initial required behaviour of a case and the more common interactions with other cases. If a case is added or modified, the interaction between this case and the other cases needs to be analysed and may need to be verified again (see Chapter 7). All test cases (bottom right of Figure 5.1) that capture the required behaviour extracted from the input examples, are stored for use in the automated verification process. Once a case and required behaviour have been designed and implemented (the implementation of a new telecommunications service may be a combination of software and hardware such as *three party call* needing specific hardware connecting three phone lines to each other), the connection between the transition rule definitions and term definitions should be kept (these links are shown in Figure 5.1 as dotted lines). These links, shown as dotted lines, are beyond the scope of this research.

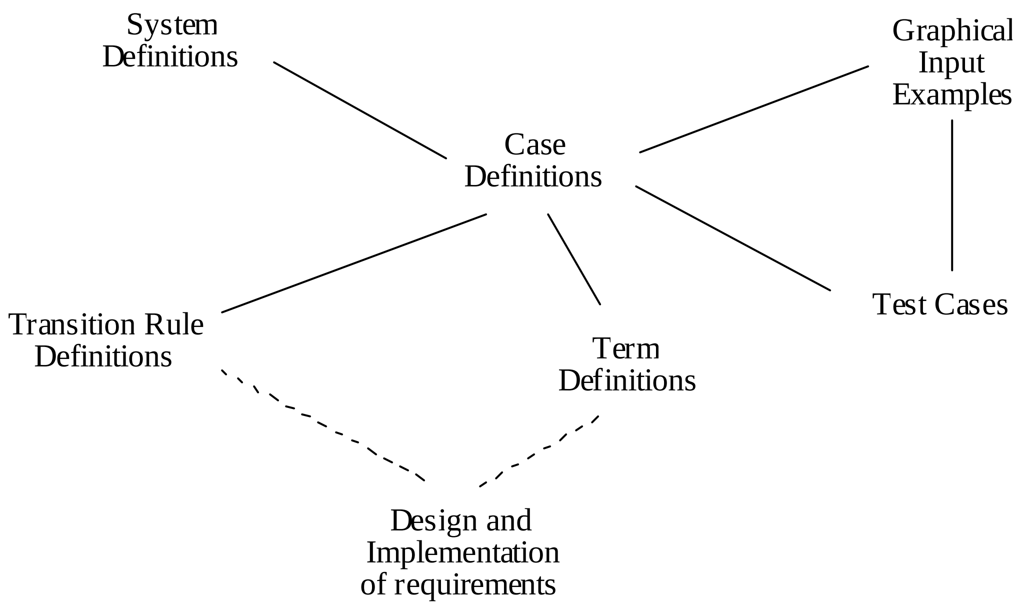


Figure 5.1: Overview of case library

In the following sections, we will describe the different parts in the case library, their use and how they are defined or modified by the user. First, we describe *terms,* which are one of the most fundamental parts in CABS, then *transition rules,* which represent the dynamic behaviour of cases, then *cases* (telecommunications services in our application domain), *systems* (sets of cases) and, finally, we describe *graphical input examples* and *test cases*. Once all the parts of CABS are explained, Chapter 6 explores how similar behaviour can be identified by input examples and using them in a matching process, in order to identify cases that may be re-used in whole or in part.

5.1 Terms

The purpose of terms is to capture a system's current state. Terms are used both in input examples and in transition rules and are an important part of defining an ontology for the domain[16]. A precise and clear meaning for each term is crucial to the interpretation and understanding of a formal specification, although few requirements methods address this issue effectively [Zave 96]. Also, if a term is used in an input example, it is important that the term is well understood by the user so that the input examples and the cases in the case library are built on the same terminology. In CABS, the user is expected to define terms with care and in detail before the term is used in input examples and in transition rules. Term definition should be one of the first tasks when approaching a new application domain or a new class of behaviour that cannot be expressed with existing term definitions. If a term does not have a clear meaning, or its meaning is modified during a specification, all previous specifications are no longer valid and have to be re-validated by the user. For a large system, where the specification may have hundreds of cases and thousands of transition rules, this will be a tedious and time consuming task. If a term's meaning in CABS is changed for some reason, all this work has to be repeated. The idea is to give elaborated definitions of the meaning of terms in order to reduce the risk of introducing problems at an early stage, which may cause costly corrections later on. Informal discussions with telecommunications experts have shown that experts sometimes disagree on the meaning of terms they use: large efforts are put into standardisation of telecommunications terminology both by telecommunications companies and international standardisation organisations, but if terms are properly defined the first time they are used, some of these efforts may be reduced.

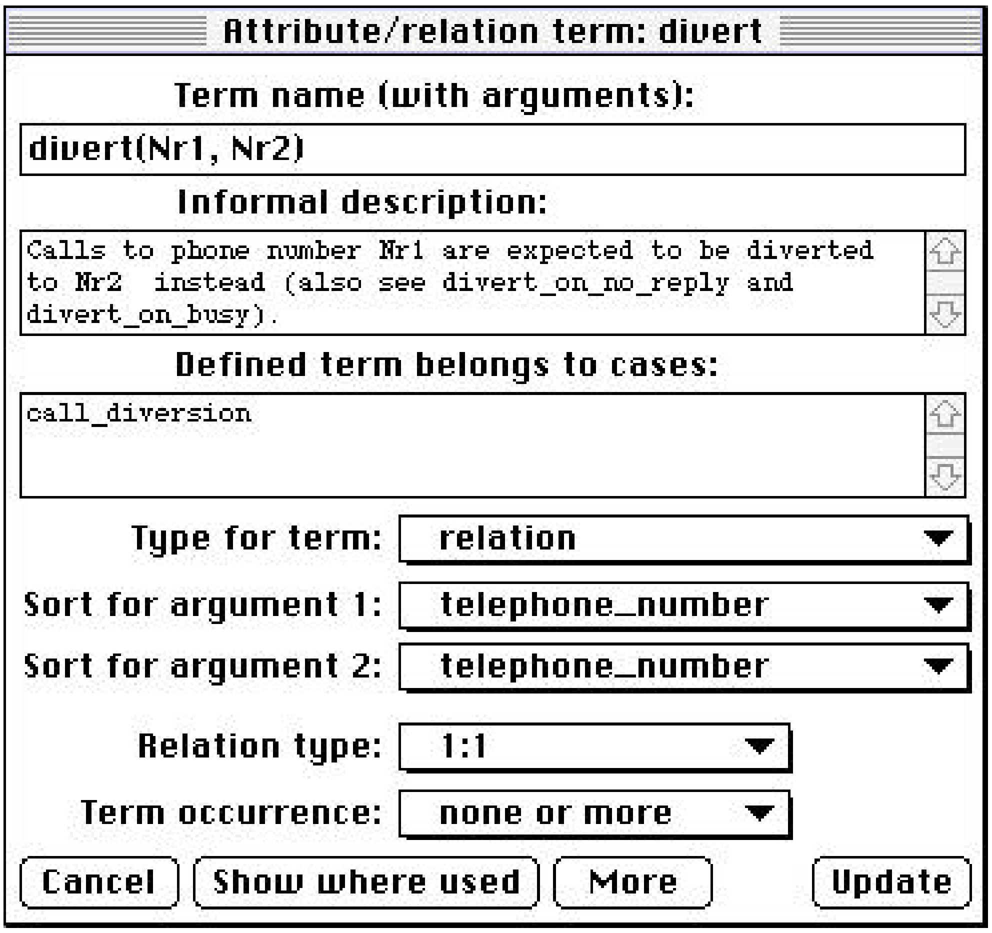
Figure 5.2 shows an example of a term definition in CABS. The purpose of the current way of defining terms is not to compete with ongoing research in conceptual modelling (see for example [Johannesson, Boman, Bubenko, Wangler, 97]). However, Figure 5.2 may provide an alternative way of presenting some of the information traditionally captured in conceptual modelling. The examples merely give an illustration of the different pieces of information of interest for formalisation/validation/verification and exemplifies how this information can be

[16] Defining an ontology is beyond the scope of this research, only a few aspects of defining an ontology are addressed, for more details on ontologies, see for example, [Uschold 95].

collected at an early stage of requirements capture. The content of Figure 5.2 will be explained briefly now and explored in more depth in sections 5.1.1 to 5.1.6.

The first field, *Term name (with arguments):* in Figure 5.2 is the term name and arguments; in this example *divert(Nr1, Nr2)* is typed in by the user when defining the new term (argument names must start with a capital letter and can contain any number of letters, numbers and underscores). The next field, *Informal description:* is an informal description of the term and arguments. The list *Defined term belongs to cases:* shows which cases in the case library the term definition belongs to. The most common situation is that a term is only defined in one case. On some occasions, it makes sense to let the same term be defined in more than one case, for example, if there are two varieties of the same case in the case library. This occurs in telecommunications since services are often adapted for different customers and markets (the service *three party call* for regular customers is different from *three party call* for Centrex, see glossary in Appendix D). Terms can be of four types in CABS: stimulus terms, response terms, relation terms (more than one argument) and attribute terms (zero or one arguments). When defining a new term, the user has to select the term type by selecting the appropriate type in the pop-up menu under *Type for term:* in Figure 5.2. The user can also choose a sort for each of the term's arguments (Figure 5.2), *Sort for argument <position number>*. The maximum number of arguments is restricted to five in the implementation of the interface, which is sufficient for the current examples in the application domain and should also be sufficient for the telecommunications domain. The size of the window is adapted automatically to the number of arguments in the term. For each term, the type of relation between the arguments can be specified by selecting the appropriate choice in the pop-up menu *Relation type:* in Figure 5.2. The set of types available depends on the number of arguments for the term: if there are zero or one arguments, the selection cannot be made. With the pop up menu *Term occurrence:* the user can select whether a term has any restrictions on its occurrence. The options for terms with one or more arguments are *one*, *any*, *zero or one*, *one or more*. Option *one* would mean that if the system can reach a state (see section 5.1.6) in which the term exists more than once or not at all, then there is a conflict between the definition and the transition rules leading to the state. For example, if the terms *current_time(1)* and *current_time(2)* are true at the same time, it is incompatible with this definition. This sort of generic information is often easiest to capture when the user defines a new term and can then be used in a number of different ways. For example, if new transition rules are generated from links or adapted from other transition rules, they can be inspected for consistency with the term occurrence definition. This information can also be used when verifying a system (see Chapter 7.6).

The button *Show where used* produces a cross reference list of all transition rules in the case library and tells the user which cases contain transition rules that use the term in their condition-part/conclusion-part (currently this is not fully implemented but it could be implemented with a simple search function). The *More* button gives some additional information, such as the times at which a term definition was created and last modified. The *Update* button updates any changes of the definition (if the user has the privilege of modifying term definitions). The *Cancel* button ignores any changes and leaves the term definition unchanged.


Attribute/relation term: divert

Term name (with arguments):

divert(Nr1, Nr2)

Informal description:

Calls to phone number Nr1 are expected to be diverted to Nr2 instead (also see divert_on_no_reply and divert_on_busy).

Defined term belongs to cases:

call_diversion

Type for term: relation

Sort for argument 1: telephone_number

Sort for argument 2: telephone_number

Relation type: 1:1

Term occurrence: none or more

Figure 5.2: An example of a term definition in the CABS system

5.1.1 Significance of Term Names

The terms used are in predicate logic (see for example [Luger & Stubblefield 89]) where the term names bear the main part of the non-instance-specific information. For example, if we would like to capture the statement that a user *a* has dialled the number 222 and that the number 222 has all its calls redirected to the number 333, that there is a user b answering calls on number 333, and c is not calling b, we could capture this with the four terms:

dialling(a,222) and redirect(222,333) and

answer_number(b,333) and not calling(c,b)

In this example, all the non-instance-specific information is captured in the term name and all instance-specific information is represented as arguments to the terms. The term names are relations or attributes that can be given a clear meaning from a phone user's perspective. The arguments are phone numbers (222, 333, 444, ...) and phone users (a, b, c, ...) which are the most central entities in the telecommunications domain.

In an entity relation model, in contrast, terms are of the form *is_a* or *has_property*, and most of the significance is shifted to the arguments. An example with low significance in term names and high significance in the arguments would be:

has_property(a,dialling,222) and has_property(222,333) and

has_property(b,answer_number,333) and not has_property(c,calling,b)

In this example most of the significance has been shifted from the term names to the arguments. Both examples contain the same information when we know the instances and in most applications, the choice between the two representations may not be of any significance. But in our approach, it will prove important as will be explained in Chapter 6 (part of the matching is based on term names and is independent of the current domain of discourse). Term names are central to the matching process and if their significance is low, this will affect the validity of the matching result.

5.1.2 Instances, Arguments and Sorts

In behavioural input examples, requirements specifications and simulations, a set of instances are needed (to be precise, names uniquely identifying the real instances in the domain of discourse, i.e. all the telephones and phone numbers). This is not to be confused with the application domain (such as telecommunications services). Instances can be classified into sorts; in the application domain of telecommunications, there are sorts such as telephone numbers, phone users, etc.

In CABS, it is an advantage to use terms with few arguments as this often gives the term name higher significance. In fact, everything that can be expressed using terms with more than two arguments can be represented using terms with only two arguments; but this may look odd even if there are advantages in doing so. For example, the facts *answer_nr(User, Nr) & accepts_incoming_calls(User)* could be represented with one term and three arguments, *user_info(User, Nr, 'incoming_calls')*. If the term *user_info(User, Nr, F)* occurs in a node, link or transition rule, a careful analysis of the arguments will tell us which information is relevant to the situation. Since our matching algorithm uses term names as its main guide in identifying relevant matches, the matching result will be more accurate if terms use fewer arguments (for details on matching see Chapter 6).

From a pragmatic point of view, any non-trivial specification will initially contain mistakes, misunderstandings and errors. Sort declarations may be used in a number of different ways to aid the requirements capture process and, hopefully, to improve the accuracy of the final specification. The most common use is to identify any mismatch with sorts and point out where these occur. The argument against sort declarations and typing is mainly that in prototype systems and small specifications made by one or a few persons, the gains are not large enough to justify the additional workload. In our approach to specification, we acknowledge both the need for an early prototype of the requirements (i.e. to arrive quickly at some intermediate result that can be partly validated and verified in order to aid the refinement and revision process) and the need to produce a validated and verified formal specification outlining the required behaviour. CABS provides, as an option, the default sort *Not specified* when selecting the sort (in *Sort for argument <argument number>* in Figure 5.2), which has all other defined sorts as a subset. This will allow the user to specify, simulate and refine the system incrementally and to decide when to declare this information. A new specification should only be accepted when all arguments have their sort declared (checking that all arguments have their sorts defined is trivial to implement, although not implemented in the CABS prototype, see for example [Cohn 85]). Furthermore, theorem provers and simulators can improve their performance by using sort information.

5.1.3 Constraints on Terms

There are a number of static constraints that can be declared on terms (static since they are valid for all states the system can reach). Much research effort has been put into the modelling of static models: entity relationship modelling is one of the most popular approaches [Wieringa 96]. A number of different graphical notations are also used and some are translated into logic [Preifelt, Engstedt, 93] or into logical programming languages such as PROLOG [Johaneson 91]. Examples of constraints on terms in the telecommunications domain are:

A user can have only one other person calling (next caller gets busy tone).

A user can have only one *last called number* (used when the *redial* service is activated).

Only one current time can exist in a given state.

This information is useful in the verification process for the specified system. A term can either be a propositional term, e.g. *lamp_is_on* or have arguments, e.g. *switched_on(lamp_1)*. A term can either be true or false: ¬ *switched_on(lamp_1)* means that it is not true that the lamp_1 is switched on. In the following sections, we will explore how to represent different aspects of terms and their properties (such as the three examples above) in more detail.

Each term is defined according to an approach similar to that used in some entity relationship approaches (for more details on different approaches see [Wieringa 96]). In the current implementation of CABS, there is no graphical representation of static constraints for terms. The four static constrains defined in CABS are: type of relation for terms; sort of arguments; relation type between the arguments; term occurrence, as shown in Figure 5.2.

If there are two arguments for a term, the choices are *1:1*, *1:m* (m for many), *m:1*, *m:m* (see examples of the relation types in Figure 5.3). The relation type *1:1* means that each object in the set of possible values for the first argument can have only one relation with one object in the set of possible values for the second argument. The relation type *m:1* means that each object in the set of possible values for argument 1 can have only one relation with one object in the set of possible values for argument 2, and objects in the set of possible values for argument 2 can have many relations with different objects in the set of possible values for argument 1 (for more on this see, for example, [Davis 90]). This information can be used in various ways in verification and validation, or when adapting or generating new transition rules.

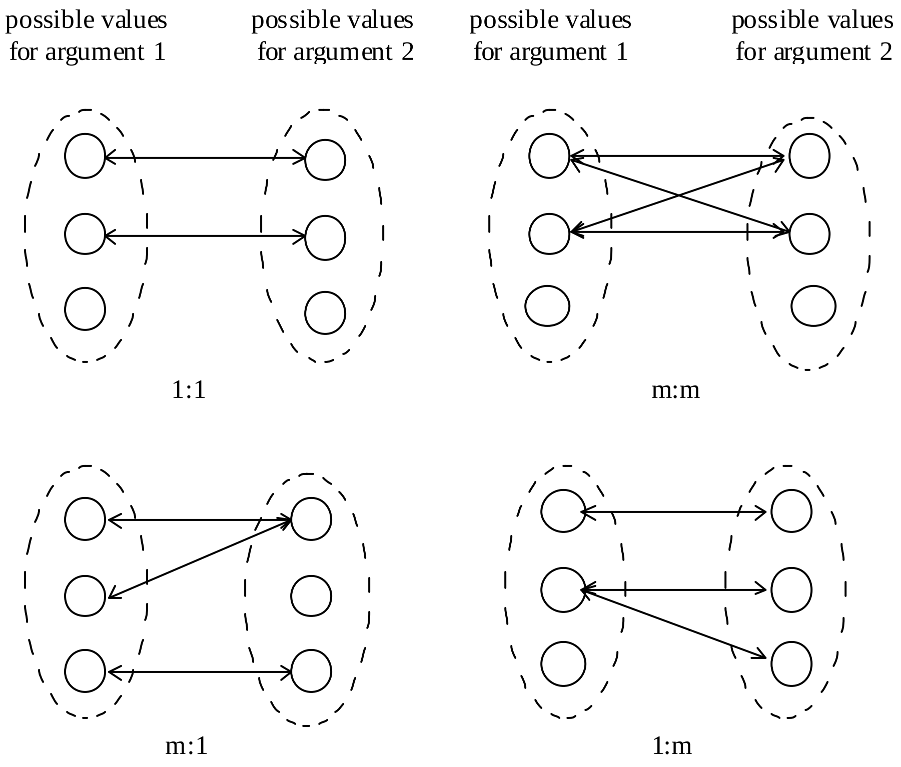

possible values for argument 1
possible values for argument 2
possible values for argument 1
possible values for argument 2
1:1
m:m
m:1
1:m

Figure 5.3: Relation type between arguments in a term with two arguments.

An example of a static constraint is a binary relation term named *answer_number* with two arguments, the first argument being a telephone user, and the second argument being the telephone number the user answers. The static constraint is that the user can have only one answer number. This is an m:1 relation, i.e. each user has only one answer number and many users can have the same answer number. For example, if it were true that *answer_number(daniel, 3990)* and *answer_number(daniel, 5555)*, that would be in conflict with the declaration. But the statement *answer_number(sandra, 3990))* and *answer_number(andreas, 3990)* is not in conflict with the declaration.

In some formal specification approaches, and often in logical systems, redundancy may be unwelcome, or even purposely avoided and eliminated. In a requirements capture task, which by nature will often be incomplete, contain errors and require revision and refinement, we should take every opportunity to collect information which is easily available and easy to capture, whether to aid the user to clarify her thoughts or for use later in verification.

5.1.4 Response Terms (Externally Visible)

Any terms visible from the environment in which the final system will operate are declared as response terms (for example ring signals, dial tones). This may be anything from an asynchronous request, to a command given to some external equipment or a message to another system. What effects the visible term will cause outside the specified system are beyond the control of the specification (with a straightforward extension of the simulator, communicating systems can be simulated, see Chapter 9). Hence, a clear understanding of the visible terms is crucial to ground the system's behaviour in its environment. In the current implementation, we provide only a facility for adding some text explaining each term's meaning (which may also contain references or formalisations).

5.1.5 Stimulus Terms (External Input)

Stimuli are the only way for the environment of a system to affect its behaviour (for example *dialling, hook_off, hook_on, recall*). A stimulus may be ignored by the system, but the most common response is an internal change of state and, eventually, an external reaction in the form of changed response terms (see transition rules). If time is an important part of a behaviour, it may be regarded as an external stimulus.

5.1.6 A State is a Set of Statements

The purpose of terms is to capture a system's current state. A state comprises a number of terms representing all statements which are true, with all other statements not stated as true assumed to be false. CABS is intended for modelling systems in which we can assume a closed world (specifications of systems to be implemented with computers mostly fall into this category, real world systems do not). The closed world assumption simplifies the logic and is the classical decision taken in many logic based languages (such as PROLOG) and knowledge based systems (such as production systems). Requirements specifications of systems implemented with computers (such as telecommunications services) mostly fall into this category (we either know that something is true or false, but do not need to reason about situations where we do not know if something is true/false).

5.2 Transition Rules

When specifying a system in CABS, the only way of causing a change is by a transition rule. If a transition rule's conditions are met, the system will change into a state where the conclusions of the transition rule are true. One of the conditions in a transition rule has to be a stimulus term. State changes can only occur in response to an external event and, thereafter, the system will wait (stay in the same state) until a new stimulus is received. This has the advantage (and for some domains, the limitation) of restricting the specification to be internally loop free. Depending on the environment of the system, it may still be possible to create external loops outside the scope of the specification (see section 5.2.1 about external loops).

Stimuli are sequenced in order to simplify the logic: we do not attempt in this high-level specification to specify what should happen when signals are competing (e.g. if two users call a third user at exactly the same time); the approach taken is that the decision of how to resolve such a situation is not necessarily a requirements choice, and can be dealt with in the design process (for many application domains, including telecommunications, assigning an arbitrary but reproducible order between competing external signals will be sufficient). Figure 5.4 shows the model used in CABS. Sequences of stimuli provided by users of telephones are used to activate appropriate transition rules. As a consequence, a sequence of states is generated, containing sets of facts that describe the system at each time a stimulus was received (*f* represents the frame axiom, which transfers unaltered facts from the previous time *t* to the current time *t+1*, see Appendix A for more details on the logic used).

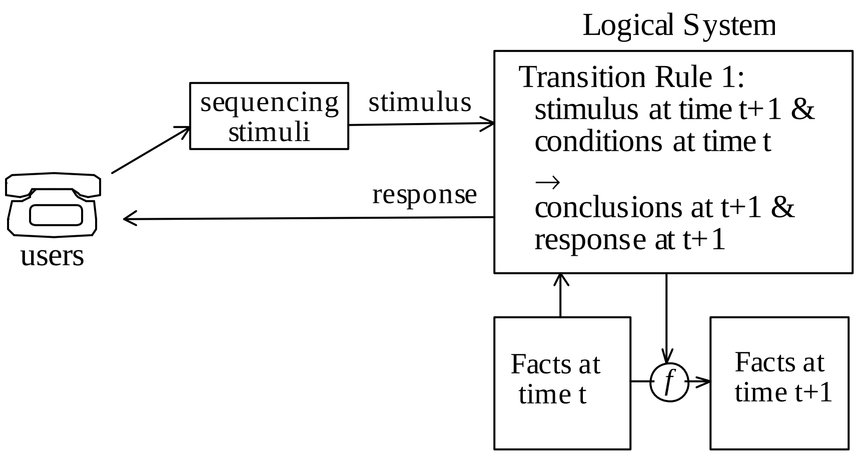
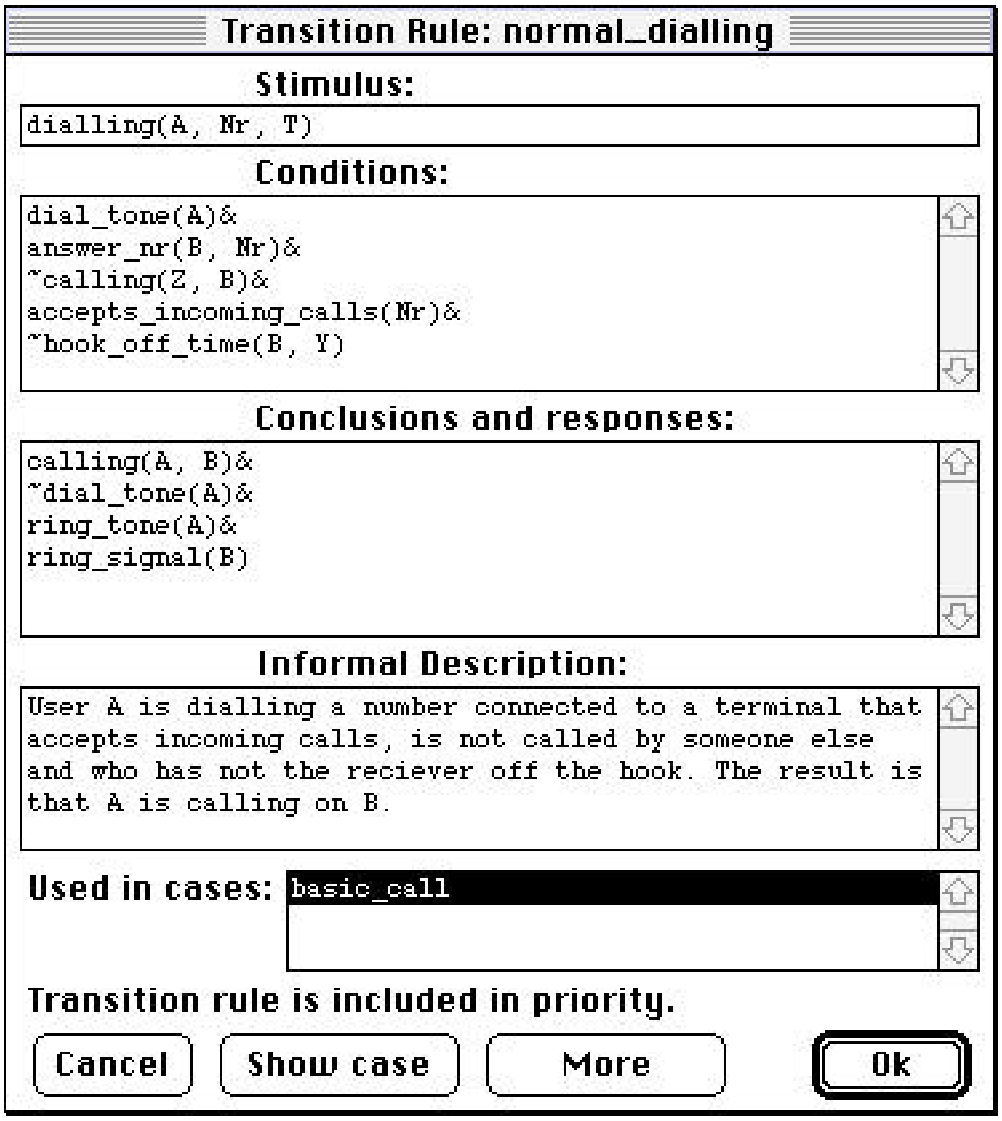


Figure 5.4: Model of the dynamic behaviour of telecommunications network

An example of a transition rule window in CABS is shown in Figure 5.5. The *Stimulus:* field shows the triggering external stimulus condition. The *Condition:* field contains a conjunction of terms defining all other conditions that have to be met. The *Conclusion and responses:* field is a conjunction of all terms that become true as a consequence of this transition rule, if the conditions are true. In the *Informal description:* field, a textual explanation of the rule, its meaning and references to relevant information are given. In the list *Used in cases:* all cases in the case library that include this transition rule are listed. The user may select a case in the list and press the *Show Case* button in order to display the case window, as in Figure 5.7.


Transition Rule: normal_dialling

Stimulus:

```
dialling(A, Nr, T)
```

Conditions:

```
dial_tone(A)&
answer_nr(B, Nr)&
~calling(Z, B)&
accepts_incoming_calls(Nr)&
~hook_off_time(B, Y)
```

Conclusions and responses:

```
calling(A, B)&
~dial_tone(A)&
ring_tone(A)&
ring_signal(B)
```

Informal Description:

User A is dialling a number connected to a terminal that accepts incoming calls, is not called by someone else and who has not the reciever off the hook. The result is that A is calling on B.

Used in cases: basic_call

Transition rule is included in priority.

Figure 5.5: Transition rule example in CABS

The *More* button gives some additional information on maintenance etc. Above the buttons, either the text *Transition rule is not part of any priority* or *Transition rule is included in priority* is shown (see Section 5.4.2 for information on priority between transition rules). Pressing the *OK* button saves the modifications and closes the window. Before saving the changes, a brief analysis of the changes is made and if in doubt, the user must confirm the changes (see Chapter 7 for circumstances under which transition rules may be modified).

5.2.1 Recursive Behaviour in Requirements

How to represent recursive behaviour, as well as the restrictions imposed on recursion by the formalism and syntax, are of major importance for requirements specifications. The main risk with recursive behaviour is that loops are specified that may be infinite under some circumstances and that this is not identified during validation and verification (this would be a major problem in any safety critical application). One advantage of recursive behaviour is that some requirements are considered easier to express with recursive behaviour. Before explaining what type of recursive behaviour is enabled in CABS, an example is given of the call diversion service (see glossary in Appendix D) in a recursive situation.

Call diversion may be used for diverting a call for more than one step. Calls to phone number 111 may have been diverted to phone number 222, and calls to 222 may be diverted to phone number 333. A careless specification of repetitive behaviour may enable specifications that exhibit unwanted behaviour, which may be difficult to validate and verify (the problem is to separate loops that always terminate and loops that under some circumstances may not terminate). If, for example, phone calls to 222 have been diverted back to 111 in the above example, some formalisms and ways of specifying the diverted call may cause an infinite loop (see example in footnote 9, page 23). A full specification (specifying all wanted behaviour and excluding all unwanted behaviour) should state what happens: an infinite loop is most likely not part of the requirements for a telecommunications systems. A requirements specification (compared with a full specification) does not include all wanted behaviour and exclude all unwanted behaviour: it merely outlines the main behaviour and leaves other parts of the behaviour open for later refinement, in order to produce a full specification (which is outside the scope of this research).

In CABS, recursive behaviour is restricted to aid validation and verification. There are two different ways to express recursive behaviour:

Expanded Recursion: If a finite recursive behaviour is to be expressed with transition rules, this can be represented with a separate transition rule for each recursive step. A recursion in *n* steps will result in *n* transition rules. Hence, we cannot create infinite loops and only one time step passes for the rule triggered (if other rules trigger in parallel, it will still be one time step, for more on this see 5.2.2). This is expressive enough for telephone services but may be awkward for some cases, especially if a user manually edits or adds transition rules capturing recursive behaviour (a more compressed syntactical notation for recursion may be introduced and automatically expanded to a set of transition rules, see Chapter 8). Both validation and verification of expanded recursion is supported in CABS (transition rules representing expanded recursion are, with respect to CABS, no different to other transition rules).

External Recursion: This mode of recursion is optional and may be forbidden if unwanted for an application domain. If a recursion is caused by a response converted externally (outside the formally specified system) to a signal, it is called an external recursion (Figure 5.6). Infinite loops can be specified in this way and are outside the control of the formal notation. The filter process may add restrictions and monitor recursion. One time step passes for each recursion. This can often be translated (manually) into expanded recursion. Even if they can be translated manually, they are different in nature to each other since in CABS, a time step will occur for every stimulus and hence each recursion will result in a time step. This may be an over-specification, especially if the requirements can be translated with expanded recursion (only one time step will pass, independent of the number of recursions). Validation of external recursion is supported by CABS, which identifies responses named *stimulus* and submits the argument as a stimulus to be simulated, see Section 5.2.1.2.

5.2.1.1 Example of Expanded Recursion

As an example, suppose we accept divert call in three steps, then we know that if there are three divert numbers (*divert(123,125) & divert(125, 139) & divert(139, 144)*) there would be three transition rules if we formalise the requirements with expanded recursion. The first transition rule would handle diversion in three steps; the second one in two steps, with the precondition that the last number does not have a divert, and the third in one step, with the condition that there is no further diversion from that number. Since there is no transition rule handling four diversions, any further diverts would be ignored by the specification (which is the aim if we limit the maximum number of diverts to three). Also, if the second divert was a divert back to 123 (*divert(139, 123)*) this would be no problem since the effect is that phone calls to the number 123 end up at 123. This is most likely a profitable situation for a service provider, since the service provider normally bills each diversion as a normal call (billed to the subscriber who activated the diversion). This would result in the subscriber for telephone number 123 paying for the call between 123 and 125, the subscriber for 125 paying for the call to 139, and the subscriber for 139 paying for the call to 123 (a triple payment for a call).

5.2.1.2 External Recursion

If recursion is specified as an external recursion, a transition rule concludes a special response, which is identified by the filtering process, and the argument is returned as a stimulus (see Figure 5.6). When behaviour is specified with external recursion, the specification exploits some known and reliable behaviour. In CABS, this special response term is named *stimulus* since its argument is one stimulus to be sent as input to the system. When the filtering process identifies a response, *stimulus(<stimulus term to be sent to system>)*, it is converted to a stimulus term and sent to the system. The external filter process is transparent for all responses directed to the users, and only identifies and filters out responses from the system that should be sent back as stimulus terms.

With this mechanism, we could specify call diversion by having a transition rule identify when a caller C calls a number N1 for which a diversion is set to number N2, and generate a response term *stimulus(dialling(C, N2, NextTime))* which the filtering process will translate to a signal *dialling(C, N2, NextTime)* and present as an input to the system.

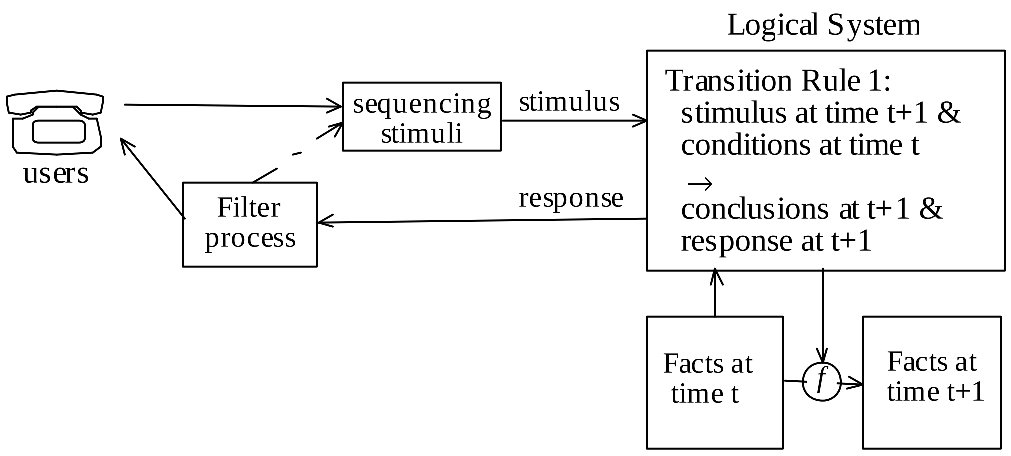


Figure 5.6: External recursion

If number *N2* has also diverted calls to number *N1*, we would end up with an infinite external loop. When *dialling(C, N2, NextTime)* is received by the system, it would correctly identify that *N2* is diverted to *N1* and generate a response term *stimulus(dialling(C, N1, NextTime))*: this may continue forever. It is theoretically difficult in general to prove that a specification containing external recursion is finite. A crude way of reducing the risk to the most obvious loops would be to add restrictions in the filter process. For example, one might only allow a stimulus with the same arguments (allowing different times) to be sent to the system a certain number of times within a given time frame. If a restriction is added that the response *stimulus(dialling(C,Nr,T))* is accepted only three times with the same arguments within one second, the loop in the example would most likely be eliminated. But since there is no way in CABS to prove that the use of external recursion will not cause an infinite loop, this way of specifying behavioural requirements should be avoided in cases where reliability is a high priority (or all external recursion should be translated to expanded recursion in the refinement process of the specification). In situations where there are good reasons for using external loops to specify some particular behavioural requirements, the part of the specification that can cause infinite loops is clearly identifiable.

5.2.2 Parallel Transition Rules and Order Independence

For requirements specification, it is useful to have transition rules that can trigger in parallel if all their conditions are met, and can also trigger independently of the order of the transition rules (transition rules are by default context independent). This can be used to separate the specifications of more or less unrelated behaviours (for example, separate call billing functionality for a telephone call from the behaviour of how to establish the call) which are triggered by the same external stimulus. Context-independent transition rules give the advantage of defining the behaviour independently of both loading order and other transition rules included in the full requirements specification (in many rule based systems, the exact behaviour of a rule can only be determined if the conflict resolution methods are known, as well as the loading order: the system may behave completely differently if the rules are loaded in a different order[17]). Transition rules which may trigger in parallel must be checked carefully to ensure that they do not have conflicting conclusions (this can be done automatically, see Section 7.6). Parallel transition rules do not affect the expressiveness of the logic and can be translated (manually) to a set of non parallel transition rules with exactly the same behaviour. Their sole purpose is to aid the separation of requirements.

5.3 Structuring Functionality in Cases

There are a number of different ways to structure functional specifications. The main aim of any structure is to make it easier for a human to understand, extend or modify a specification. It is considered to be difficult to structure large systems in predicate logic. If a structure is required for a formal specification based on predicate logic, it has to be introduced either in the formal notation itself or on a meta-level. The most common approaches are to modularise a specification or to divide the specification into a number of communicating objects (not to

[17] Since telecommunications systems requirements are composed of hundreds of different services (cases), it would be a major task to handle loading order for transition rules.

be confused with the objects in the domain of discourse, hence I will call these objects 'process objects').

In the chosen telecommunications domain, the functionality is divided into functional parts[18] called services, where each service reflects some particular behaviour of the final telecommunications system. Services are often classified as either basic services, capturing some main functionality, or as services which add functionality to these basic services. In telephony, the basic functionality is to establish a voice or data connection between two users. Examples of services adding functionality are *call return*, *call minder* and *call waiting* (for more examples see "Selected services, User guide, BT" and Appendix B). The basic service in telecommunications is decreasing as part of the total functionality and the overall functionality is getting more complicated. In our example domain of telecommunications services, we implemented each service as a separate case, which follows the traditional way of structuring telecommunications services.

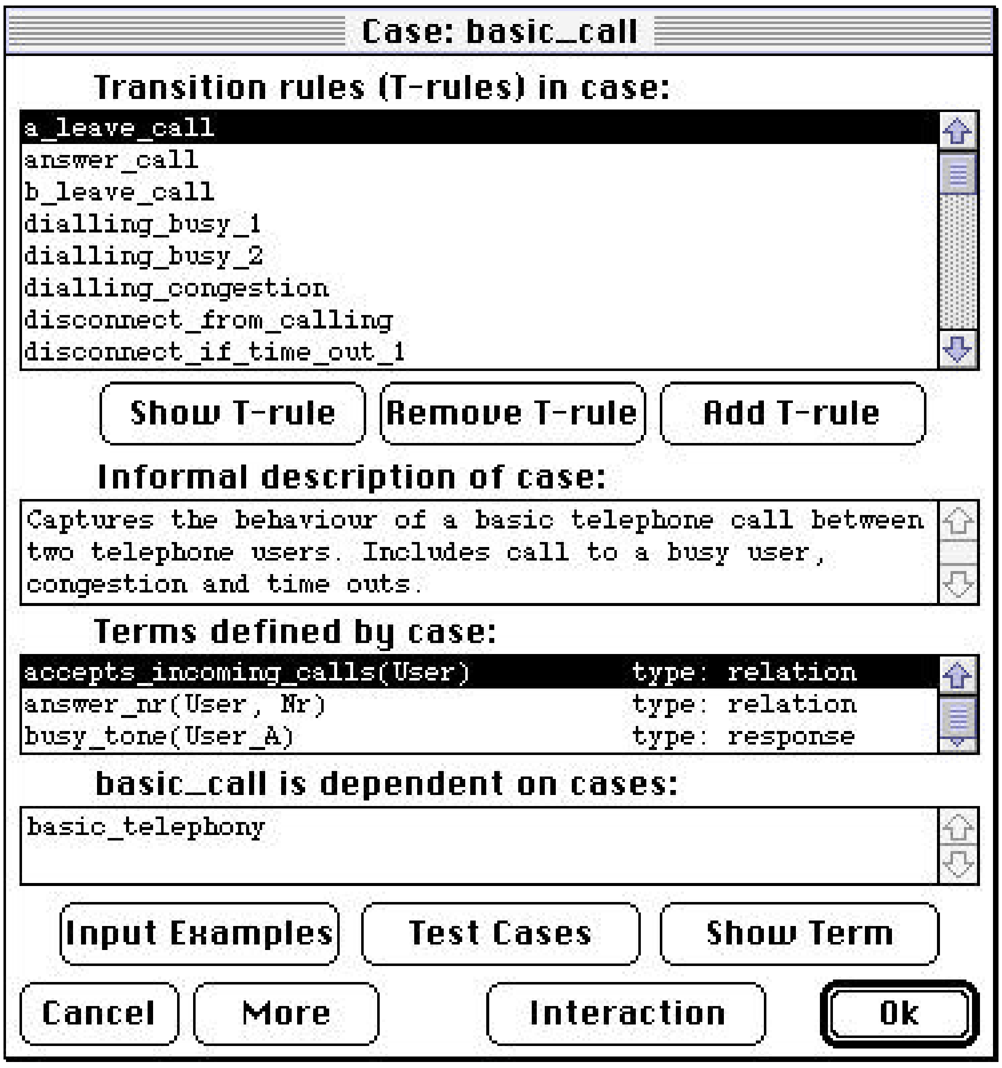
Figure 5.7 shows an example of how a case is displayed in the CABS system. In the scroll list under the text *Transition rules (T-rules) in case:* a list with all transition rules belonging to the case is shown. The user may chose to inspect a particular transition rule by selecting the appropriate button. This will show the window for the transition rule as shown in Figure 5.5. In the same way, a transition rule can be removed from or added to the case. The informal description gives a brief description of the case, its purpose, functionality and links to relevant documentation. In the list *Terms defined by case:* a list with all terms defined by the

[18] In telecommunications, it is also common to have an object oriented structure at the design and implementation level (but not at the requirements level) where different parts are handled and implemented separately (trunk lines, protocols, regional processors, access points, etc.). In contrast, functional requirements specifications should ideally be as independent as possible of design and implementation decisions.

case is shown. The button *Show Term* will show the selected term in the list. This information is used to determine relationships between cases.

If a case specifies behaviour added to that of a previously specified case, in the sense that a system does not include the base case, the extension case does not make sense on its own (in telecommunications, three party call may be specified as an extension to *call waiting*). If a case specifies some behaviour added to a previously specified case, this is shown in the list *<case name> is dependent on cases:* in the window.

The button *Interaction* makes an in-depth analysis of relations and dependencies between cases (some of the interaction can be determined automatically in CABS, see Chapter 9 for more details). The user can choose to inspect the input examples on which the case has been based on by pressing the *Input Example*s button, or can choose to inspect the test cases used for the verification of the case by pressing the *Test Cases* button (if no *Test Cases* have been generated automatically from the case, this can be initialised). The *More* button gives some additional information, such as when a case was last modified.



```
a_leave_call
answer_call
b_leave_call
dialling_busy_1
dialling_busy_2
dialling_congestion
disconnect_from_calling
disconnect_if_time_out_1
```

Captures the behaviour of a basic telephone call between two telephone users. Includes call to a busy user, congestion and time outs.

```
accepts_incoming_calls(User)      type: relation
answer_nr(User, Nr)               type: relation
busy_tone(User_A)                 type: response
```

```
basic_telephony
```

Figure 5.7: The case window in CABS

In a requirements specification, it should be obvious which parts of the specifications are requirements and which are merely there to aid the human user in handling a large specification. To represent both the specification and these additional structures in logic may complicate the logic used to represent the specification and it may be difficult to extract the part of the specific ation relating purely to requirements. With an object oriented approach, the distinction between specification and supporting structure may be difficult to make, since dividing a functionality into a number of communicating objects may be a requirement or just a decision taken in order to make the specification easier to understand. If a large system with varied functionality is divided into large numbers of communicating objects, this may require extensive communication and large numbers of communication protocols to understand and handle. If so, this may reduce the benefits from structuring the requirements into objects, or, in the worst case, lead to a specification which is more complicated than if specified without a communicating object structure.

In CABS, a case consists of a set of transition rules. Cases also contain references and information aiding human understanding, re-use, modification and evaluation. The logical formalism does not know what a case is and treats all transition rules as one large set of axioms. The main reasons for this design are:

CABS is aimed at people who are not skilled in logic, so it is important to keep the logic as clear and simple as possible.

To avoid complications in the verification and simulation of specifications.

To keep the distinction clear between what parts of the requirements are requirements and what parts are an aid to human thought processes.

One of CABS' aims is to stretch a simple, executable logic as far as possible and to explore the benefits and drawbacks of this minimalistic approach in a real application domain.

As mentioned earlier, a case may be specified as being dependent on another case. The opposite would be that a case is independent of all other cases and doesn't interact with any other cases (not common in the telecommunications domain). If such an approach can be taken for an application domain, each case may be viewed as a single process that can be specified, re-used, validated and verified in CABS. If a domain contains individual autonomous components exhibiting external communication only, there is no need to consider interaction and dependencies. Such a domain would be well suited for CABS (or, even better, a simplified version of CABS where all parts especially included to handle interaction and dependencies are excluded). One current limitation in CABS is that if the overall behaviour of the system is determined by a set of communicating cases (communicating with each other by external stimuli), this may be simulated, but not formally verified in CABS (beyond the scope of this research).

5.3.1 Case Relations

A telecommunications service may be dependent on other services (adding functionality to them) or interact with another service, i.e. there is a new behaviour when both services are in the same system. For this reason, requirements have to be carefully validated and verified to determine where and how cases affect each other and the overall behaviour.

Cases being dependent on each other is a common feature of a system that is structured according to functionality. A case X may add functionality to case Y, hence case X is meaningless if case Y is not included in the constructed system. This information has to be captured during the initial specification. Also, analysis of where terms are used, and in what way (as a condition, conclusion, negated, ...), may identify dependencies and relations between cases, since terms are specified as belonging to a case. How a term is used is important during analysis. For example, if a term is used in the condition part of a transition rule, the rule can never be triggered if no other transition rule has the term in its conclusion part. Some cross-reference tools have been developed in order to analyse the transition rules and their use of terms (these tools have not been integrated in the current version of CABS).

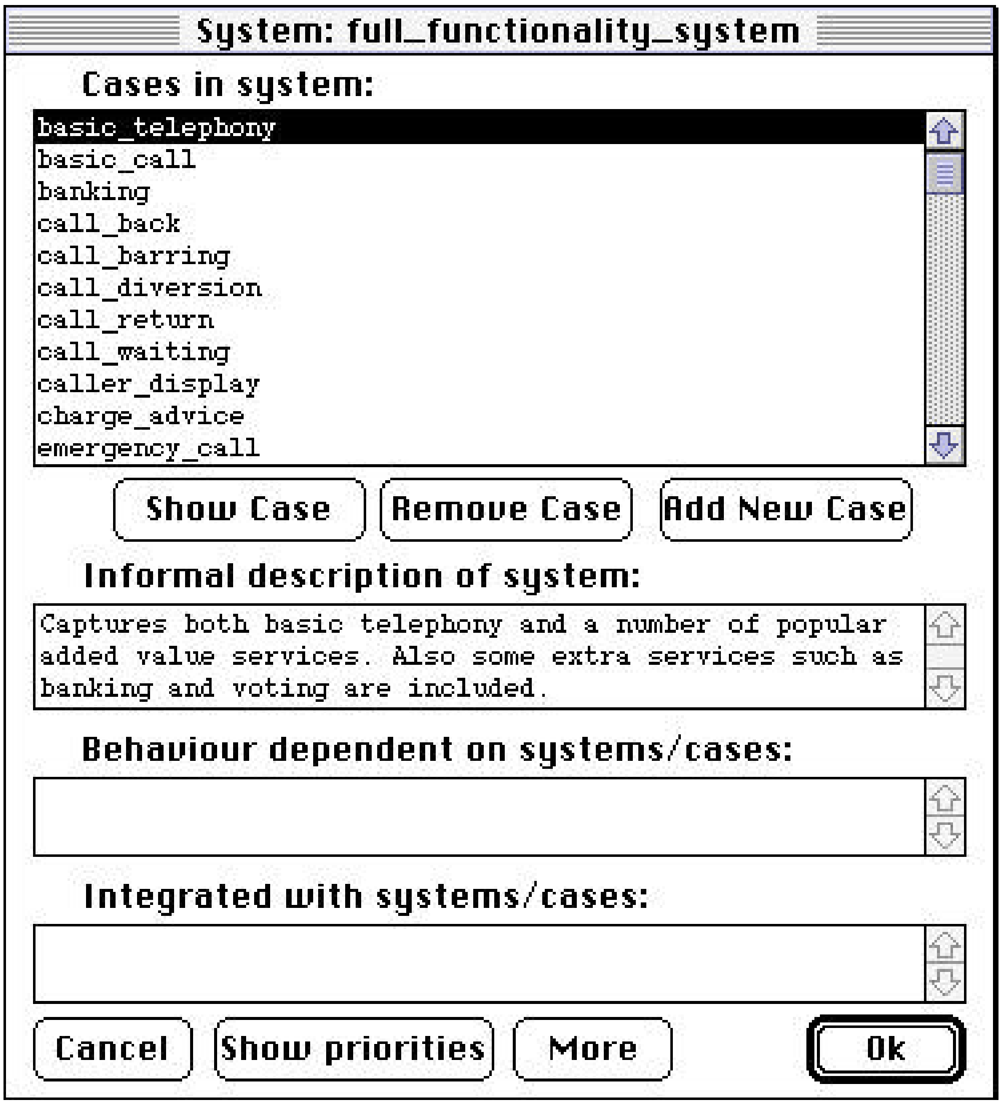
5.4 System Requirements (Sets of Cases)

The requirements specification of a system specified in CABS is a set of cases whose behaviour (including the most common interaction between them) has been validated and verified. Systems requirements may include additional input examples, exemplifying interaction between different cases in the system. In the telecommunications domain, system requirements may denote all telephone services supplied to a particular country, service vendor, local or global company. Interactions between systems may also have input examples exemplifying certain interactions. When a case has been modified or a new case is added to a system, all input examples describing interaction with other cases should be verified again.

Also, the system that includes these modified or new cases should have all its interaction with other systems verified. In CABS, automated verification of sets of test cases is implemented, assuming that we can select which system or systems to verify, and select which input examples to verify.

In Figure 5.8, an example of the system window in CABS is shown. First, a list of all cases included in the system is shown. The user may inspect, remove, replace or add cases to a system. An informal description is given as a textual description of the system, with links to relevant material. The list *Behaviour dependent on systems/cases:* contains the names of systems and cases on which the system is dependent to specify a meaningful behaviour. If the list is empty, then the system specifies a meaningful behaviour on its own. If not, then in telephony it is most likely a set of add-on services (specially designed services adding functionality for which phone users are prepared to pay extra, which in turn increase income for telecommunications service providers). If there are cases in the list, then the system is dependent on any system including these cases. In telecommunications, there may be a large number of different systems where only a few cases differ for each system, and so it is preferable if an add-on system is dependent only on the parts of the system that are the same for all these different systems. This increases the possibility of re-using the system and facilitates adaptation and integration.

The list *Integrated with systems/cases:* is a list of systems or cases with which the particular system has been verified and validated. In telecommunications applications, it is important to keep track of these, since there are a large number of different systems designed for specific categories of users, vendors, service providers, etc. If it is a case in the list, then the same reasoning applies as for the *Behaviour dependent on systems/cases:* situation. Also, when validating and verifying a particular service, CABS needs to know in which context the service is to be tested (a set of cases/systems).


System: full_functionality_system
Cases in system:
Show Case
Remove Case
Add New Case
Informal description of system:
Behaviour dependent on systems/cases:
Integrated with systems/cases:
Cancel
Show priorities
More
Ok

basic_telephony
basic_call
banking
call_back
call_barring
call_diversion
call_return
call_waiting
caller_display
charge_advice
emergency_call

Captures both basic telephony and a number of popular added value services. Also some extra services such as banking and voting are included.

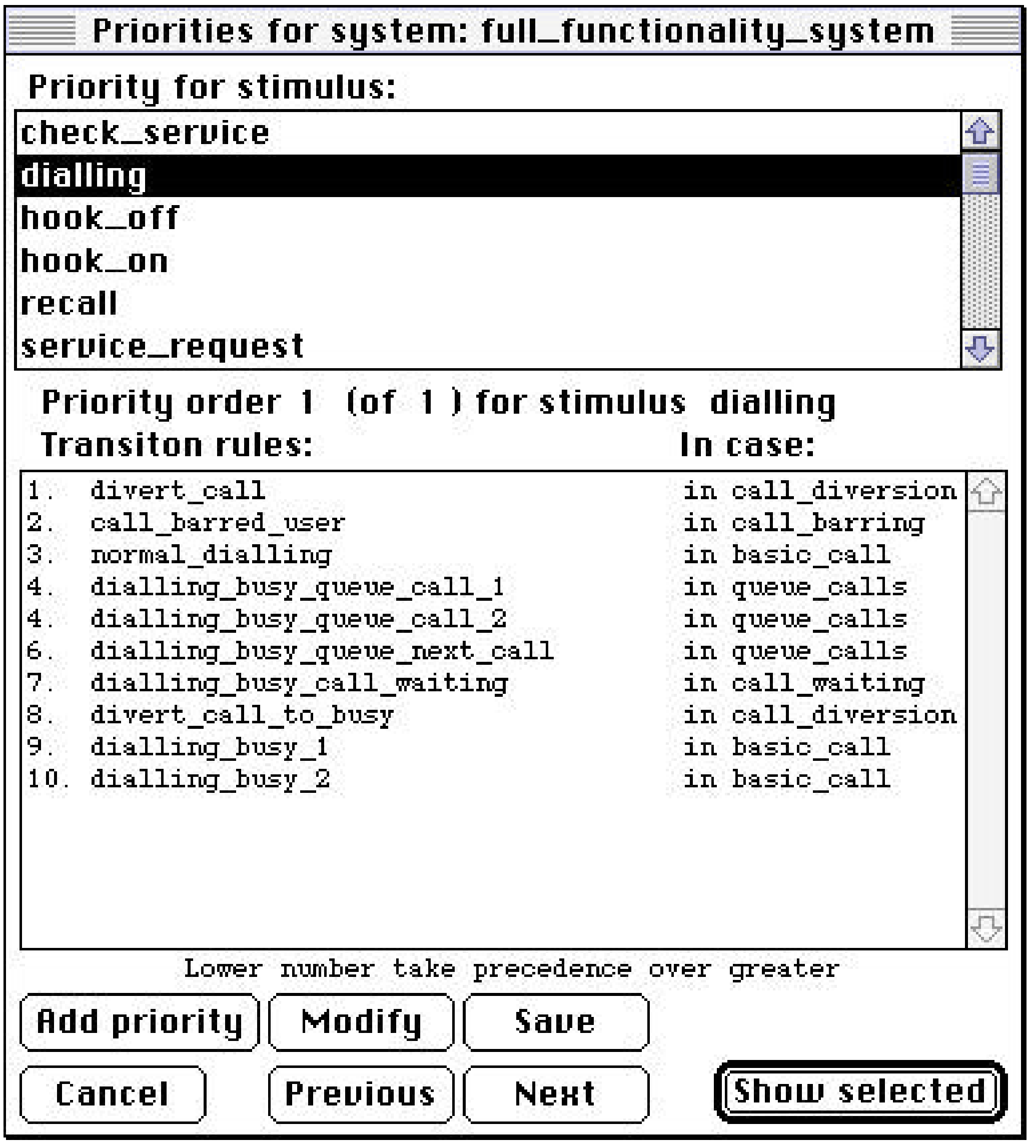
Figure 5.8: System window in CABS

5.4.1 Different Application Domains

A case library normally contains cases from just one application area, since different parts of the case library can have only one unique meaning. If a case library captures different, but related, application domains, where terms may have different meanings, great care has to be taken to ensure that any reasoning and re-use is not based on terms from the different application domains having similar but not equivalent meanings. A requirements capture process, whether formal or informal, has the main purpose of outlining the requirements as closely as possible, and if this reasoning process is based on terms not clearly defined, or even having different meanings, it would complicate matters considerably.

5.4.2 Priority for Transition Rules in Systems

In some situations, it may be desirable to have context dependent rules on a local level. Since every transition rule has a unique name, we can define local orders between transition rules, i.e. if transition rule *divert_call* triggers (has all its conditions met) then *normal_dialling* should not trigger. Such a request can be specified with transition rules by including all conditions from *divert_call* as a negated conjunction in the transition rule *normal_dialling*. If there are more than two or three transition rules that are exclusive, or overriding each other, this solution is somewhat tedious as the conditions will get very large. Therefore, we allow the user to define explicitly a local order between a number of named transition rules (see Appendix A for more details on logic). Figure 5.9 demonstrates setting the priorities for transition rules triggered by stimulus *dialling*. To inspect or modify a priority, the user first selects the stimulus to which the priority applies (by selecting the stimulus in the list *Priority for stimulus*). The current order shown is the number after *Priority order* followed by the total number of priority orders for this stimulus in the brackets. In the next list, the name of the transition rules (with the name of the case in which they are defined) and their local priorities are displayed. For example, transition rule *1. divert_call* will override *9. dialling_busy*. If *divert_call* has its condition met, all the following transition rules in the list cannot trigger. The same transition rule may occur in different orders which enables the user to specify a lattice. If transition rules are exclusive (they cannot have their conditions met in the same state), they may be given the same priority numbers (as is the case for transition rule *dialling_busy_queue_call_1* and *dialling_busy_queue_call_2*). Protection against circular priorities should be provided when new priorities are added or existing priorities are changed (not implemented).


Priorities for system: full_functionality_system

Priority for stimulus:

check_service
dialling
hook_off
hook_on
recall
service_request

Priority order 1 (of 1) for stimulus dialling

Transiton rules:		In case:
1.	divert_call	in call_diversion
2.	call_barred_user	in call_barring
3.	normal_dialling	in basic_call
4.	dialling_busy_queue_call_1	in queue_calls
4.	dialling_busy_queue_call_2	in queue_calls
6.	dialling_busy_queue_next_call	in queue_calls
7.	dialling_busy_call_waiting	in call_waiting
8.	divert_call_to_busy	in call_diversion
9.	dialling_busy_1	in basic_call
10.	dialling_busy_2	in basic_call

Lower number take precedence over greater

Figure 5.9: Priority window in CABS

The explicit local order is purely syntactical and, from a logical point of view, the priority is expanded into negations in the transition rules (explained in Appendix A). This local order allows us to make the meaning of the transition rules independent of the order in which they are loaded, as discussed in Section 5.2.2.

5.5 Graphical Input Examples

All previous graphical input examples on which a specification is built are stored in the case library, including both their graphical layout (created by the user) and the detailed requirements added to them under refinement. Since the graphical input examples are the original source on which the formalised requirements are based, we have to keep them for further modifications and extensions of the system. In the CABS system, the user can create new and re-open previously created input examples, and modify and save them in their graphical form. All information is stored in the case library.

5.6 Storing and Re-using Test Cases

Test cases are generated from input examples and in some cases, revised or added by a user (user initiated simulations may be stored as test cases; some parts of this are implemented in CABS). All the test cases are needed in order to verify a modified system. If changes have been made to some parts of the system, all test cases that can be theoretically affected by the change have to be re-tested in order to verify that the required behaviour is still captured by the requirements specifications.

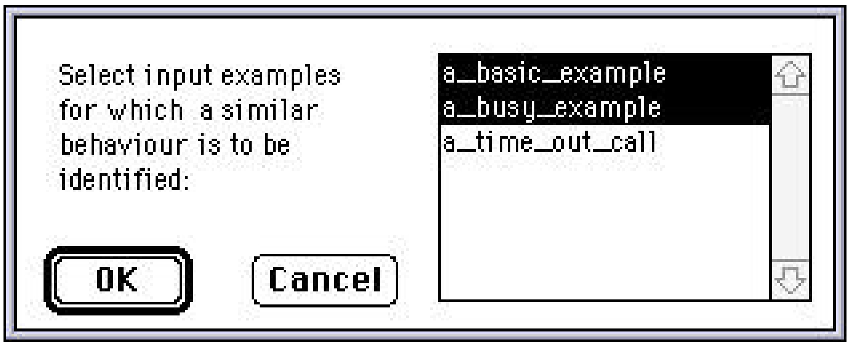
We also need to maintain the link to the input examples from which the test cases originally stem. This gives us the ability to identify which test cases are still valid or have to be removed due to changes in the input examples on which they are based. How test cases are used in the validation and verification task is explained in Section 7.5 and Section 7.6.

Chapter:

6. Matching and Identification of Similar Behaviour

The purpose of the matching process is to identify cases, or parts of cases, hold in the case library which have similar behaviour (as exemplified by the input examples) and which may be considered for re-use. A computationally fast and uncomplicated matching algorithm aimed at identifying similar behaviour is used in CABS. The result of the matching must be narrow enough to identify candidates for re-use and broad enough not to exclude relevant cases. The final selection will be carried out by the user, validating and verifying the selected match with the tools provided in CABS. If the user is not satisfied with the result of the matching, she may redo the match after refining the input examples or modifying parameters, thus directing the matching process in order to identify more suitable candidates.

When a user of CABS wishes to make a match, she selects *'Match…'* from the CABS pull down menu. A dialogue window (Figure 6.1) with all the input examples on which the match may be based is shown. The user selects the input examples to be used in the match (*a_basic_example* and *a_busy_example* have been selected in Figure 6.1). When the *OK* button is pressed, the system will try to identify cases in the case library that capture the same or similar behaviour. The result is shown in Figure 6.9.


Select input examples for which a similar behaviour is to be identified:

a_basic_example
a_busy_example
a_time_out_call

Figure 6.1: Selecting input examples to match.

CABS implements a two-step matching process based on comparing sets which results in a fast and fairly easy to understand matching algorithm. First, transition rules capturing the same or similar behaviour (as exemplified in the detailed links from the input example) are identified, and then cases capturing similar behaviour exemplified in the input examples are identified. Both individual transition rules and whole cases may be re-used to create a new requirements specification capturing the exemplified behaviour. In Figure 6.2, the matching algorithm is outlined in pseudo-code.

```
For all links from the input examples, $L_n$:
    For all transition rules in the case library, $T_m$:
        Analyse the different features indicating closeness of
        behaviour for $L_n$ and $T_m$,
        Calculate the score for the behavioural closeness between
        $L_n$ and $T_m$ (calculation based on the features and
        parameters set by user).
For all cases in the case library, $C_i$:
    Calculate an overall score for $C_i$ based on the closeness scores
    of the transition rules in $C_i$.
Sort transition rules and cases according to their overall
score for closeness of behaviour.
```

Figure 6.2: Outline of matching algorithm

Requirements specification, as well as re-use of requirements specification, is seen as an iterative process: parts of the result of the matching can be confirmed by the user before a partial re-match is carried out, possibly with a different set of matching parameters.

Any matching algorithm able to identify cases with the same or similar behaviour to the input examples may be considered for the task. The matching may be semantic or syntactic. Syntactic matching may be a straightforward keyword based matching or a more elaborate one, using knowledge about the structure in order to improve the matching result. A syntactic matching which is sufficiently fast and accurate for the task of identifying similar behaviour has been chosen for CABS. The matching algorithm used is based on set intersections and unions.

For some application domains, a computationally faster choice would be a pure keyword based search, identifying terms occurring in both the detailed links and the transition rules from the case library. A keyword based search produces good results when there are one or more unique keywords (terms) that may be identified in the input examples, or by the user, in order to determine relevant cases and parts of cases. This is true for some of the services specified in CABS in the telecommunications domain (for example, *redirect calls*, which defines and uses the term *redirect*). Many services in the application domain of telecommunications do not have easily identifiable unique terms like *redirect calls* does (*pick up call* and *voting* are examples of services not having any terms defined and if there are variants of a service in the case library, they will all have the same terms defined), so keyword matching cannot be used as the only method of identifying cases. Also, similar services or variants of the same service do not, in most cases, have discriminating terms, making keyword matching less accurate. If no unique terms are present in the set of terms, and many cases use the same set of terms, too many matching cases may be identified as possible candidates. Since telecommunications services requirements are based on a fairly small set of different terms used by most services (terms such as *answer_number*, *calling*, *ring_signal*, *busy_tone*, *in_speech*), straight keyword matching is unlikely to produce reliable results in this domain. Keyword based matching could complement the algorithm used in CABS, since keyword matching is even faster, and if there are some specific terms related directly to the behaviour exemplified in the input, the relevant cases can be identified. However, keyword matching is not implemented in the current system. The matching used in CABS has the advantage of capturing features, thus allowing the user to make some semantic assumptions about a match that may be useful in the selection process or when modifying matching parameters. For more on optimising matching and different methods on how to prune a search see for example [Althoff, Auriol, Barletta, Manago 95].

In this chapter, we first explore the terms what “similar behaviour” and “closeness of behaviour” mean, and establish how to identify and score transition rules capturing behaviour which is similar to the detailed links. After that, the process of identifying similar cases is described (this process is based on the identified transition rules capturing a similar behaviour to the links).

6.1 Defining Similar Behaviour

One of the main issues in case based reasoning systems is the choice of appropriate features for cases. A case in the case library is only of use if there is a way of identifying when the case can be re-used in whole or in part. If indexes are badly selected, it will require great effort or even be impossible to locate relevant cases. If the indexing vocabulary [Kolodner 93] is well chosen, it will be easy to compare stored cases to the given task, and to determine if a case is of interest or not. Hence we need to investigate both the application domain and the semantics of cases, and to carefully select features to be used in the matching process. The features used should be fairly easy to understand and to explain to the user, which will aid in the task of adapting matching parameters to a particular application domain. The algorithm implementing these features should also be computationally fast enough to produce a result within an acceptable time.

Before we define the features (see section 6.4) used in the matching algorithm, a number of expressions are defined. These are used as the basis for feature definitions, which make the assumptions and compromises necessary to produce acceptable results and achieve a computationally efficient implementation of the matching algorithm.

In our application domain, it is always possible to determine if a link[19] from the input examples and a transition rule from the case library capture exactly the same behaviour. If a transition rule and a link have exactly the same behaviour, they must have the same conditions (stimulus and other conditions) and conclusions (responses and other conclusions). It will therefore be obvious that all behaviour included in the link is included in the transition rule, and all behaviour excluded by the link is excluded by the transition rule. In the following

[19] If we use ‘link’ without a discriminator, we mean a detailed link (the expanded graphical link with extended conditions and conclusions). The term ‘graphical link’ will be used to refer to a graphical link from the input example.

definitions, we will treat the links as transition rules, since they are so similar syntactically that there is no need for a distinction in the definitions. When translating the definitions into features, the difference is of importance and will be reintroduced, since the features capture some of the semantic aspects of the differences between links and transition rules.

Definition 0, exactly the same behaviour: Two transition rules exhibit *exactly the same behaviour* if and only if all conditions (stimuli and other conditions) and conclusions (responses and other conclusions) in the transition rules are equal.

If there is more than one link in the input examples which has the same behaviour as a particular transition rule, the relevance of this transition rule may be more significant (for further details on combined links, see Section 6.5). The notation of capturing *exactly the same behaviour* is not sufficient in the telecommunications domain since it is very unlikely that a link and transition rule have exactly the same conditions and conclusions. The reasons for this are that a behavioural input example represents a particular example of the behaviour, but a transition rule captures many cases, and also includes interaction with other telecommunications services. This usually results in links having fewer conditions and conclusions than transition rules. For this reason, we need a more fine grained vocabulary to be able to reason about closeness of behaviour.

Definition 1, same external triggering condition: Two transition rules have the *same external triggering condition* if and only if their stimulus term conditions are equal.

It may be useful to know whether there is a contradiction between a transition rule and a link, i.e. if they cannot apply to the same states and hence not capture the same behaviour. This is done in definition 2.

Definition 2, under no circumstances capture the same behaviour: Two transition rules can *under no circumstances capture the same behaviour* if there is a contradiction between their condition parts or their conclusion parts or both.

It may also be useful to know whether a link and transition rule apply to the same state.

Definition 3, same originating state: Two transition rules have the *same originating state* if and only if all their conditions are equal (stimulus conditions do not need to be equal).

If definition 3 is not met, it may be useful to know if there is any state in which the link and transition rule have their conditions met. We do not distinguish between a reachable state and a possible state. The difference between this and definition 3' is that even if there is a state (a set of terms) under which both transition rules may have their conditions met, there may be no possible sequence of stimuli that can bring the system into this state. Such an analysis may be used as an additional source of information when determining how similar two transition rules are, but may be computationally expensive for large requirements.

Definition 3', some originating states in common: A transition rule, T_1 has *some originating states in common* with another transition rule T_2 if the conditions of T_1 are a subset of T_2's conditions and there is no contradiction between T_1 and T_2's disjunction.

The relationship between the terminating states may also be of interest:

Definition 4, cause the same effect: Two transition rules *cause the same effect* if their conclusions are equal and they have some originating states in common.

A weak form of definition 4 looks at the question of whether there is any state in which both the link and the transition rule have their conclusions met.

Definition 4', some terminating states in common: Two transition rules, T_1 and T_2, have some *terminating states in common* if T_1's conclusions are a subset of T_2's conclusions and they have some originating states in common.

In the application domain of telecommunications services, the external visible side effects (response terms) may have a higher significance than other conclusions, hence we introduce separate definitions (definitions 5 and 5') for externally visible side effects (responses).

Definition 5, same externally visible effects: Two transition rules have the *same externally visible effects* if and only if the response terms in their conclusions are equal and they have some originating states in common.

Definition 5', some externally visible effects in common: Two transition rules, T_1 and T_2, have *some externally visible effects in common* if T_1's response terms is a subset of T_2's response terms and they have some originating states in common

Because of the fact that links are expected to be part of some particular input example, it is unlikely that there are input examples and transition rules meeting the definitions fully, hence we need to define a set of matching features based on the definitions, which allow for some flexibility. Features should be defined in such a way that their subsequent use is computationally efficient. The result should also aid us in determining the closeness of behaviour between an input example and a set of transition rules from the case library. These definitions have been selected since they can easily be translated into features which can all be determined fairly accurately at a low computational cost, using the structure inside transition rules and comparing sets of terms.

In the next sections, we will explore how these definitions are used to define features which are useful in the evaluation of behavioural closeness. We will then look at how these features can be translated into values, and how these values are then combined into a single value, which gives a sufficiently accurate estimate of the closeness of the behaviour between links and transition rules, or input examples and cases respectively.

6.2 Using Parts and Sets to Analyse Similarity

Before exploring the connection between the definitions, features for estimating closeness and structural matches between transition rules and links, the syntactic structure used for comparison is detailed. The transition rules and the links are each partitioned into seven parts:

Transition rule: Stimulus part (extracted from condition part)

Condition part (stimulus and negative conditions excluded)

Negative condition part (stimulus and non negative conditions excluded)

Conclusion part (response parts and negative conclusions excluded)

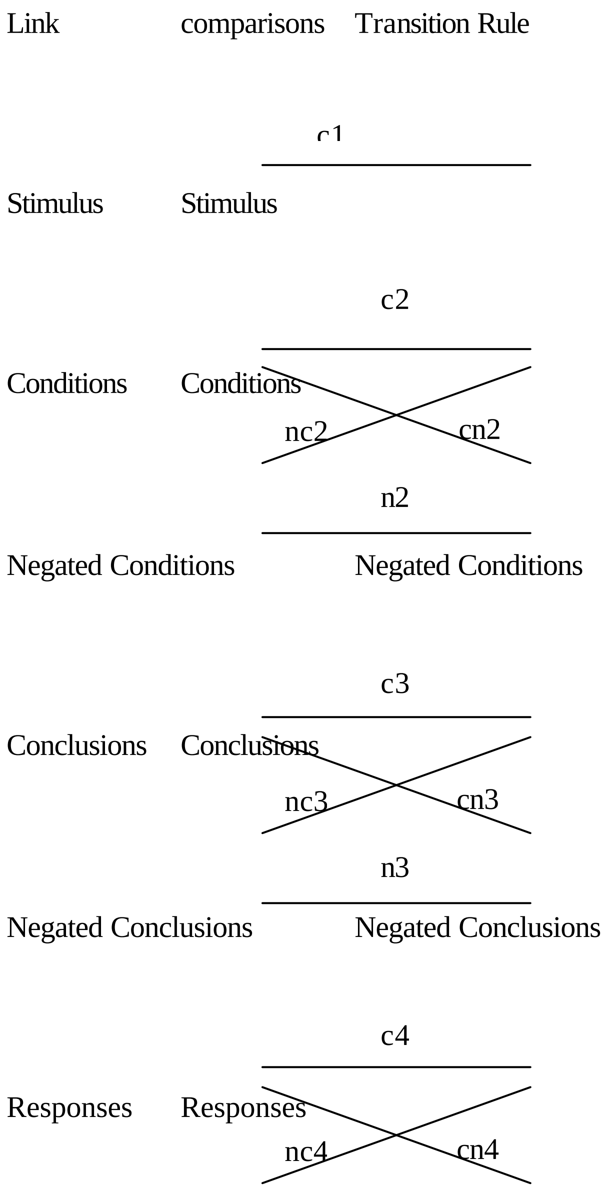
Negative conclusion part (response parts and non negative conclusions excluded)

Response part (extracted from conclusion part)

Negative response part (extracted from conclusion part)

An analysis of arguments for terms is not made at this stage of the matching. Sufficient assumptions can be made which exclude a large number of transition rules from further analysis and rate the remaining matches without an in-depth analysis of arguments and variable bindings (a variable refers to a specific entity in the application domain, such as a specific phone number or subscriber without naming the entity). The exclusion is made conservatively, since care must be taken not to exclude transition rules that may be good candidates. Each part is treated as a set with zero or more terms. This can be done safely because the condition, conclusion and response parts are all restricted to conjunctions of terms. With current restrictions on expressions, disjunctive terms (where no brackets are allowed, and conjunction has priority over disjunction), may be allowed to occur in a transition rule, and any disjunctions which occur can be expanded to a set of transition rules containing only conjunctive terms.

The partitioning of transition rules is trivial since terms are typed as stimulus, response, attribute or relation before they are used in links or transition rules. The stimulus part is restricted to only one non-negated term of the type stimulus, and the stimulus terms are only allowed to be used in the stimulus part. The partitioning of terms gives us a basis for comparison and for drawing some conclusions to be used in the closeness of behaviour rating. Negated terms in parts are handled separately, so seven features may be compared for each link/ transition rule pair, and six cross comparisons (negated/ non negated parts, see line nc2, cn2, nc3, cn3, nc4, cn4 in Figure 6.3) may be made. Selected comparisons are used for defining features. They are translated into numerical form and used to create an overall score, which in turn is used in the final rating of the "closeness" between the transition rule and link. These comparisons have been chosen because they are computationally fast to determine, fairly easy to understand and the fact that they can be used to indicate if a link and a transition rule capture similar behaviour. The choice of which of these comparisons to use as features and their connection to the definitions are explored in the following sections.



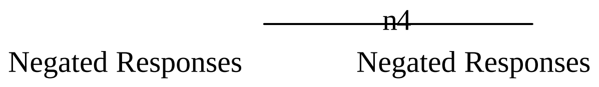


Figure 6.3: Possible comparisons between parts in link and transition rule

For reasons of computational cost, we do not calculate every comparison for every pair of link/ transition rules, since, if some comparisons are below a threshold set by the user, the transition rule is classified as uninteresting and no further evaluation on the transition rule will be made. These thresholds set by the user should ensure that no relevant matches are excluded but, if in doubt, the threshold values can always be set to zero and all matches will be included whatever the score is. This may take a considerable time for a large case library, and it is up to the user or system manager to weigh up the advantages of a faster match against the risk of missing possible matches (see section 6.5.1). Since the comparison is set based without any computationally expensive calculations, it is computationally fast and only marginally slower than keyword matching since the comparisons all are implemented as a number of keyword matches (each term in the link/transition rule is used as a keyword for the corresponding set). Hence, a linear relationship, depending on the number of terms in the link and the transition rule, determines the upper limit of the computational cost. In telecommunications specification, the number of terms in transition rules are expected to be below 35 (in our case library no transition rule has more than 30 terms). In links from input examples, even fewer terms are expected.

6.3 Translating Comparisons to Values

Before defining the features used to estimate how similar the behaviours of a case and input examples are (Section 6.4), we will describe how to calculate the values used in these features. It is not necessary to understand this section in detail to be able to understand the feature definitions. A comparison (all possible comparisons are shown in Figure 6.3) between a part from a link and a part from a transition rule is first translated into an integer triple, where the first number is the number of terms in the link, the second is the number of terms in the intersection and the third is the number of terms in the transition rule from the case library. These triples are then used to calculate two coverage percentage values used for calculating the features.

For each comparison, two values called the intersection coverage percentage are calculated. The intersection coverage percentage values are called ICL (Intersection Coverage of Link) and ICT (Intersection Coverage of Transition rule). The terms in the part of the link and the transition rule under consideration are both regarded as two sets (L and T respectively) and the intersection $L \cap T$ is a set called I. The value for ICT = 100 * number(I) / number(T) and ICL = 100 * number(I) / number(L). The value is given as a percentage value between 0 and 100, appropriately rounded since decimals would not make any significant difference. If $L=\emptyset$ or $T=\emptyset$ (a rare situation in our application domain) then ICL (respectively ICT) is set to zero.

In Figure 6.4, the five main situations for coverage are shown. In the first case (top left example in Figure 6.4) the sets L and T are equal, hence the intersection, I, is also equal to L and T $((I=L \cap T) \wedge (L=T)) \Rightarrow I=L=T$). The intersection covers 100% of the terms in the link, hence ICL = 100. The intersection fully covers the terms in the transition rule, hence ICT = 100 in this case.

If there are 3 terms in T and 2 terms in L and $L \subset T$, the intersection I = L and contains 2 terms. The intersection has 2/3 of the terms in T giving an ICT value of 67 (67 %) and an ICL value of 100. This corresponds to the top right example in Figure 6.4.

If there are 2 terms in L and 3 terms in T and the intersection I contains 1 term, then ICL is 100*1/2 = 50 and ICT = 100*1/3 = 33. This example corresponds to the middle left example in Figure 6.4.

The middle right example corresponds to the top right example (L and T have their positions switched, $T \subset L$). The bottom example illustrates when the intersection I between the two sets is empty $(L \cap T) = \emptyset$. Both ICT and ICL are assigned the value 0 for the last situation.

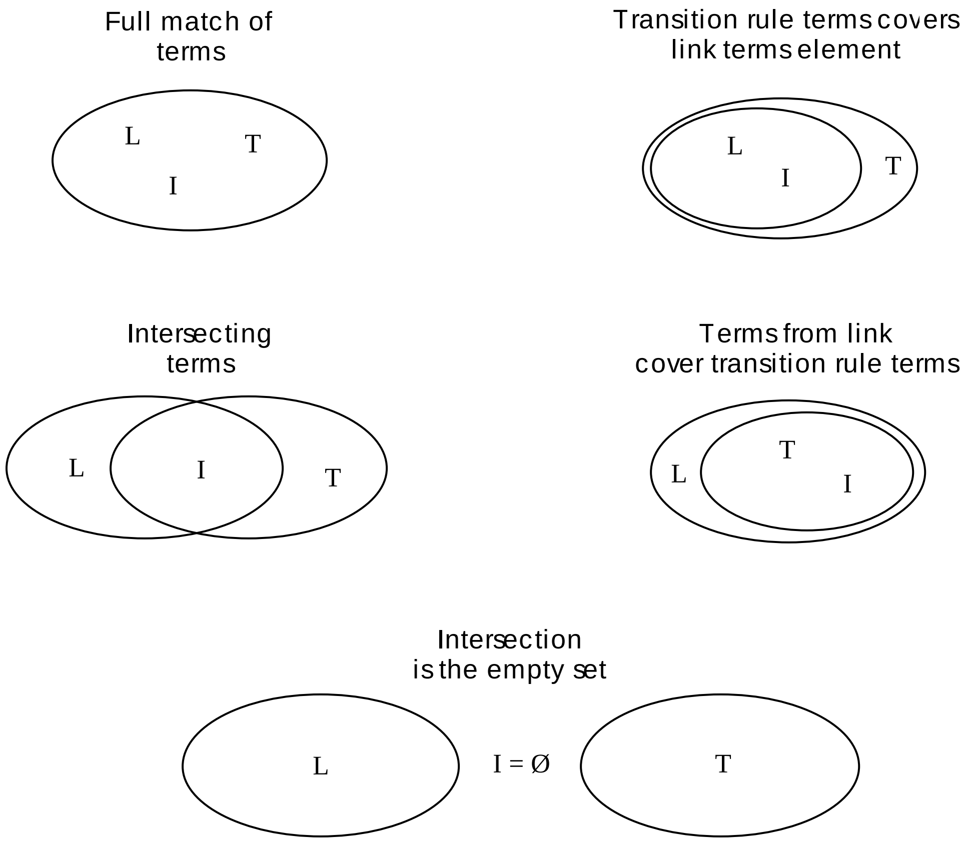

Full match of terms
L
T
I
Transition rule terms covers link terms element
L
I
T
Intersecting terms
L
I
T
Terms from link cover transition rule terms
T
L
I
Intersection is the empty set
L
I = Ø
T

Figure 6.4: Examples of different matches when comparing parts (sets)

In the next section, we will define the different features used to measure closeness between a link and transition rule, based on the definitions in the previous section and examine how to translate the features into numerical values.

6.4 Features for Measuring Closeness of Behaviour

Feature 1, based on definition 1, same external triggering condition (stimulus).

Can the transition rule and link be triggered by the same external stimulus?

Feature 1 is a straightforward match between the stimulus part of the links and the transition rules (see Figure 6.3, comparison c1).

If a link and a transition rule have the same stimulus as their triggering condition, feature 1 may be used as an indication that it is relevant to analyse them further for similarity. For example, if a link has the triggering stimulus *hook_on* and a transition rule has the triggering stimulus *hook_on,* it is obvious that the link and transition rule will trigger in the same situation if all other conditions and arguments are equal. We can also conclude that a transition rule with the triggering stimulus *dialling* cannot trigger in the same situation as the *hook_on* link (no parallel stimuli are allowed in the CABS model of the telecommunications domain). Since links and transition rules are restricted to having only one triggering stimulus, the match can either be full (the intersection between the two stimuli sets is equal to the triggering stimulus in the link and the transition rule), or empty (the intersection is the empty set). Intuitively, we can draw the conclusion that any transition rule not having the same triggering stimulus as the link cannot capture the same behaviour and that this is sufficient to exclude the transition rule from further investigation, thus reducing the search space considerably (see Figure 6.5 for how the matching in such a case is more efficient).

The difference between definition 1 and feature 1 is that feature 1 matches the stimulus name but makes no full analysis of the arguments (exemplified below). Feature 1 will give good results if the term name bears high significance (as described in Chapter 5.1.1). A successful match for feature 1 would occur when the stimulus *dialling(a1, 123, 12:00)* in a link is matched with the stimulus in a transition rule *dialling(A, Nr, Time)* and where no variables are bound to some other values throughout the transition rule (see Appendix A for details on logic). An example in which feature 1 would reduce the score is when *switch_service_on(a1, redirect, 123, 12:00)* is matched against *switch_service_on(UserA, hotline, Number, Time)*. The second argument (*redirect* and *hotline*) are not equal. A difference between feature 1 and definition 1 would occur in the situation where two variables, or one variable and one constant, are matched and later on in the condition part of the transition rule are bound to a specific value. For example, if *switch_service_on(a1, redirect, 123, 12:00)* is matched against *switch_service_on(UserA, Service, Number, Time)* and the conditions in the transition rule contains the term *equals(Service, hotline)*, feature 1 would not identify the binding of variable *Service*, since at this stage of the match, no analysis of the condition part is made. The main reason for this is efficiency: a large number of transition rules can be excluded from further matching at a low computational price, hence the decision was made to not include further analysis of variable binding at this stage of the matching (see Figure 6.5) in order to be able to exclude some additional transition rules.

CABS also allows the definition of similar stimuli. This facility can be used if there are stimuli which have different term names, but a similar semantics in the application domain. An example in the telecommunications domain would be the origination of a call which may be initiated in two ways, either by dialling a number (*dialling* stimulus) or by a *set_up* stimulus from an ISDN terminal. Thereafter, the matching algorithm will treat them as the same stimulus for matching purposes.

Feature 2, based on definition 2, exclusive transition rules:

Is there any contradiction, such that the behaviour in the transition rule cannot include the behaviour exemplified in the link?

The cross comparisons between the non-negated and negated parts of the link and transition rule (cn2, nc2, cn3, nc3, cn4, nc4 in Figure 6.3) are most useful in determining if a transition rule is of low or no interest for further investigation. If a contradiction exists between the link and transition rule, they cannot capture the same or similar behaviour and we may exclude the transition rule from further investigation. When matching the arguments to terms, there are situations in which it is difficult to determine if it is a real contradiction or just appears to be one (e.g. whether *answer_number(A,B)* and *not answer_number(C,D)* is a contradiction or not). If unbound variables exist in both negated and non negated forms in the link or transition rule (see the example at the end of this section) we take the conservative approach and do not classify this as a negation. With this conservative approach, exclusion of transition rules that may be appropriate candidates is avoided.

An example of the successful identification of a contradiction between a link and a transition rule (example of comparison cn2 in Figure 6.3) is when the condition part of a link has the term *dial_tone(a1)*, the transition rule has the condition *not dial_tone(UserA)* and *UserA* has been instantiated to *a1* by matching the stimulus (the only way of binding arguments during matching). A more difficult example would be if a link has the condition *answer_number(a2, 222) & ...* and a transition rule has the conditions *answer_number(UserB, Nr1) & not answer_number(UserC, Nr2) & ...* . In this situation, it is difficult to determine if there is a real contradiction. Since feature 2 does not perform a full analysis of arguments, feature 2 cannot discriminate between the negated and non-negated term, and should not be reason enough alone to exclude a transition rule.

After identifying and removing matches with contradictions above the user-set threshold in Figure 6.6, the numerical value of contradictions (the sum of the number of terms in the intersections for cn2, nc2, cn3, nc3, cn4, nc4 in Figure 6.3) is calculated. Since all the other comparisons have a percentage value between 0 and 100 apart from feature 2, we translate it with a linear function to a percentage value where 100% signifies no contradictions and 0% signifies the maximum allowed number of contradictions. If the maximum number of contradictions is set to 0, then the value for feature 2 is 100% for all transition rules that are scored. In this case, it does not make sense to give feature 2 any weight in the final scoring. If the maximum number of contradictions is C_{max} and the number of contradictions is C_{tot} and $C_{tot} = C_{max}$ and $C_{max} > 0$ then the ICL and ICT are set to $100 - 100*C_{tot}/C_{max}$ for feature 2. The fact that feature 2 is calculated in a different way from the other features may require a careful selection and tuning of the weight for feature 2 (see Chapter 6.5.1).

Feature 3, based on definition 3', some originating states in common:

Can the transition rule trigger in the same or similar situation ?

For feature 3, we can directly apply the result from comparison c2 and n2. If the intersection of the conditions of the link and transition rule is empty, it is less likely that a behaviour similar to the link is captured by the transition rule. If the intersection captures most of the terms in the link's condition part, the behaviour of the link may be captured in the transition rule. The additional terms in the transition rule may be additional interactions and may be used to exclude special situations handled by a separate transition rule in the case. Since interactions are common in telecommunications services, we expect that there are more terms in the transition rule capturing interaction.

In the situation where the condition from the link has terms which are not present in the condition from the transition rule, it may be that the transition rule is more general and deliberately does not include these terms. A match is often better if most of the terms from the link are included in the transition rule. By setting the appropriate parameter values, the final scoring will rate this as an indication of a possibly good match and use the result to create an overall score of closeness for the transition rule.

An example of a successful indication of a similar behaviour using feature 3 is if the condition part of a link is *answer_number(a1, 111) & redirect(111, 222) & answer_number(a2, 222) & not calling(Z, a2)*, and the conditions in a transition rule are *answer_number(A1, Nr1) & redirect(Nr1, Nr2) & answer_number(A2, Nr2) & not calling(Z, A2) & not dont_disturb(A2)*. In this example, the condition part of the link is a subset of the condition part in the transition rule, so there exists at least one state in which both condition parts are true.

An example of a match in which there is a difference in the result between feature 3 and definition 3' is a link that has its condition part equal to *answer_number(a1, 111) & redirect(111, 222) & not dont_disturb(222)* and a transition rule that has its condition part equal to *answer_number(A1, Nr1) & call_back_request(Nr1, A1) & not dont_disturb(Nr1)*. In this situation, feature 3 identifies that the terms *answer_number* and *not dont_disturb* are present in both condition parts, but that the rest of the condition terms are different. Feature 3 would give the match some significance but since the *not dont_disturb* is actually two different identities in: *answer_number(a1, 111) & not dont_disturb(222)* and the same in: *answer_number(A1, Nr1) & not dont_disturb(Nr1),* they would not be regarded as equal by definition 2' since Nr1 and Nr2 cannot have the values 111 and 222 at the same time), only one of the terms would count as a match. In some application domains, feature 3 may be preferred, since definition 3' may exclude interesting matches.

The numerical results for feature 3 are based on the conditions for the link and transition rule (stimulus excluded for both). These two sets of terms are translated into the numeric ICT and ICL values (in accordance with Section 6.3).

Feature 4, based on definition 4', some terminating states in common**.**

Can the transition rule end in the same or a similar state as the link

If the conclusions from the link and the transition rule match fully, it would signify that both are causing the same changes to the states to which they apply (responses not considered). This is a similarity that may be worth noticing even if there is not a full match in the conclusions. In the telecommunications domain, a transition rule may include conclusions needed for other services, for example, to note the starting time of a call in order to provide the charging service with sufficient information. It may also be the case that the link has omitted terms in the conclusion which are not obvious to the user making the input examples.

Situations may also occur when a link includes conclusions that are redundant and are known to be already true in the previous situation and, hence, a match, as shown in Figure 6.4, middle left example, is expected. For example, if a user puts the phone down (*hook_on*), we may specify a generic transition rule concluding that the user is idle. If this transition rule always triggers when a *hook_on* stimulus occurs, other transition rules can ignore this conclusion. If accuracy of matches of an application domain specified with parallel[20] transition rules, gives poor results for feature 4, adapting the matching of feature 4 to consider transition rules that may apply in parallel could improve the matching result.

An example of the successful indication of a similar behaviour by feature 4 is when the conclusion part of a link is *calling(a1, a7)* and the conclusion of a transition rule is *calling(A1, A2) & last_call(A1, Nr)*. In this example, the conclusion part of the link is a subset of the conclusion part of the transition rule and, therefore, there exists a state in which both conclusion parts are true.

An example of a match where there is a difference in the result between feature 4 and definition 4' is a link that has its conclusion part equal to *calling(a1, a7) & last_call(a1, 777)*, and a transition rule that has its conclusion part equal to *calling(reminder, A2)*. In this situation, feature 4 identifies that the term *calling* is present in both conclusion parts, but that the rest of the conclusion terms are different. Feature 4 would give the match some significance but overlooks the fact that the transition rule could never match the link if the arguments are those set out for definition 4' (a call from a "reminder" is a special case where the service reminder call initiates a call and where the reminder is not an ordinary user).

The numerical results for feature 4 are based on the comparison between the conclusions for the link and transition rule (c3 and n3 in Figure 6.3). These two sets of terms are translated into the numeric ICT and ICL values according to Section 6.3.

Feature 5, based on definition 5', some external visible effect in common**.**

[20] Not to be confused with parallel stimuli which are not allowed in order to avoid indeterminism and added complexity. See Model of the dynamic behaviour of telecommunications network, Figure 5.4.

Is the externally visible result (responses) from the link included in or similar to the responses from the transition rule?

If response terms from the link and the transition rule fully match, it would mean that both may result in a state with the same response. In telecommunications services, this is an important indication that it may be a good match but, on its own, it is often too general (many different transition rules have responses such as *ring_signal/ ring_tone* in their conclusions). On the other hand, if the response terms do not match, it is less likely that it is a good match, assuming the user has specified the externally visible side effects accurately (in telecommunications services, the side effects alone are rarely affected by interaction with other services). For example, if a link ends in a situation with a *ring_signal,* transition rules with no *ring_signal* as a conclusion are probably not good candidates, and transition rules having *ring_signal* as a conclusion would be candidates for further analysis.

An example of a successful indication of a similar behaviour by feature 5 is if the conclusion part of a link is *not ring_tone(a1) & not ring_signal(a2)* and the conclusion in a transition rule is *in_speech(A1, A2) & not ring_tone(A1) & not ring_signal(A2).* In this example, the response part of the link is a subset of the response part of the transition rule so there is at least one state in which both response parts are true. As with previous features, there is a risk that feature 5 gives a match too much credit since no in-depth analysis of arguments occurs.

The above example may give too much weight to some transition rules since the link does not reveal if user a1 has made a *hook_on* (*ring_tone and ring_signal* have to be cancelled) or if user a2 has made a *hook_off (ring_tone and ring_signal* have to be cancelled since a speech connection has occurred which is a completely different situation and transition rule). In most cases, the combination of features reduces the risk of such mistakes and in the above case, feature 1 would have indicated that the stimulus does not match between the link and the transition rule, and so the transition rule should not be used in further investigations.

The numerical results for feature 5 are based on the comparison between the conclusions for the link and transition rule (c4 and n4 in Figure 6.3). These two sets of terms are translated into the numeric ICT and ICL values are in accordance with Section 6.3.

6.5 Overall Score for Matching

First, we have to produce an overall score for each transition rule that is a candidate for a link from the input examples. When that is done, we need to produce an overall score for cases (sets of transition rules) in the case library. After the best matching transition rules and cases have been identified, both of these results are shown to the user, who must decide if the match is good enough, or if the input examples need to be extended or the matching parameters tuned. First, we describe the process of scoring transition rules and after that, we describe the scoring of the cases.

In order to make a rating of the closeness of transition rules, the results from comparing these different features and their values are weighted and combined into one value (according to the matching parameters set by the user). This value is then used as a measurement of the closeness between a link and transition rule. In order to adjust the match parameters for a domain, these comparisons and their meaning have to be understood. In the following sections, we explain how an overall score is calculated for a comparison, when transition rules are excluded from further calculations, and how the ranking of transition rules and cases is performed.

6.5.1 Scoring a Match Between Link/Transition Rule

The algorithm for calculating features, reducing the search space and calculating the final score for a match between a link and transition rule is outlined in Figure 6.5. There are two types of parameters that can be adjusted in CABS:

Threshold parameters reducing the search space by excluding uninteresting matches.

Parameters guiding the overall scoring of a match (capturing information about the validity of different features and their relationship in the application domain).

Much computational effort can be saved by excluding transition rules from further calculations: to minimise the calculations, the user set threshold values are checked after each feature is calculated. If the result is below the user set threshold, the transition rule does not need further investigation and the next transition rule can be explored (see Figure 6.2). The main purpose of the threshold for the features is to make the matching faster and to reduce the search space (with one exception, which is explained further on). Another advantage with the threshold settings is that some of the application domain knowledge about when a transition rule is uninteresting and can be exempt from further calculation, is captured.

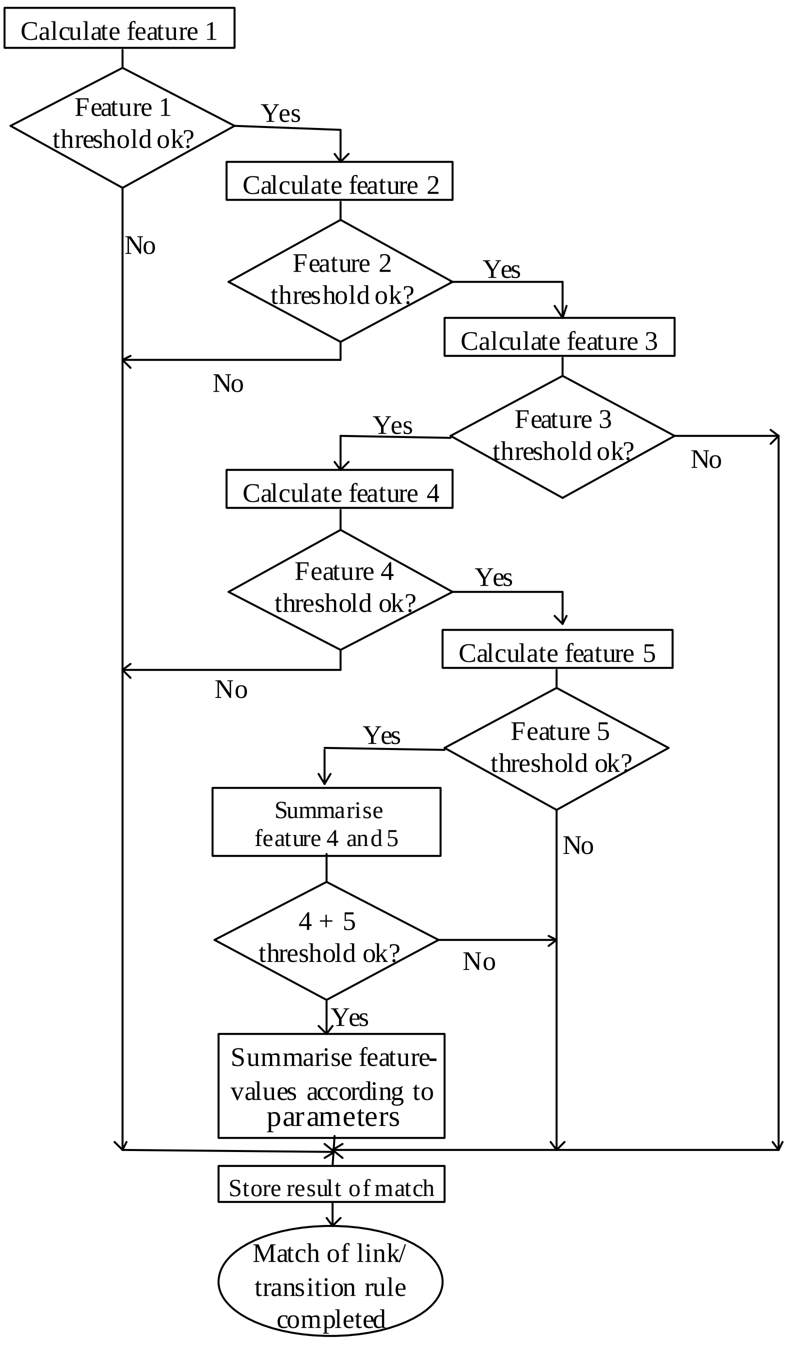


Figure 6.5: Flow diagram for link/transition rule match

The different threshold values have to be selected carefully, so that they do not exclude relevant matches within a particular case library. If these values are set too high, good matches may be removed before the final scoring. Once the values have been tuned for a particular case library (and do not exclude interesting cases), they do not need much attention. CABS provides a default setting of these parameters, which is initially set and tuned for the case library currently used (these default values may need updating if the case library changes greatly). The experienced user can also load and save settings of threshold and parameter profiles. These may be used if the matching algorithm is identifying less acceptable matches. Less acceptable matches can have three causes:

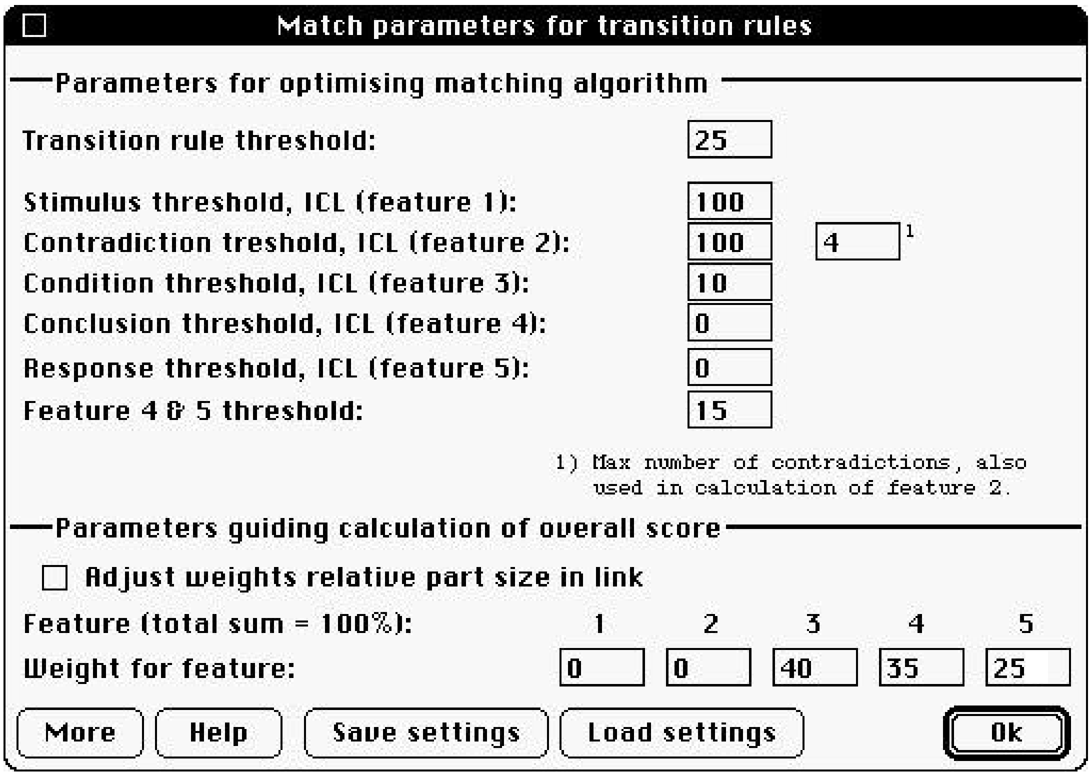
The input examples do not point out suitable cases well enough. Solution:
- Add/refine input examples.
- Exempt proposed transition rules and cases from a rematch.

There is no good match in the case library. Solution:
- A new case may have to be constructed/generated.

Threshold and parameter setting are not well chosen for the case library. Solution:
- Load an alternative set of threshold and parameter values and rematch.
- Modify threshold and matching parameters.

The threshold and parameter settings seem to be fairly robust for both the telecommunications domain and the case library used for evaluation (see Chapter 8).

After all the features have been calculated, an overall score for each transition rule is calculated. For this overall score, an overall threshold value can be set; if a transition rule does not meet this threshold it will not be considered as a match to the corresponding link (see *Transition rule threshold* in Figure 6.6). If this value is not met, the match will neither be used for the identification of matching cases (see Section 6.5.2 on parameter and threshold settings for cases), nor presented to the user as a possible match for a link. For more detailed settings and optimisation of matching, there are five individual threshold settings for each of the five features (see Figure 6.6). Only the ICL (Intersection Coverage of Link) is used for thresholds, since ICL is the most significant value. For feature 2, there is an additional value where a maximum number of suspected contradictions is set. This value is also used in the calculation of feature 2's value, as explained in Section 6.4. There is also a separate threshold value for the combination of features 4 & 5. The combination of features 4 & 5 is used when a case library may have cases that are of interest if at least one of the features has good scores (which is the case for some transition rules in the telecommunications domain). These weights should provide sufficient opportunities for tuning the matching for case libraries for different application domains.


Match parameters for transition rules

Parameters for optimising matching algorithm

Transition rule threshold:	25	
Stimulus threshold, ICL (feature 1):	100	
Contradiction treshold, ICL (feature 2):	100	4 [1]
Condition threshold, ICL (feature 3):	10	
Conclusion threshold, ICL (feature 4):	0	
Response threshold, ICL (feature 5):	0	
Feature 4 & 5 threshold:	15	

1) Max number of contradictions, also used in calculation of feature 2.

Parameters guiding calculation of overall score

☐ Adjust weights relative part size in link

Feature (total sum = 100%):	1	2	3	4	5
Weight for feature:	0	0	40	35	25

Figure 6.6: Parameters for transition rule match

When all the features have been calculated, we have to calculate an overall score for each relevant match. Calculation of an overall score is based on domain knowledge that captures the value of the different features for the application domain. In the telecommunications domain, stimulus and response terms usually have higher significance than other conditions and conclusions, and hence should contribute more towards the final score than other terms in the conditions and conclusions. In fact, the example setting in Figure 6.6 has the stimulus threshold set to 100% and transition rules that do not have the same triggering stimulus as the link are exempt from further matching. Therefore, there is no need for a weighting of feature 1 (see *Weight for feature*, field 1), as we know that all matches qualifying for an overall score calculation, have the value 100 for feature 1.

The ICL and ICT value for every feature in a match is used to calculate a total ICL and ICT value for the transition rule. If all weights are set equally and the weighing is not adjusted according to the number of terms in the link, the total score for ICL and ICT respectively would be the sum of all the values for the features divided by the total number of features. In the generic formula for the calculation of ICL and ICT scores for a match, TotTerms is the total number of terms from the link used in the calculation of the features, F_n(ICL) and F_n(ICT) are the ICL and ICT scores for the feature n, WF_n is the weight for the feature n and LF_n is the number of terms of the part in the link on which the calculation is based. The total score is a pair of values, where the ICL value is given the highest significance. When sorting all matches for a link, the matches with the highest ICL will come first and matches with the same ICL will be ordered according to their ICT value. f is the set of features used for calculating the total score. If a feature weight is set to zero, it is not used in calculating the total score. x is either L or T.

The total score for the ICL or ICT is calculated as:

$$\text{SCORE}(IC_x) = \sum \text{weighted\_score}(F_n(ICx), LF_n, \text{TotTerms}, WF_n)$$

The weighted score for a feature is calulated by the formula:

$$\text{weighted\_score}(F_n(ICx), LF_n, \text{TotTerms}, WF_n) = \frac{F_n(ICx) * WF_n * LF_n}{100 * \text{TotTerms}}$$

If the check box *Adjust weights according to number of terms in link* is unmarked, then LF_n and TotTerms are both set to the value 1 before the calculations start.

6.5.2 Scoring a Matching Case

After all transition rules have been scored, the task for the matching algorithm is to identify cases capturing similar behaviour to the input example. The overall score for each case depends on the matches between the transition rule in the case and the links in the input examples. If we look at a particular case, C1, from the case library (see Figure 6.7), some of the transition rules (squares) are matches for links in the input examples, indicated by broken lines to the matching link. The example in Figure 6.7 has six matches (m1 to m6) between links from the two input examples, E1 and E2 (the two input examples are indicated by broken circles around a group of links).

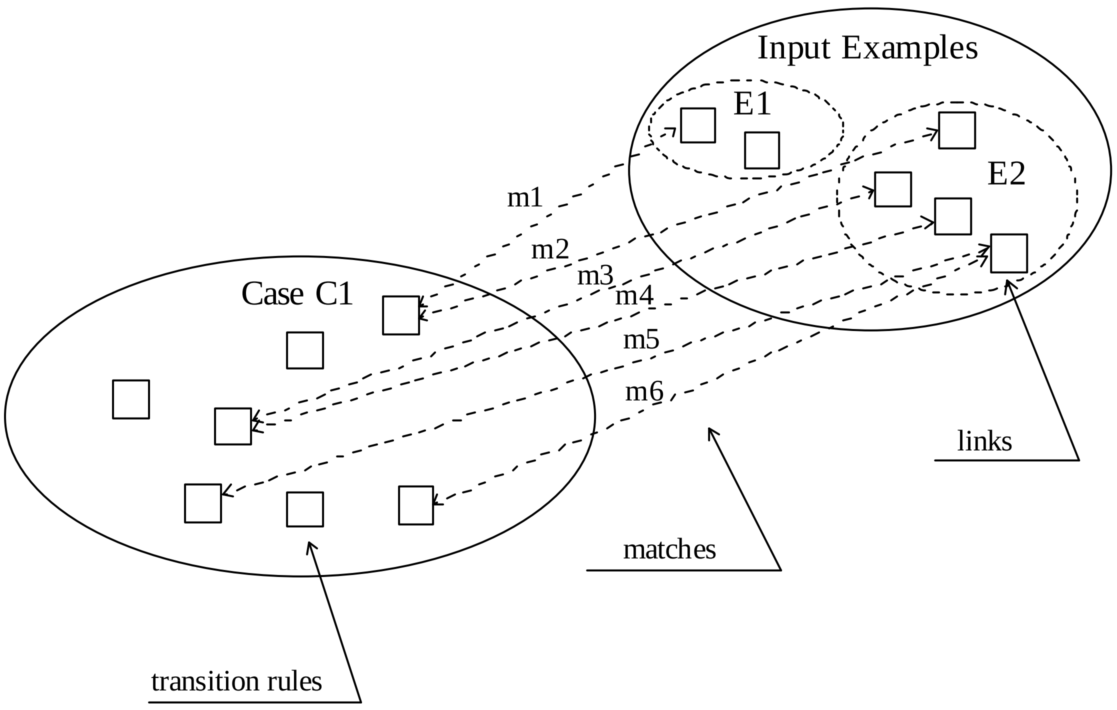


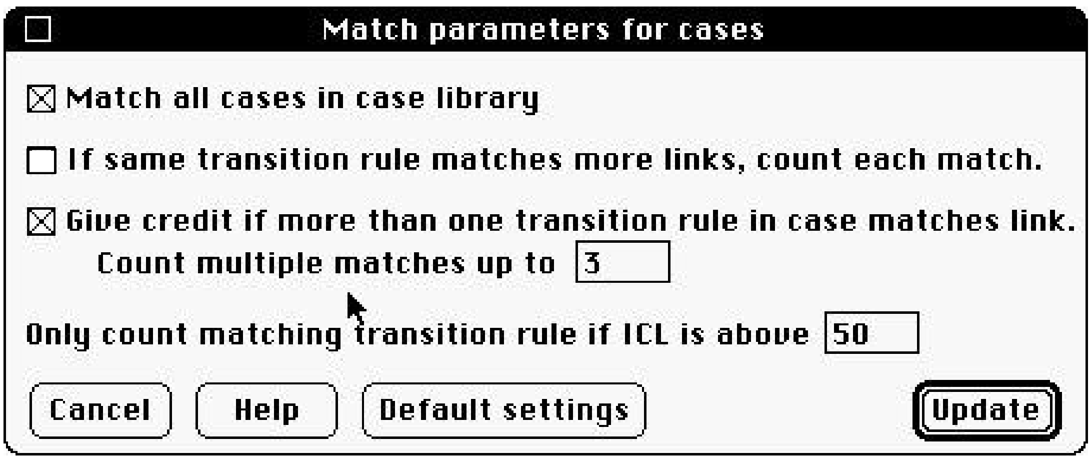
Figure 6.7: A match of a case and an input example

If the *Always match cases* box is selected in Figure 6.8, CABS will identify and rank similar cases (for some situations only matches of transition rules may be relevant). To score a case, the matching algorithm counts the matches between all links and the transition rules in the case (m1 to m6 in Figure 6.7). A case with a greater number of matches is ranked higher than a case with a lower value. This naive approach seems to be accurate enough (see evaluation in Chapter 8) in most instances of identifying cases of relevance, after adjustment of some additional parameters guiding the final ranking has occurred.

If a transition rule in the same case is matched by more than one link (an example of this is match m1/m2 and m3/m4 in Figure 6.7), we do not know if the transition rule is capturing many different transitions, if the links in the input examples are a repetition of a similar link (for m1/m2), or if the application domain allows parallel transition rules to occur in the same case (for m3/m4). In our telecommunications service examples, we chose to allow parallel transition rules only if they are from different telecommunications services (different cases). If the application is specified with transition rules of a more general character (including a large number of transitions), then different links may be covered by the same transition rule. If the applications are specified with more specific transition rules, then the fact that the same transition rule is matched by more than one link may just be a less relevant match, and hence should not be included in the scoring. This choice is shown in Figure 6.8: the second choice *If same transition rule matches more links, count each match* is not selected.

A decision also has to be made as to what to do if there is more than one matched transition rule in the same case (m5 and m6 Figure 6.7). If the other transition rule captures a similar but not exactly the same behaviour, this information may be useful, since it may increase a case's relevance. The relevance for multiple matches can be set by selecting the third choice *Give credit if more than one transition rule in case matches link. Count multiple matches up to NR* in Figure 6.8. An upper limit, NR, on how many matches should be counted can also be set, in order to avoid over-scoring cases which have a large number of very similar transition rules (set to three in the example).

A parameter, defining a threshold value for when a transition rule should count as a match for a case, can also be set by the user (*Only count matching transition rule if ICL is above NR* in Figure 6.9). This is a different value than the threshold setting for the total score for transition rules. A score for a match passing the threshold set for transition rules allows the rule to be presented as a possible match for a link, but in order to be counted as a match for a case, the match has to pass this second threshold. If a large number of cases have a high score, the value may be set higher, to reduce the number of good matching cases.



Match parameters for cases

[✓] Match all cases in case library

[ ] If same transition rule matches more links, count each match.

[✓] Give credit if more than one transition rule in case matches link.

Count multiple matches up to 3

Only count matching transition rule if ICL is above 50

Figure 6.8: Parameters for case match

6.6 Presentation of Matching Results

When the system has completed the match, the result is presented to the user. Both the best matching cases and the best matching transition rules are shown. The user is asked to select a solution that she will use as the proposed solution (or refine the input examples so a better match may be achieved). Figure 6.9 shows an example of a result from matching two input examples: *a_basic_example* and *a_busy_example*. In the upper left corner under the text *Best matching cases (descending order)*, a scrollable list with the best matching cases from the case library is shown. The number in brackets after the name of the matching case tells the user how many links from the input examples are matched by the case. The user may inspect a matching case by selecting the case in the list and pressing the button *Show Case*, which will result in the system showing the case window as shown in Figure 5.7. The *Exclude Case* button will be explained in Chapter 7.

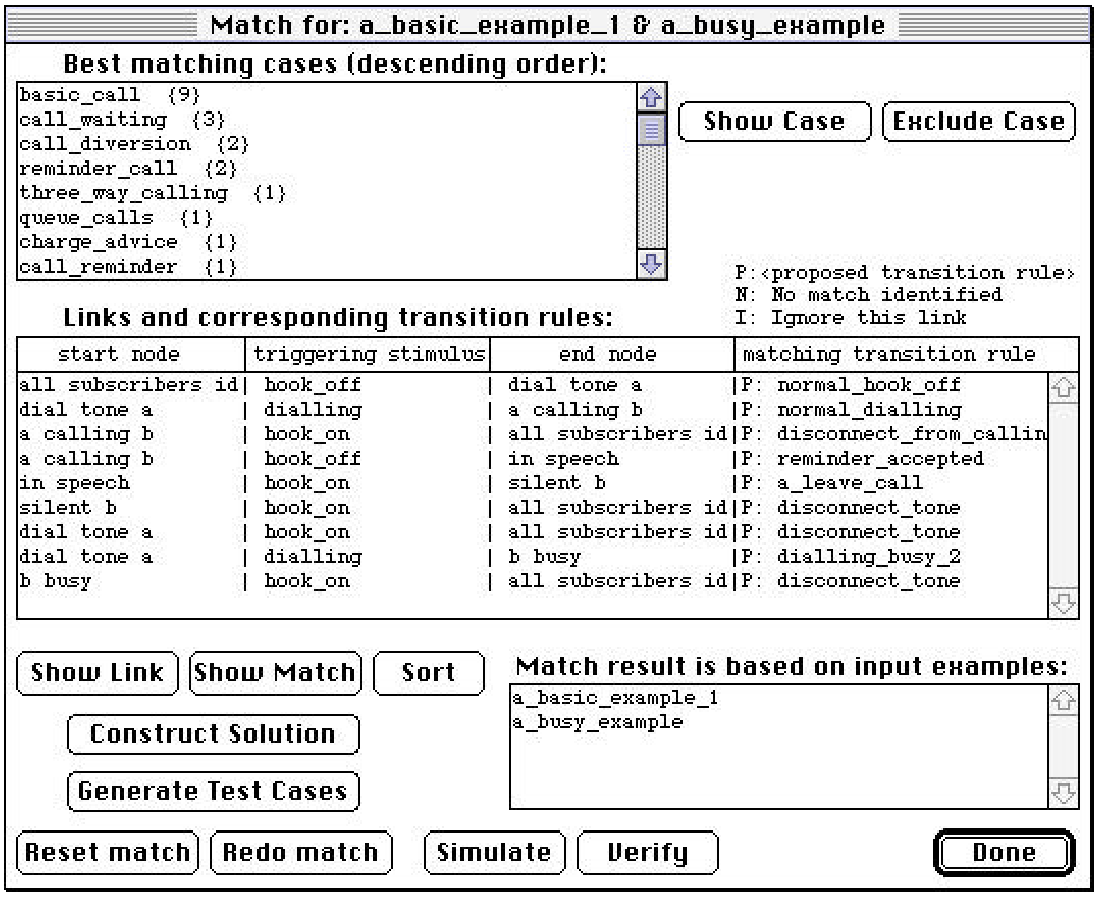
In Figure 6.9, *Links and corresponding transition rules* show the links from the input examples identified by their start node, triggering stimulus and end node. In the table *matching transition rule*, the proposed/selected transition rule is shown. There are five different types of prefixes to the transition rules:

P: <transition rule name> - The best matching transition rule in the case library according to the matching result is shown. If the user wishes to see all the matching rules (sorted in descending order) this can be viewed in the link window (Figure 4.6).

N: No match is shown when there is no matching transition rule in any case that meets the set transition rule threshold set in Figure 6.6.

I: Ignore this link - If the user has labelled a link to not be included in the match. This choice can be selected when showing the link. The user may set this if it is obvious that a link captures behaviour from another case on which the new case is dependent. In telecommunications, it could be a service based on a basic call and therefore, getting the proposal *basic_call* as the first and best proposal may not be useful. By pointing out those links that are not crucial for the new functionality, the matching result is narrowed down to find cases that capture the selected parts of the input examples.

The user can inspect a link in more detail by selecting the link in the list and pressing the button *Show Link* in Figure 6.9, which results in a link window showing the selected link in detail. If there are many links, the user may wish to sort the links after the start node, stimulus, end node etc. This can be selected by pressing the button *Sort list* (these sorting choices are not fully implemented in CABS).



Best matching cases (descending order):

```
basic_call  {9}
call_waiting  {3}
call_diversion  {2}
reminder_call  {2}
three_way_calling  {1}
queue_calls  {1}
charge_advice  {1}
call_reminder  {1}
```

P:<proposed transition rule>
N: No match identified
I: Ignore this link

Links and corresponding transition rules:

start node	triggering stimulus	end node	matching transition rule
all subscribers id	hook_off	dial tone a	P: normal_hook_off
dial tone a	dialling	a calling b	P: normal_dialling
a calling b	hook_on	all subscribers id	P: disconnect_from_callin
a calling b	hook_off	in speech	P: reminder_accepted
in speech	hook_on	silent b	P: a_leave_call
silent b	hook_on	all subscribers id	P: disconnect_tone
dial tone a	hook_on	all subscribers id	P: disconnect_tone
dial tone a	dialling	b busy	P: dialling_busy_2
b busy	hook_on	all subscribers id	P: disconnect_tone

Match result is based on input examples:

```
a_basic_example_1
a_busy_example
```

Figure 6.9: Presentation of result from match

If the user does not accept the proposal in Figure 6.9, she can add input examples and redo the match, which will hopefully result in a solution that can be accepted as a proposed solution (although it may need refinement). For this purpose, the button *Exclude Case* can be used when there are proposals in the best matching cases list that have been inspected and are not relevant. Chapter 7 explores how the user selects, revises, validates and verifies the solution selected in Figure 6.9.

Chapter:

7. The Requirements Design Process in CABS

In the previous chapters, we looked closely at the central parts of CABS and explained the graphical input examples, the case library and the matching process. In this chapter, we put these parts in the context of requirements design and examine how a requirements designer may use such a system to produce formalised, validated and verified requirements. The examples are given in the context of the chosen application domain, where the most common task is to modify and extend a large system (a large number of closely interacting telephone services) and where the requirements designer is not necessarily an expert at applying scientific methods in order to produce requirements. CABS aims to simplify the task of requirements engineering so that a person with some idea about a new or modified behaviour can outline their ideas, and then refine, validate and verify them. Graphical input sketches, case-based reasoning and formalisation are tools used in combination to aid this creative process and are not aims in themselves. Persons performing this task may be service vendors, sales staff or even end-users of the telephone system (or any combination of these), who would benefit from being able to express and formalise their behavioural requirements. For this reason, we have adopted the terms: *requirements design* and *requirements designer* instead of the traditionally used *requirements engineering* and *requirements* *engineer* which, for many people, imply some technically advanced and complicated task. Design often implies a more creative process, such as outlining and sketching an idea, so is a better choice of name for the task CABS aims to support and simplify.

Modifying and adding behavioural requirements to a requirements specification mostly includes refinement cycles. When an idea for a new behaviour has been formalised, validated and verified, a large number of iteration and refinement steps may have occurred. In CABS, these cycles are treated as central parts of the process of producing requirements. In Figure 7.1, the whole process from idea to a validated, verified and formalised requirement is outlined. The process of producing a requirements specification starts with an idea for a new behaviour (the top of Figure 7.1). In the application domain of telecommunications it is most likely that the new behaviour is being added to some already specified behaviour. The first step is to decide if the new behaviour can be expressed within the existing ontology or if the ontology has to be extended (see Section 7.2). Once the ontology is approved, the requirements designer can provide input examples outlining the main behaviour with the graphical input editor in CABS (third oval from top in Figure 7.1, see Section 7.3). Once the user has expressed some parts of the new behaviour with input examples, including some refinements of nodes and links as described in Chapter 4, the matching can start. The matching will identify candidates from the case library as described in Chapter 6. The user selects a solution and validates the selected solution. If the user does not accept any of the solutions proposed by CABS, the user has three choices, i1, i2, i3 (which are also shown in Figure 7.1). These are:

i1. The user believes that there is some fundamental problem with the idea of the behaviour to be specified. This is a restart and it may be necessary to modify the idea, ontology and input examples. In Figure 7.1, this situation is shown with the arrow pointing to *Revise Idea*.

i2. The user decides to refine or add new input examples which may be based on the assumption that the current input examples do not capture the behaviour to be specified well enough (*Refine Input Examples* in Figure 7.1).

i3. The user assumes that the result from the matching can be improved by adjusting matching parameters and modifying these before a rematch is carried out (*Prepare for Re-match* in Figure 7.1).

Once a solution has been selected (based on the matching result) the next task is to validate the proposed solution with the simulator provided (see Section 7.5). If the validation results in a rejection of the proposal, the user has the same choices as described when the matching result is rejected (i1, i2, i3 in Figure 7.1), as well as an additional choice, i4, of revising the solution, which is a more traditional way of modification where the user may edit the transition rules (described in Section 7.5.1).

If the validation is successful, and the user is convinced that the intended behaviour is captured by the proposed solution, the solution has to be verified. The input examples are used to generate test sequences (called *test cases*) of behaviour that should be included in the formalised solution. These are automatically or semi-automatically verified against the formalised solution. If the case includes all behaviour that is included in the input examples, the verification against the input examples is successful. If the verification is unsuccessful, the skilled user may use the feedback from the verification in order to locate the problem and modify the solution (i4, *Revise Solution* in Figure 7.1), or iterate back via i3, i2 or i1. The text to the right in Figure 7.1 is the part (or parts) of CABS aiding the process/step to its left.

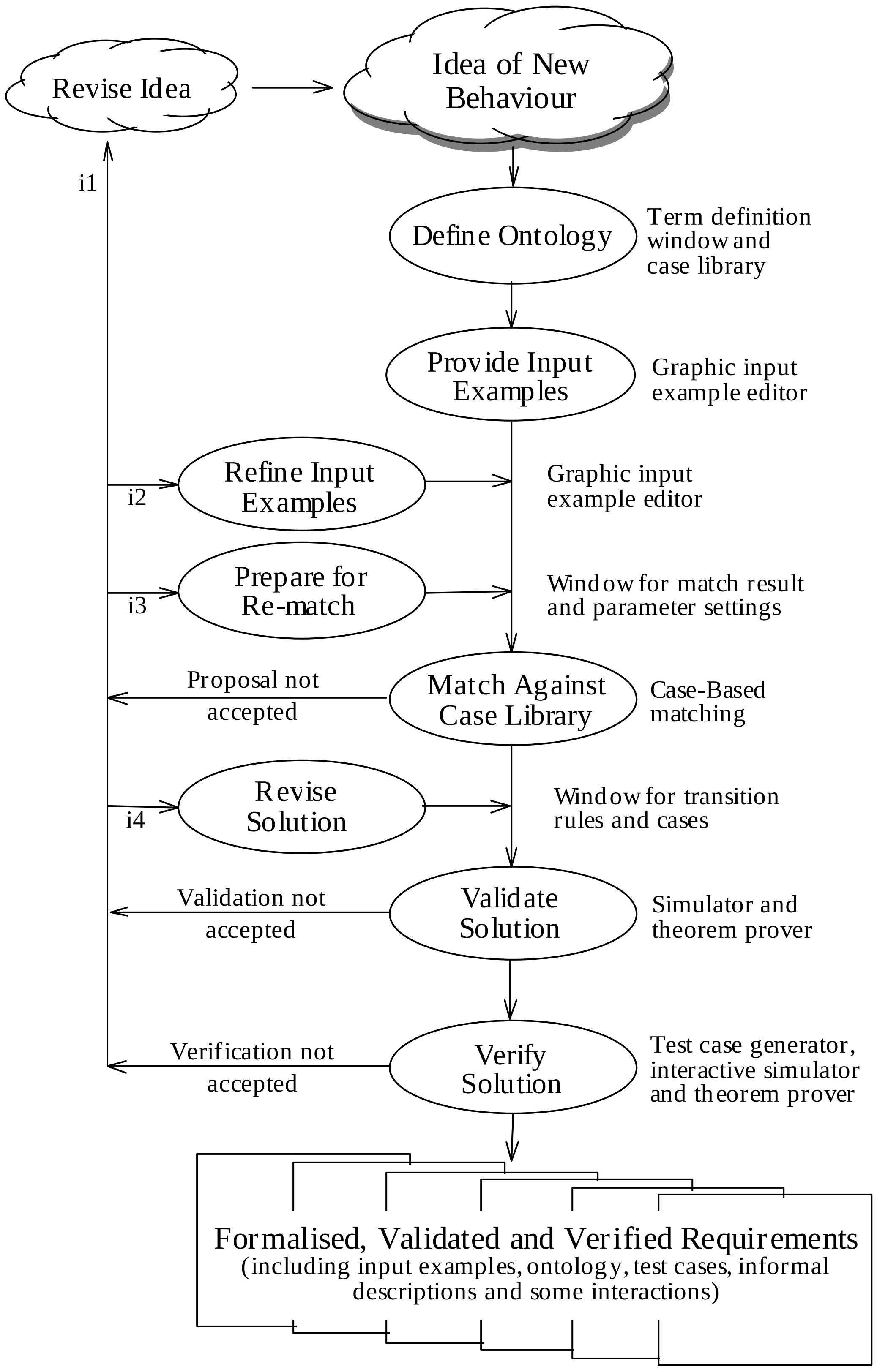

Revise Idea
Idea of New Behaviour
i1
Define Ontology
Term definition window and case library
Provide Input Examples
Graphic input example editor
Refine Input Examples
i2
Graphic input example editor
Prepare for Re-match
i3
Window for match result and parameter settings
Proposal not accepted
Match Against Case Library
Case-Based matching
Revise Solution
i4
Window for transition rules and cases
Validation not accepted
Validate Solution
Simulator and theorem prover
Verification not accepted
Verify Solution
Test case generator, interactive simulator and theorem prover
Formalised, Validated and Verified Requirements
(including input examples, ontology, test cases, informal descriptions and some interactions)

Figure 7.1: Overall process from idea of behaviour to formalised solution

7.1 Idea for New Behaviour

Before starting a new specification, an idea of the behaviour to be added has to be created (the “cloud” marked *Idea for New Behaviour* at the top of Figure 7.1, with the cloud indicating that the idea is a mental product “stored” in the users mind[21]). The initial idea is, by its nature, always implicit since it is in the head of a person or a group of people. Often, the overall goal with an idea is to add some behaviour to an existing implemented behaviour in order to add value to the total behaviour (in telecommunications, this is called an *added value service*). In CABS, the main concern is the process of formalising an idea for a new behaviour so it can be validated and verified before any larger commitments, in time and money, have been made, and also provide a basis for decision making, design and implementation.

7.1.1 Revising an Idea for Behaviour

If the requirements designer for some reason decides to rethink the idea of the behaviour (major changes, for refinements see 7.3), all steps after the initial *Idea for New Behaviour* in Figure 7.1 have to be performed again. Revising an idea may involve respecification of ontology and may require major changes in input examples. Revising the behaviour at this stage (within CABS) is not a major disaster because, at this stage, only a small investment in the new functionality has taken place (a few hours work). Most likely parts of the previous formalisation of the idea can be re-used by manually copying ontology, input examples or parts of input examples and even parts of the solution that could be re-used by refinement.

[21] For more on mental representation both from a philosophical perspective and in the context of theories of cognition see [Cummins 89].

7.2 Defining Ontology

Defining an ontology is a main issue in knowledge acquisition and in enabling re-use of knowledge. Many requirements specification approaches have neglected ontological issues (most likely due to more pressing problems) but their importance is now widely acknowledged and research into their use is increasing. The purpose of an ontology is to capture the conceptualisation of a domain and to define (informal, structured, semi-formal or formal [Uschold 96]) all relevant concepts and terms. There are three main areas in which an ontology is useful:

Communication between all involved parties.

Interaction between systems.

System design and engineering.

For CABS, the first area above is the most relevant: when a specification of a behaviour is made, it is essential that the entities, attributes and relations used in the specification have a clear meaning for all involved parties (customers, requirements designers and end users). The view taken in CABS is that information which is easy to capture and may be useful at a later stage (revision / design / implementation), should be captured at the earliest convenient stage. The definition of an ontology is not the aim and focus of CABS (it is in fact a research topic in itself), but defining an ontology is still a main part in the process for transforming an idea of a behaviour to a formalised requirements specification. Therefore, only a simple approach has been implemented in CABS where entities, attributes and relations are defined partly informally and partly formally. For the telecommunications domain it is often possible to identify and use previously specified definitions stored in the case library (which have been validated and verified). If not, any addition or modification of the ontology should be carefully investigated, validated and agreed upon by all involved parties, in order to minimise the risk of serious problems at a much later stage in the development process [Zave, Jackson, 96].

7.3 Expressing an Idea with Input Examples

As described in Chapter 4, the user can give a set of graphical input examples where each example exemplifies a category (categories such as *basic behaviour*, *odd case*, *error case,* etc.) or combination of categories of the new behaviour. Once the requirements designer has an idea for the behaviour, the behaviour is captured using the graphical examples that are produced with the graphical input example editor. Nodes and links are refined thereafter using definitions from the case library (the ontology of the domain). Once the requirements designer has outlined the main characteristics of the new behaviour with input examples, which capture the most common behaviour, whilst leaving out less usual behaviour, a match against the case library can be performed.

7.3.1 Refining Input Examples

Refining input examples is done with the graphical input example editor in the same way as new examples are produced. The user can copy and rename graphical input examples, as well as add, remove and modify links and nodes until satisfied. Links may also be excluded from matching for different reasons (some links may not be part of the new behaviour, merely putting the new behaviour in the context of previously specified behaviour).

7.4 Matching Input Examples Against the Case Library and Selecting a Solution

The matching process identifies cases in the case library, capturing similar behaviour to the behaviour exemplified in the input examples, as described in Chapter 6. This enables the requirements designer to identify and select a proposed solution.

7.4.1 Prepare for Match or Re-match

Before the user starts the matching process, he or she has to choose which input examples are to be used (Figure 6.1). If a match result is not satisfactory and a re-match has to be performed, selecting a different set of the input examples may be the preferred choice in an effort to improve the result of the matching. Some of the input examples may guide the matching better than others and there may even be input examples that misguide the matching (this will be explained further on). Since the final rating of cases is directed by the number of matching links/transition rules for the cases, it is obvious that if most input examples direct the matching in one direction, then a few input examples with links pointing to another case will have less effect on the final ranking. Matching parameters are normally not changed, but if matching using the method mentioned above (using different sets of input examples for the match) does not produce acceptable results, the user may consider tuning the matching parameters[22] in order to try to achieve a better matching result (Figure 6.6 and Figure 6.8). In the future, the system may also be involved in the process of improving the matching result by asking the user for some specific input examples, outlining the behaviour of parts of the functionality. This will enable it to confirm or exclude cases from the case library (an adaptive approach to case-based search [Callan, Fawett, Rissland, 91]). This possibility has not been explored in the current implementation of CABS.

If CABS proposes solutions that are rejected by the requirements designer, these proposed cases can easily be removed from further re-matches by selecting the proposals and pressing the *Exclude Case* button in Figure 6.9. In the same manner, the user may exclude links from the match if these are judged as being less relevant when searching for a matching case (these may be links that are known to belong to a case to which the new behaviour is complementary, but not included in, hence these links may direct the matching in an unwanted direction). When the user is ready for a re-match, the *Redo Match* button in Figure 6.9 is selected and a dialogue window is shown where the user can select the input examples on which the rematch will be based.

[22] Note that to tune the matching parameters, knowledge of the matching process is needed.

7.4.2 Selecting a Proposed Solution

When confronted with the matching result (as shown in Figure 6.9), the user must select a solution. The scroll list *Best matching cases (descending order)* may include a proposal that the user might decide to explore. The interface enables the user to inspect any of the proposals in the list by selecting the case and pressing the *Show Case* button. If the user accepts a proposal, the proposal has to be validated and verified (see sections 7.6). If the proposed case has been validated and verified, the task is completed and the user has identified a case that captures the required behaviour. In telecommunications, a case may be re-used directly or with minor modifications, if there is a variant of the service (a case that has been implemented for some other customer or market but where the main behaviour and functionality is matching) already specified and implemented. If no similar service is identified, the use of parts from different cases may be combined into a new service, which will be explained in the following section.

7.4.3 Adapting a Close Match

If there is a matching case that captures most of the main behaviour, but not all of the behaviour, the user may select this case as the proposed solution. Then, through validation and verification, he/she can locate the differences and construct a solution covering all wanted behaviour by adding transition rules from other cases (the transition rules may need modification, see Section 7.5.1). All links have their best matches shown in the menu *Match selected for link:* in the link window (Figure 4.6), where the user can select a matching transition rule that is not part of the proposed solution (a manual selection will by default exclude the link from a rematch). This allows the user to construct a new case with parts from other cases (modified or unmodified) by adding in missing behaviour. If some behaviour exemplified by a link is not included in the solution, this behaviour may be added in three different ways:

The user selects a transition rule from the case library which is good enough to be adapted and modifies it until it captures the desired behaviour.

The user lets the system generate a new transition rule capturing the behaviour of the link (how transition rules are generated from links is described in Section 7.4.4).

The user may manually construct a new transition rule.

In all three cases, validation and verification will identify if the transition rule is fulfilling its purpose. Once all links whose behaviour was not captured by the selected solution have been handled in this way, we have a solution that can be fully validated and verified. When transition rules are used from different cases and added to the new case, the new behaviour is a combination of parts from previous specified cases. In telecommunications, parts of behaviour in different services often show similarity (end users mostly require a uniform interface to services) and hence finding parts of behaviour from different services that can be used when specifying a new service is likely.

7.4.4 Generating a New Case

If there are no cases in the case library that can be re-used for the new behaviour, the input examples can be used to automatically generate a set of transition rules which can be used as a starting case. A solution case generated in this way will be a naive solution in the respect that it is merely a generalisation of the input links from all input examples only including the behaviour of the input examples. It is missing other wanted behaviour that has not been explicitly exemplified (error cases, odd situations, interaction, etc.) which would have been included if a previously specified, designed and implemented case had been re-used as starting point for the new behaviour. A generated case is most likely good enough as a starting point for refinements, modifications and adaptations, as described in Section 7.5.1.

CABS generates transition rules from the input examples by putting all conditions into the condition part of the generated transition rule and all conclusions into the conclusion part of the transition rule. Since most heuristics are most likely application domain dependent they should be given as an external set of rules enabling an easy way of changing them (the CABS prototype has not implemented these heuristics and the user has to do these adaptations manually). Since generating transition rules from input examples is not a main issue in CABS, this part is only briefly outlined and implemented to point at the possibility and to capture the situations where no good matching case or set of transition rules exist in the case library. This part is based on earlier experiments with rule induction [Funk 88], [Verpers 91]. There are interesting research results in the area of rule induction [Quinlan 87] and logic program induction [Muggleton 90] which should be used in order to extend this initial approach.

7.5 Validating a Proposed Solution

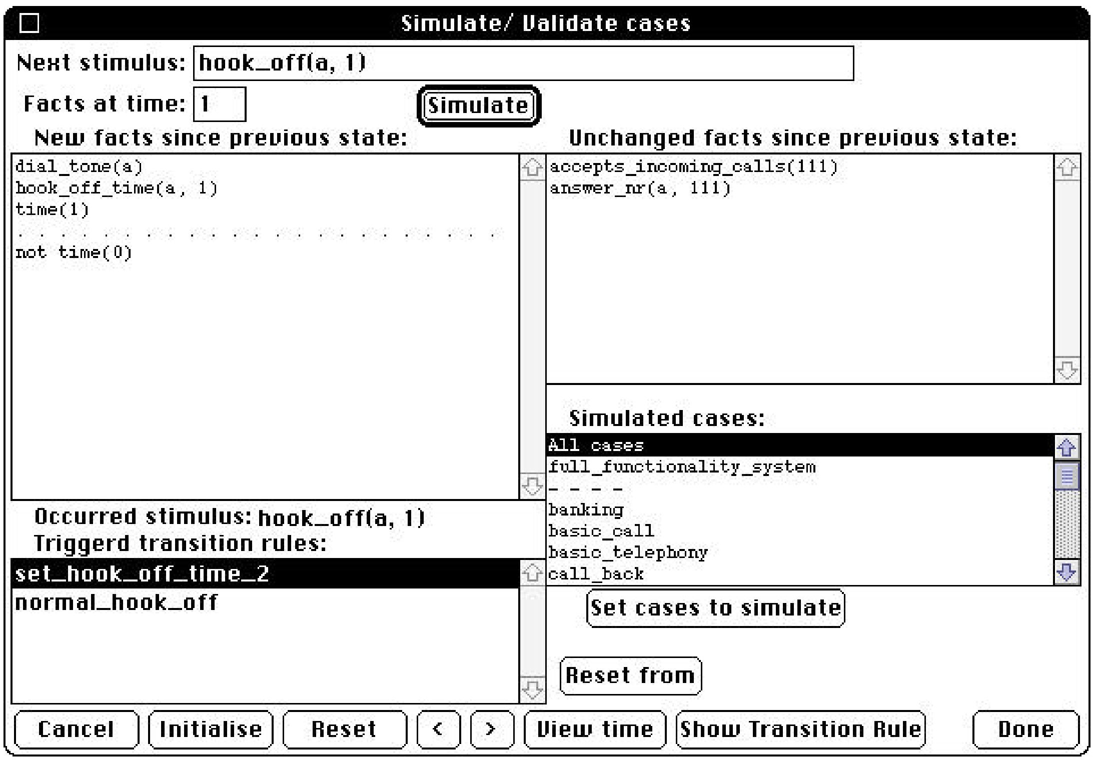
Executable specifications have lately become more popular and, in addition, for many non-executable formal notations, there is an ongoing research effort to identify executable subsets/extensions [Fuchs 92]. One of the main advantages of executable specifications is that the requirements designer can explore the specified behaviour (under different circumstances) by simulation. Executable specifications can be used as part of the communication about the system functionality between customers, system designers and programmers. The simulation allows an interactive exploration of the required functionality (the required dynamic behaviour) captured by the requirements specification. If any unexpected, unspecified or unwanted behaviour is encountered then the solution needs refinement: the requirements designer can refine, revise and/or extend the specification (as described earlier in this chapter and shown in Figure 7.1), so that it captures correctly the intended behaviour.

Since the requirements designers intention of the behaviour is not fully covered by the examples, and since the proposed solution includes more behaviour than explicitly exemplified in the input examples, the specification has to be validated. In CABS, we have implemented a basic text based simulation tool as shown in Figure 7.2. If simulation is to be used with customers of the system it would need to be improved and the logical notation better encapsulated. A graphical representation or simulation animation would be one way of further assisting understanding for people not skilled in formal notations [Hughes, Cooling, 91]. Some experiments in graphical and icon based representation for simulations and specifications have been performed in the domain of telecommunications services [Preifelt, Engstedt, 92].

In the simulation tool, the user can create an initial state (the *Initialise* button in Figure 7.2), give a sequence of stimuli to the simulator, and explore which transition rules have been triggered and what facts and responses are concluded. This gives the user a powerful tool with which to explore the behaviour of the formalised requirements. The user starts a simulation by initialising the facts. In Figure 7.2 one subscriber is answering calls to number 111, *answer_number(a, 111),* and calls to number 111 are accepted, *accepts_incoming_calls(111)* are the initial facts as shown in the top right field. The user gives a stimulus (which may be selected from a menu containing all valid stimuli) in the text field *Next stimulus:* at the top of Figure 7.2 and selects the *Simulate* button. The *New facts since previous state*, *Unchanged facts since previous state* and *Triggered transition rules* fields will be updated and show the state after the stimulus has occurred. If the user wishes to inspect why a transition rule has triggered, the user can select the button *Show Transition Rule* which shows the transition rule with variables replaced by actual values from the simulation. The user can also explore why a transition rule has not triggered by choosing the *Why Not* button[23], selecting a transition rule that will be shown with the conditions which have or have not been met. The field *Facts at time* shows the current time step: if the user has simulated a number of steps, the <, > or *View time* button can be used to traverse forward and backward in the simulation space (in this implementation, a new

[23] *Why Not* button and the corresponding functionality has not been implemented in the final simulator for CABS. Such a functionality is a minor extension and was implemented in an earlier versions of the simulator.

stimulus can only be given at the last time step, but it would be desirable if tree structured simulations could be built and a different simulation branch could be started from any simulation step). Before a simulation is started, the user has to decide with which cases the new behaviour should be simulated (only transition rules from these selected cases will be triggered by a stimulus). For telecommunications services [Funk, Raichman, 1990], it is often an advantage to first simulate a new case without other interacting cases initially, and once this behaviour has been validated and refined so that it covers the basic idea, additional cases can be explored. If the user wishes to reset a simulation from a particular step, the button *Reset from* is used. If the *Initialise* button is chosen, the current simulation is cleared and a new initialisation can be selected (either select from previously defined initialisation or define a new initialisation containing facts that are true at time step zero).



Next stimulus: hook_off(a, 1)

Facts at time: 1

New facts since previous state:

```
dial_tone(a)
hook_off_time(a, 1)
time(1)
. . . . . . . . . . . . . . . . . . . . . . . . .
not time(0)
```

Unchanged facts since previous state:

```
accepts_incoming_calls(111)
answer_nr(a, 111)
```

Simulated cases:

```
All cases
full_functionality_system
- - - -
banking
basic_call
basic_telephony
call_back
```

Occurred stimulus: hook_off(a, 1)

Triggerd transition rules:

```
set_hook_off_time_2
normal_hook_off
```

Figure 7.2: Example of simulation window in CABS

7.5.1 Revising a Solution

If missing behaviour which is part of the input examples is identified, then the proposed solution needs to be extended (by identifying matching transition rules for the links not covered by the solution or by refining the transition rules). If missing behaviour, which is not a part of the input examples, is identified and classified as relevant to include in the initial behavioural requirements, the input examples should be extended to include this behaviour. In the domain of telecommunications services, the number of behaviours to be captured in a specification may be so large that it is not feasible to make input examples for all behaviours, only for the more common and normal ones. Other more unusual situations and interactions[24] are captured by the formalised requirements (a refinement of the behavioural requirements towards a full specification).

If behaviour is added to the formalised requirements, but not included in the input examples, there is still a possibility to perform some verification, if the simulation traces are kept as test cases for later re-verification and to formally prove that any modifications/alterations to a case have not accidentally changed any of the previously captured behaviour represented by the simulation traces. Verifying modifications/alterations of cases is a major issue for telecommunications service providers since services are often modified for different markets and users, or altered to interact in a desirable way with new services. It is a well known fact that alterations are one of the main causes of errors. This risk of accidentally introduced errors is reduced if previous input examples and previously performed simulations are re-used to verify that none of these behaviours have been accidentally altered (see [Buchanan,

[24] If looking at a telecommunications service such as *call diversion* or *three party call*, it could be argued that the behaviour normally encountered by the phone user is the main issue for the top level requirements sketch. The more unusual situations should of course eventually be catered for, but this can be left for a later stage in the process, after the main behaviour of the new service has been validated, verified and approved for full implementation.

Shortliffe 84]). Storing simulations has not been implemented in CABS but is a trivial extension to the automatic verification described in Section 7.6.

The solution may be revised on the level of transition rules (I4 in Figure 7.1) by editing the transition rules in a traditional way until they capture the behaviour exemplified in the input examples (individual transition rules can be edited in the transition rule window, Figure 5.5). If transition rules are revised to capture the behaviour exemplified in the input examples, the solution can be verified as normal, as described in Section 7.6 (no extra verification with simulation traces as described previously is needed).

7.6 Automatic and Interactive Verification of Results

Validation of new cases can be done more or less systematically but as long as traditional methods for validation are used, there is no guarantee that all requirements exemplified in the input examples are captured in the formalised requirements. In CABS, a step of formal verification is added where the input examples are translated to test sequences (called test cases) that are used by the verification tool. This is done automatically and can prove that the behaviour exemplified in the input examples is captured in the case and its environment, i.e. all the other cases with which it is expected to coexist, and with which it may also interact or be dependent on. If behavioural examples outlining excluded behaviour have been given, these have to be proven not to be included in the behaviour (negative input examples have not been implemented in CABS but is a straightforward extension of the existing implementation). In CABS we have implemented this automatic verification for positive input examples. If a case does not capture some specific behaviour exemplified in links in the input examples, CABS will point out which behaviour in the input examples is missing from the formalised requirements. This indicates that the transition rules in the formalised requirements specification corresponding to these links fail to fulfil their task of capturing the exemplified behaviour. Hence, the verification has failed and the user has to refine the input example or add another input example in order to give more information, so that a transition rule meeting the requirements can be identified by the matching process or generated from the input examples.

Once a case and its transition rules have been altered, all cases that include this transition rule directly or indirectly need to be verified. Those cases which need to be re-verified can be determined automatically (which can be done without a search through all the transition rules).

By using a formal notation, we also have the possibility of identifying inconsistency in rule sets [Funk 93]. A program performing some consistency checks on rules has been implemented but not integrated in the CABS system (see Chapter 9).

7.6.1 Generating Test Cases from Input Examples

A test case is a sequence of triples of preconditions (facts and responses), stimulus and postconditions (facts and responses) that are expected to hold before and after the stimulus has occurred. The input examples are a set of links and nodes. The links contain conditions (both conditions from the originating node and additional conditions) and conclusions (both originating from the terminating node and additional conditions) which can be used directly to produce test sequences, containing sequences of stimuli, preconditions and conclusions that are expected to hold before/after the stimuli have been received. If a link has some additional conditions that are not a conclusion of some previous link or a part of any previous node, these terms can be added to the initial start situation if this option is selected. Input examples always have a finite number of nodes, so we only need to generate all possible routes between all the denoted start and end nodes. We do not need to expand loops since if we follow a branch of stimuli between start node and end node and encounter a node in the input example that has already been traversed, this branch needs no further exploration since each node has already had all its branches explored.

Once all branches for an input example have been expanded between start node and end node in the input examples, we have a number of test cases to verify. As well as using test cases, we may also show different properties, such as *liveness [Segala, Gawlick, Søgaard-Andersen, Lynch 98],* i.e. if a branch cannot reach an end node within a reasonable number of stimuli (for instance a phone user is only expected to do a reasonable number of actions resulting in stimuli, dialling, putting calls on hold, joining them into three party calls etc. which can be limited to a safe maximum number of stimuli), this can be identified.

7.6.2 Verifying a Test Case Against Formalised Requirements

The purpose of the verification is to verify (formally prove) that all the behaviour captured in the input examples is included in the formalised requirements and that the behaviour of negative input examples is excluded from the formalised requirements [Atkinson, Cunningham 1990].

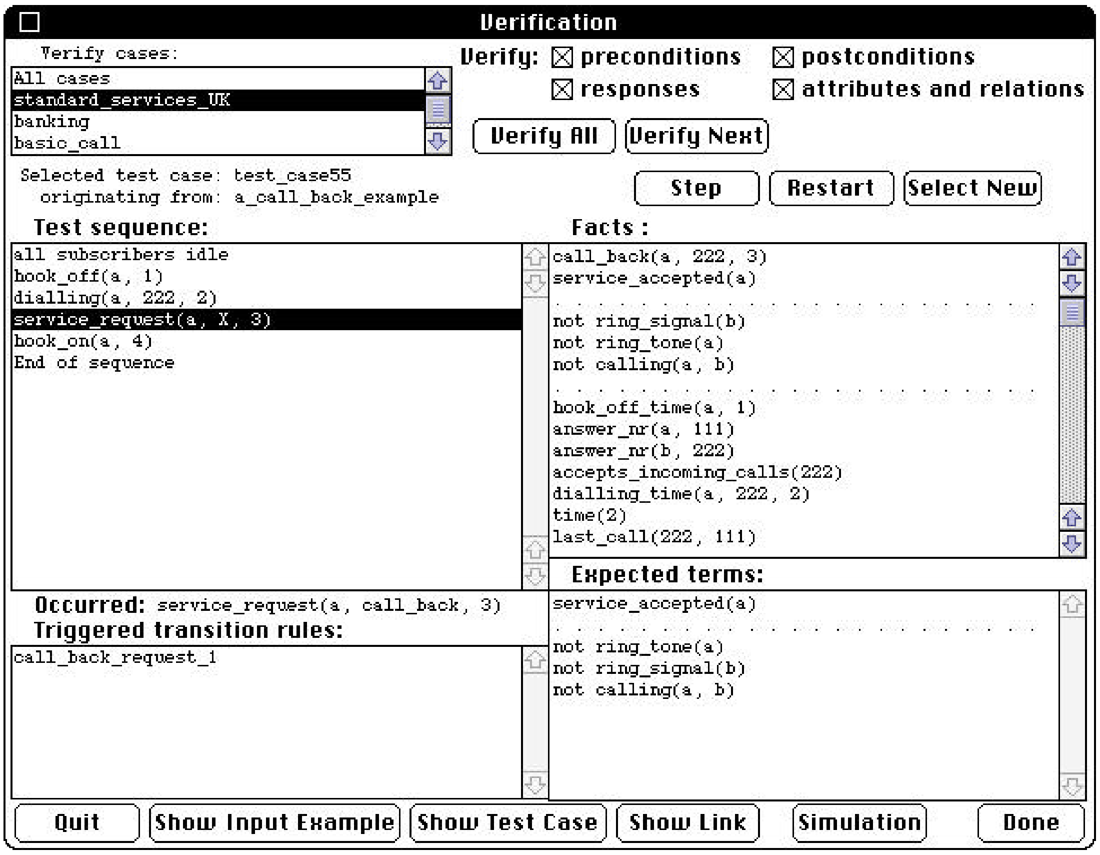
Definition of *included behaviour*: Given the same sequence of stimuli, the formalised requirements capture the behaviour of the input examples if and only if the formalised requirements exhibit a list of responses which can be mapped to the list of responses in the input examples: Note that there may be responses in the formalised requirements that are not present in the list of responses from the input example.

Definition of *excluded behaviour*: The formalised requirements exclude the behaviour of the input examples if and only if the formalised requirements *do not* exhibit the same responses, given the same sequence of stimuli as exemplified in the input examples.

In CABS, the requirements designer selects which cases or set of cases are to be verified by selecting from the list *Verify Cases* in Figure 7.3. If more cases are selected, interaction between these cases is also verified (if input examples exemplifying interaction between these exist). If the check box *precondition* is ticked, the verification will check that preconditions connected to stimuli in the test case are checked and any differences are reported. If the check box *postcondition* is ticked, the postconditions are checked in the same way. If the check box *response* is ticked, the externally visible response terms are checked (same response for same sequence of stimuli). If the check box *attributes and* *relations* is ticked, attributes and relations connected to stimuli in test sequences are checked. These settings may be useful if a verification fails because of differences between the exemplified behaviour of the input examples and the captured behaviour of the proposed solution and gives the user a tool that may be of help in the exploration of the differences. If the *Verify All* button is pressed, all existing test cases for the selected cases are verified (if the verification of a test case fails, the verification stops and the failing situation is shown in the *Verification* window). If the *Verify Next* button is pressed, the name of the next un-verified test case is shown in the *Verifying test case:* field. Test cases are always named after their originating input example name merged with a number (the number is the order number in which the test case was generated). If the requirements designer wishes to step through a test case, the *Step* button is pressed and one stimulus at a time from the stimulus list *Test sequence* is verified (the highlighted stimulus in the *Test sequence:* list is the last verified).

In the step mode, the result after every step is shown in the *Facts:* list, listing all the facts true in the state. What facts have been changed since the previous time are listed first. After the dotted line the facts that are not true any more are listed and finally after the second dotted line, all the facts that have not been changed since the previous step are listed. The *Expected terms:* list shows what the test case expects for terms in the state and the *Triggered transition rules:* list shows all the transition rules that have been triggered as a direct consequence of the stimulus. A discrepancy is an indication of a behavioural difference between the initial requirements and the formalised specification sketch. The user has to decide if the initial requirements have changed or if the formalised requirement sketch has to be revised. The *Restart* button is used to reset the current test case to its initial start state, which may be useful when stepping trough a test case. The *Select New* button allows the user to select and initialise the Verification window with another test case.

The verification uses the simulator in batch mode. This has the advantage that if any discrepancies are identified and the verification is halted, the *Simulate* button can be pressed and the last test case can be explored with the simulator (stepping forward/backward, resetting from a particular time and simulating different stimuli and their effects). The original graphical input example can also be viewed by pressing the *Show Input Example* button. The test case can be viewed by pressing the *Show Test Case* button. Each step in the *Test Sequence* list has a reference to its originating link in the graphical input example which can be viewed by pressing the *Show Link* button.



```
All cases
standard_services_UK
banking
basic_call

Selected test case: test_case55
  originating from: a_call_back_example
```

Test sequence:

```
all subscribers idle
hook_off(a, 1)
dialling(a, 222, 2)
service_request(a, X, 3)
hook_on(a, 4)
End of sequence
```

Facts :

```
call_back(a, 222, 3)
service_accepted(a)
. . . . . . . . . . . . . . . . . . . . . .
not ring_signal(b)
not ring_tone(a)
not calling(a, b)
. . . . . . . . . . . . . . . . . . . . . .
hook_off_time(a, 1)
answer_nr(a, 111)
answer_nr(b, 222)
accepts_incoming_calls(222)
dialling_time(a, 222, 2)
time(2)
last_call(222, 111)
```

Occurred: service_request(a, call_back, 3)

Triggered transition rules:

```
call_back_request_1
```

Expected terms:

```
service_accepted(a)
. . . . . . . . . . . . . . . . . . . . . .
not ring_tone(a)
not ring_signal(b)
not calling(a, b)
```

Figure 7.3: Example of verification window in CABS

The verification also handles test cases where variables are used. In Figure 7.3 in the *Test sequence* list, the third step, *service_request(a, X, 3)*, can under the given restrictions (preconditions and postconditions), only be equal to *service_request(a, call_back, 3)* as shown after *Occurred*. If the variable causes indeterminism and the variables can be instantiated to different values, the user has to make a selection to make the test case valid.

7.7 Revising and Refining the Solution

A solution may be directly modified by editing transition rules. This does not conflict with the methodology of CABS since verification and, most likely, validation has to be performed before the task can be considered complete. The verification ensures that the solution still conforms to the input examples. If the verification is unsuccessful, the question to explore is if the input examples or the formalised requirement specification has to be modified. Once the original idea has been formalised, validated and verified, the solution includes the behaviour of the input examples. If the input examples reflect the behaviour of the new functionality, then the solution meets the original requirements.

If the proposed solution needs some revision (such as adding in the behaviour for unusual situations), or if there is no single case that meets the user's requirements, a more traditional approach of editing transition rules may be necessary. This requires knowledge of production systems and rule based approaches.

Chapter:

8. Evaluation of CABS

As mentioned in Chapter 2, there are hundreds of different telephone services implemented by modern telephone networks. These exist in different variations where adaptations have been made for different countries, companies and telephone operators. The CABS case library contains seventeen telecommunications services (127 transition rules, 54 terms), reflecting a variety of different types of telephone services commonly supplied to phone users which are often used in experiments and research involving service specifications [Funk, Raichman, 1990]. The case library selected for the evaluation contains the following services[25]: basic call; call barring; call diversion; call waiting; call reminder; call back; call return; charge advice; emergency call; three-way calling; pick-up call; banking; voting; queue calls; caller display; basic telephony.

For case-based reasoning, there are a number of key issues to be evaluated (described in Section 8.1). The most desirable approach for an evaluation is when a set of objectively-measurable criteria can be defined and proven: for example, if the aim of a research project is to apply an approach enabling micro-processors which are ten times faster, compared with

[25] For details on some of the services, see for example BT's brochure "Welcome to Selected Services, Your User Guide".

currently available technology, a prototype that meets this criterion is clear evidence that the claims of the research hold. In the area of mathematics, a precise answer may be a mathematical formula or proof. In artificial intelligence and knowledge based systems, where different areas and approaches are combined and integrated to achieve the desired results, an empirical approach to evaluation is usually the preferable choice [Mark, Greyer, 93].

An important question is: with what data should an evaluation be carried out. For the case library, a set of services is chosen that is commonly used in experiments with telephone services [Funk, Raichman, 90], [Klusener, Vlijmen, Waveren, 93]. For these services, input examples were created in the same way in which end users are expected to use the system. These are used to evaluate the robustness of the system, and the results reported give an indication of how well it meets its claims (identifying similar behaviour and verifying the solution against the input examples). The results are reported in the tables of the following sections.

The decision was taken that end user evaluation was not appropriate, for two main reasons. Firstly, real end users are not accessible; telephone services designers are in great demand, and they would not grant time for the evaluation of CABS. The second reason is that since the implementation is fairly large, any results from an end user evaluation would be questionable as it may be difficult to separate the evaluation of the prototype (an end user may like or dislike a particular implementation depending on background knowledge, experience and personal preferences) from the evaluation of the general approach.

8.1 Issues to Evaluate in Case-Based Retrieval

The success or failure of case-based reasoning systems depends on five key issues listed in Table 8.1, each with a brief reference to CABS. They are in no particular order and are extended and adapted from [O'Leary 93] and [Ketler 93]:

How easy is it to use the system (giving input examples on a suitable abstraction level).
CABS uses graphical input examples. Graphic notations are common in telecommunications

applications and the notation used is considerably less complex (due to a reduction in expressiveness) than notations traditionally used (SDL, MSC, CP, etc.). To evaluate the notation is beyond the scope of this research and the view is taken that the notation should be adapted and tailored to meet the user's wishes.

Consistency and uniformity of knowledge representation (sufficient for all involved parties and also enabling automated verification, adaptation, etc.).
CABS uses a predicate logic notation based on Horn-clauses.

Clustering of cases (application domain feature).
Telecommunications services, and in particular telephone services, are on a behavioural level often similar to each other. Different countries and service providers offer similar, but not identical, services to telephone users. Re-use is high on the agenda in telecommunications.

Metrics for the retrieval of cases.
A set of structural features, based on an analysis of the semantics, is used to identify and retrieve cases capturing similar behaviour.

Assessment of the solution produced by the system.
CABS uses input examples to verify solutions. Simulation is used to explore behaviour not covered by the input examples. Theorem proving is a further extension (partly implemented but not integrated in the prototype system, see Chapter 9).

Table 8.1: The five main issues to be evaluated

This research focuses on the identification of similar behaviour for re-use and to confirm that the final solution captures the behaviour exemplified in the input examples, so issues 4 and 5 in Table 8.1 are the main issues in this evaluation and will be explored in depth in sections 8.2 and 8.3.

To evaluate issue one to three is beyond the scope of this research but they are discussed briefly because they are of relevance if a full scale implementation of a system based on the CABS approach is considered:

Issue 1 (Table 8.1): The behaviour imagined by the user has to be expressed in some notation as input examples, in CABS. To use a graphical notation is an obvious choice for the domain of telecommunications since graphical notations are often used in this application domain for a variety for different purposes. CABS has a very basic graphical representation (the notation should be adapted to the user's needs and also for different application domains. This is beyond the scope of this research.). The main requirement for the input examples is that it should be possible to translate them into transition rules used for matching and for generating test cases used in the verification. Whether the input examples capture the desired behaviour correctly can only be assessed by the designers, making evaluation of the problem description difficult (especially without access to end users).

Issue 2 (Table 8.1): For a number of reasons (convenience being one of them), CABS uses a subset of predicate logic extended with a frame axiom as its knowledge representation language. With this simple but sufficiently expressive predicate logic, the implementation of matching, simulation, verification and translation from input examples to transition rules is realised with reasonable effort. Translation to and from natural language has also been explored for a notation similar to the one used [Dalianis 95].

Issue 3 (Table 8.1): The application domain of telephone services has the features needed to make re-use beneficial since similarities between services are common in telecommunications. Re-use is considered an important matter, and is high on the priority list for service development. Since new telephone services are designed and implemented all over the world in different company branches, companies and service vendors, it is assumed that a lot of work is repeated and that there is a large potential for re-use. Effort to standardise service independent building blocks has been undertaken by the international telecommunications union but this will not lead to standardised services (as discussed in section 2.3). Section 8.4 shows that CABS has the capability to considerably reduce repetitive work by identifying similar services.

8.2 Evaluation of Retrieval and Solution Assessment

Figure 8.1 gives an *evaluation view* of CABS (the large box) and the two main issues: (i) identifying and retrieving similar behaviour for re-use (issue 4 in Table 8.1) and (ii) verifying the proposed and selected solution against the input examples (issue 5 in Table 8.1). In the telecommunications service domain, CABS is not expected to find a case in the case library exactly meeting the exemplified behaviour in the input examples since it is unlikely that the user would give an example of a behaviour that exactly matched a case in the case library (When this occurs, either the service is uncomplicated or the user knows exactly how the service behaves). CABS proposes a list of similar cases that are candidates for the behaviour expressed in the input examples. The requirements designer makes the final selection, eventually changing the initial idea of the behaviour exemplified (changing input examples or accepting input examples belonging to the case). The overall question to evaluate is whether or not the matching heuristics are practically useful and produce a set of similar cases, which is small enough to be manageable, yet broad enough not to miss relevant cases[26]. If we know the solution case for a set of input examples, we can find out how well the features used by CABS work to identify the solution. At the same time, it would not be desirable if the matching only gave the single most expected case as a solution, since a case capturing exactly this behaviour need not necessarily be the solution sought (a requirements designer may revise and extend the behavioural ideas). Therefore, a set of similar cases

[26] Even so, similarity-matching may not, in a fully functioning system, be the only approach to identifying relevant cases: keyword matching, text-based matching on informal descriptions of cases, and matching new input examples against input examples stored with cases in the case library are some interesting extensions to CABS.

where the most similar solution has a high ranking is preferable. In Section 8.3, the input examples are selected and matched, and the results are summarised and their implications discussed.

Another central feature of CABS is to *verify proposed and selected solutions* (see Figure 8.1). The matching process should purposely give a set of more or less similar cases from which the user can select the one(s) they want. The verification, on the other hand, should confirm that the behaviour exemplified in the input examples is included in the selected solution and if not, describe where it differs. If it does differ, the requirements designer has to explore why this is so. In Section 8.5, proposed and selected solutions are verified against the input examples.

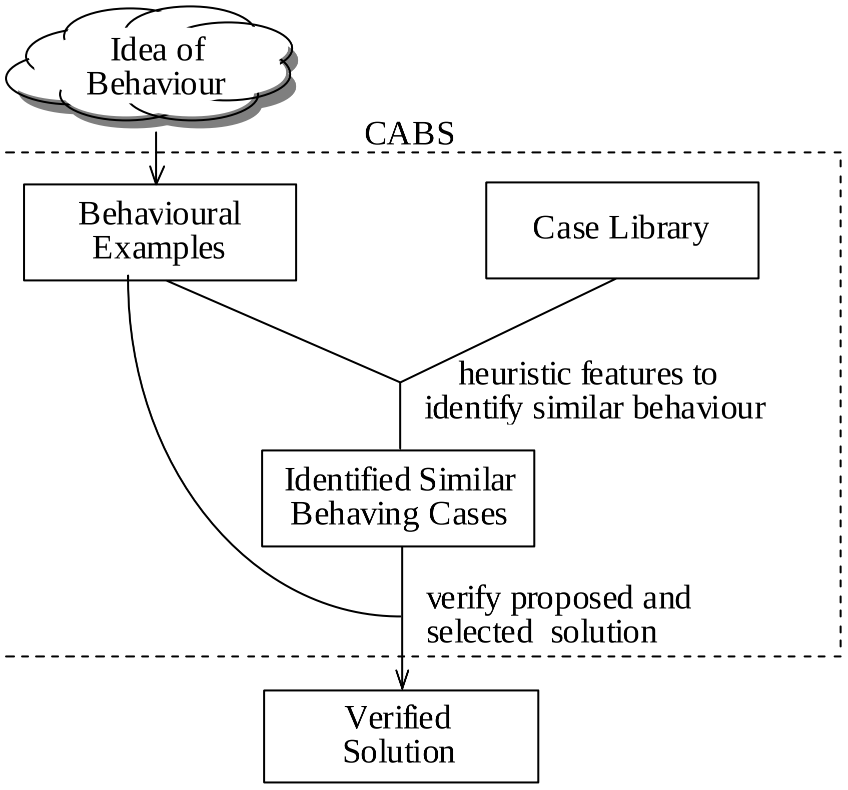

Idea of Behaviour
CABS
Behavioural Examples
Case Library
heuristic features to identify similar behaviour
Identified Similar Behaving Cases
verify proposed and selected solution
Verified Solution

Figure 8.1: A verification view of CABS.

8.3 Selection of Input Examples and Target Cases

As mentioned erlier the set of cases that are stored in the case library are commonly used in experiments with telephone services [Funk, Raichman, 90], [Klusener, Vlijmen, Waveren, 93]. For all cases in the case library, one input example, giving an example of the behaviour of the corresponding service, was designed. An effort has been made to produce input examples which are similar to those a requirements designer might give, without knowledge of the behaviour of any service implementing the exemplified behaviour. This is fairly easy to achieve, as there is often little choice in how to exemplify a particular behaviour with an input example. A good illustration of this is basic_example_0 (Figure 8.2) which contains four nodes: *all subscribers idle; dial tone a; a calling b; in speech*. The node *dial tone a* has the condition *dial_tone(a)* and the node *a calling b* has the condition *calling(a, b) & ring_tone(a) & ring_signal(b)*. The nodes are connected with the links illustrating the actions the telephone users can make. This is sufficient for the matching algorithm to identify *basic_call* as the best matching case (for matching results see Table 8.3). Different requirements designers would most likely express the same behaviour in a similar way with the given set of nodes.

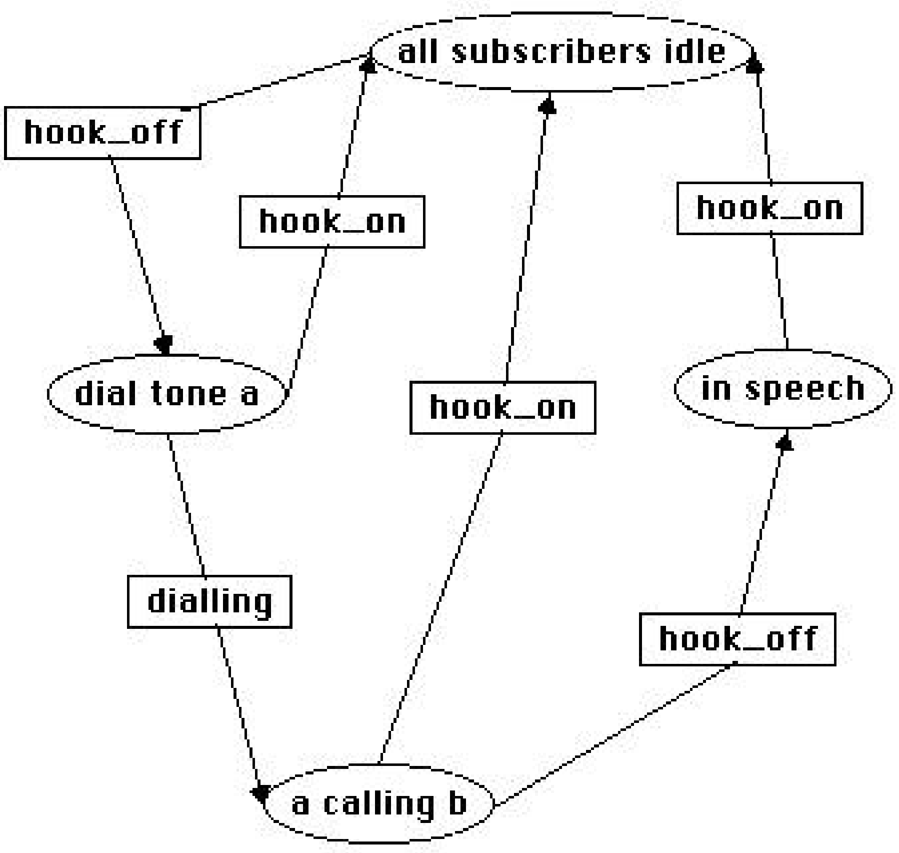


Figure 8.2: input example basic_example_0

In Table 8.2, the names of the input examples are given with the corresponding target case (telephone service). Appendix B lists all the cases in the case library and Appendix D gives all the input examples used for the evaluation (as listed in Table 8.2).

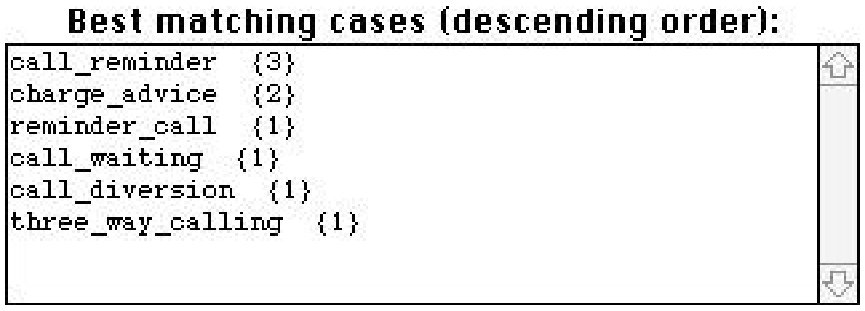
Input example	Case in Case Library
a_banking_example	banking
a_barring_example	call_barring
a_basic _behaviour_example_0	basic_telephony
a_basic _behaviour_example_1	basic_telephony
a_basic_example_0	basic_call

a_basic_example_1	basic_call
a_busy_example	basic_call
a_call_back_example	call_back
a_call_last_caller	call_back
a_call_reminder_example	call_reminder
a_call_return_example	call_return
a_call_waiting_example	call_waiting
a_charge_advice_example	charge_advice
a_divert_example	call_diversion
a_multi_call_example	tree_way_calling
a_pick_up_call_example	pick_up_call
a_queue_example	queue_calls
a_show_number_example	caller_display
a_voting_example	voting
a_wake_up_call	reminder_call
an_emergency_example	emergency_call

Table 8.2: Input examples and target cases

8.4 Evaluation of the Matching Algorithm

Each input example targeting the same case has been used for evaluating the matching algorithm. Test cases are all defined as being dependent on the basic_call service and basic telephony service (except input examples describing basic call and basic telephony), so these services are not considered as a solution and are excluded from the matching result. The parameters for matching transition rules and cases have been left at their default values. In Figure 8.3 the matching result for the input example a_call_reminder_example is shown (for an example on a full matching result, see Figure 6.9).



```
Best matching cases (descending order):
call_reminder  {3}
charge_advice  {2}
reminder_call  {1}
call_waiting  {1}
call_diversion  {1}
three_way_calling  {1}
```

Figure 8.3: Match result for input example *a_call_reminder_example*

The column *Best matching cases* in Table 8.3 contains the matching result for each input example. The result from Figure 8.3 is shown as a list with numbers {<u>3</u>21111} in Table 8.3. After the results list a number is shown (/6) with the number of links the match is based on. Since we know the solution case for the match, the number representing the best case is underscored. Cases that have the same ranking are not ordered in any way.

This rating is actually quite crude; if a more precise ranking is needed for a large case library, it could be refined by taking the individual scores of transition rules into account when accumulating the total score for a case, rather than counting the sum of the number of matching transition rules. The crudeness of the ranking cannot alter the set of proposed services, but in some cases causes results in two or more cases being ranked equal highest. Since the requirements designer makes the final selection among all proposed services and the total number of services were manageable, their ranking seemed to be a good enough guide for the final selection, and a more discriminating algorithm was not implemented.

If there is only one best match then the matching process has led the user directly to the solution. If the number is greater than one, then there are several cases in the case library which share characteristics with the input example. As explored in detail in Chapter 7, the requirements designer is expected to handle this situation (by adding more input examples, excluding links from the input example, exploring and selecting the most appropriate case, combining more than one case, etc.). Excluding links from the input example is an easy approach to improve a matching result if it is obvious that the best proposed cases are not acceptable. This can be done directly from the detail window for links, by selecting *Link not relevant for match* in *Match select for link:* (see Figure 4.6). One should bear in mind that excluding links will not extend the search (the same or fewer cases are proposed as a solution) and will only be useful if the solution case is within the list of proposed solutions. In the column *Excluded links* (Table 8.3), some links which are obviously not relevant for the match have been excluded from the match and the match has been re-done. The number of links used in the match is given as a number in the same way as in the column *Best matching cases;* the number of links will obviously always be less since links have been excluded from the match.

If the total number of proposed cases which scored higher than one is too high the requirements designer may increase/reduce appropriate matching parameters. If a service has few characteristic features, it is expected that this total will be large, whereas if the service is very specific in its behaviour, there will be fewer cases. No matching parameters have been altered during the evaluation presented in Table 8.3 (every transition rule scoring higher than 10 is counted as a match).

Some matching results clearly point out the solution, for example match 6 in Table 8.3. Case 17, where 10 proposals are ranked, has two proposals ranked highest and this match is regarded as having a weak focus towards the solution. If the focus is weak the input example (and the service) may be of more general character and share features with many other services.

Input example cases		Best matching links	Excluded
1.	a_banking_example	{**2**111111}/4	v
2.	a_barring_example	{**1**11}/3	v
3.	a_basic_behaviour_example_0	{**4**33311}/4	v
4.	a_basic_behaviour_example_1	{**2**111}/3	v
5.	a_basic_example_0	{**6**211111}/6	v
6.	a_basic_example_1	{**7**211111}/7	v
7.	a_busy_example	{**2**11111111}/2	v
8.	a_call_back_example	{**2**2211}/6	v
9.	a_call_last_caller	{**1**111}/4	4 {**1**}/1
10.	a_call_reminder_example	{**3**21111}/6	v
11.	a_call_return_example	{**3**2211111}/8	v
12.	a_call_waiting_example	{**2**11}/5	v
13.	a_charge_advice_example	{3**2**2111111}/7	nb {**2**2111}/4
14.	a_divert_example	{**2**221111111}/7	v
16.	a_multi_call_example	{**3**21}/6	v
15.	a_pick_up_call_example	{**1**1111}/5	5 {**1**}/1
17.	a_queue_example	{**2**211111111}/6	v
18.	a_show_number_example	{**1**}/3	v
19.	a_voting_example	{**1**111111}/3	7 {**1**111111}/1
20.	a_wake_up_example	{**2**1111}/3	v
21.	an_emergency_example	{22**1**11}/6	nb {**1**11}/1

Table 8.3: Match result for input examples

The input example, a_basic_example_0, is in fact faulty because a node is missing[27], hence one link is missing and one is faulty. It is interesting that the solution case was identified in spite of this mistake. This result was unexpected, but on analysing the result it becomes clear that this is exactly one of the desired benefits of case-based reasoning compared with other more precise approaches (e.g. some logical proof of equivalence). Input examples may lack details or even be partly faulty, but if the heuristics for the match (the features used) are well chosen, the matching algorithm should be robust enough to identify relevant solutions based on the part of the input example which is not faulty. During the evaluation, a more obscure fault was identified in the matching (if matching transition rules had constants in their stimulus part, variables were accidentally bound in stimulus terms with these constants). Coincidentally, this problem only caused the matching algorithm to miss the correct solution in one example and after correcting this problem the four input examples got one additional proposed case.

8.4.1 Over-Diffuse Identification of Solution

For all input examples used in the evaluation, the solution case is amongst the proposed solutions, but in two cases (13, 21) the correct solution case was not amongst the highest ranked, and in two cases (9, 19), more than three proposals where ranked highest. Before analysing these cases, a brief summary of how such a result may be tackled by the requirements designer is given. If a requirements designer does not find an appropriate case among the proposed cases, one of their first actions is to refine the input examples (as described in Chapter 7), either by supplying more input examples or refining those already given. One way of refining input examples is to label links as not directly being a part of the

[27] When two users are talking to each other and one of them puts the receiver down, the other user will have silence until their receiver is also put down, the input example makes both the caller and the called person idle when one person puts the receiver down, this is not true since the person who did not put down the receiver cannot receive a call or lift the receiver *(hook_off)*.

behaviour sought for in the case library, which as shown below, often gives a better matching result. For example, in the service *charge_advice*, everything in the input example 13 (Table 8.3) up to telephone user *a* talking to telephone user *b* (for input example see Appendix C) is a normal call, but the matching process does not know that and should still identify similar services to propose for this part of the input example; this may misdirect the search in some situations or result in a less focused proposal, depending on how large a part of the input example is part of the target case. If these links (up to node *speech*) are marked as being irrelevant for the search[28], the search focuses on the part in which the requirements designer is interested. For input example 13 this brings the correct service (charge_advice) to the top of the ranking list (shared with *call_reminder* which could be classified as having a similar behaviour to the example[29]); before this selection of links *charge_advice* was ranked to be amongst the second best proposals. The re-match result is shown in the column *Excluded links* in Table 8.3.

After the requirements designer has excluded selected links in the input example, example 19 still shares the solution with other proposals which may be considered as a weak focus on the solution, but when inspecting the matching result of the link, the highest ranked transition rule belongs to the service *voting*, hence the service *voting* is correctly identified as the best match (a list with proposed and ranked similar transition rules can be viewed in the detail link window, see Section 4.2.1). This shows that the link/transition rule matching is able to correctly rank the transition rule from the solution case as the highest. This information is not carried forward when ranking cases in the case library due to the crude approach of counting the number of matching links for each case. Also, for input example 21, the solution would

[28] The links are still relevant when verifying the behaviour.

[29] Reminder_call may even have parts that could have been re-used to create a new service charge_advice if such a service had not existed in the case library. No analyses of this possible re-use has been explored.

have been ranked the highest if the link/transition rule ranking had been carried forward to the ranking of the cases. Hence the ranking of cases would benefit from receiving and using more information from the link/transition rule match. Using more information from the link/transition rule match when calculating the overall score for matching cases is considered a minor alteration. This would further improve the matching results, especially if the matching result is based on a few links from the input example. It would add a few calculations to each ranked case in the case library which would be negligible with other calculations performed for each transition rule and case (for more on time efficiency of matching algorithm see Section 8.7).

8.4.2 Conclusions for Match Evaluation

For all input examples given, CABS was able to identify the corresponding solution amongst the highest ranked proposals and for 14 (out of 21) input examples, it ranked the solution as the best proposal. In 19 (out of 21) input examples, the solution was amongst the three highest ranked proposals. When it did not rank the solution amongst the highest, excluding irrelevant links in the input example, it put the solution case amongst the highest ranked, but for input example 19, seven other suggestions were ranked at the same level. This is sufficient in the case library used for the evaluation, but may give the requirements designer too many cases to select from in a large case library. By using more information from the links/transition rule match when ranking, cases from the case library would help in the identification of the best solution. In the input examples, we purposely avoided using solution specific terms since, in a larger case library, the user may not always be able to identify and chose these terms. For example, the service *voting* has a term *vote_counter(VoteNumber,TotalVotes)* used as a counter and the service *call_diversion* has a term *redirect(FromNumber, ToNumber)*. These terms were purposely not used in the input examples in order to simulate a less knowledgeable service designer. It may be argued that a more experienced service designer, when designing input examples and selecting from a list of 52 terms, may select one of these terms. This would focus the search considerable (but not necessarily exclude a solution not containing these terms), and improve the matching result.

This result is sufficient to enable a requirements designer to identify the corresponding case in the case library. If this was the hit-rate in a full-scale system, it would be very good, since if this represented all services that would have been fully specified, evaluated, verified, integrated with other telephone services and implemented[30], a large amount of work would have been saved.

In some cases it would be beneficial to provide the designer with both similarity matching and some additional matching approaches, for example keyword matching. Keyword matching would in many situations be less accurate and miss possible solutions when compared with similarity matching, but it may be able to focus the search, especially in small case libraries, since it is more likely that there are specific terms unique for a particular service. If an experienced requirements designer can identify the terms discriminating the solution service from other services, the service would be found with keyword matching (in telecommunications services this is less common since many services do not introduce new terms even if they were, they may not always be easy to guess, even with access to all term definitions). As mentioned earlier, a restrictive attitude towards using terms discriminating a solution from other cases was adopted when producing input examples for the evaluation. Also CABS is not dependent on cases having particular keywords discriminating them from other services since the matching is bases on a careful analysis of the semantics of transition rules, translated to a number of syntactic features.

A relevant question is what happens if the matching cannot identify a suitable case if there is no similar case (a new type of service not yet specified and implemented) in the case library.

[30] Implemented in a way where all references between requirements, specification, design and implementation are kept, and where the design and implementation is structured in a way that re-use is enabled (for example an object oriented approach).

Some case-based reasoning approaches cannot handle such a situation. In CABS, input examples are translated to transition rules which are not expected to contain all details, interactions etc. These *input transition rules* can be used by the requirements designer as a starting point for the new service and the input example may be refined and extended to generate transition rules closer to what is needed for the new service. Hence, the approach does not falter if there are no suitable cases in the case library or if the requirements designer (with the help of CABS) fails to identify a suitable case in the case library.

8.5 Evaluation of Automatic Verification

All the input examples that describe a full behavioural example from a start node to an end node have been used to produce test cases (for consistency, all test cases are listed in Table 8.4). Cases marked with "-" in the *Generated Test Cases* column in Table 8.4 have input examples not including a start node and end node or are not detailed enough to generate test cases. If a test case does not include a start and end node, it may just be a fragment of some required behaviour which may be sufficient to identify a matching case or it may be an addition to other input examples (7b, a_busy_example in Table 8.4 is an addition to 7, a_basic_example_1, so, it is not sufficient on its own to generate test cases, but generates test cases in conjunction with a_basic_example_1). If the requirements designer accepts a match, all input examples belonging to the search should be used to generate test cases and these should pass the automated verification before the solution is accepted. The verification process of test cases do not accept differences as the matching does and will therefore identify possible problems. In those test sequences used, the test sequences identified problems both in the input examples and in the solution case. After correcting these, the input example will pass. The input examples identified one ore more of the following problems (no particular order):

a) Variables were used in input examples that might cause unwanted indeterminism. Refining input examples by changing variables to constants makes them more specific. Verification can handle variables in stimuli if there is only one variable binding possible (no indeterminism).

b) Faults/misunderstandings in the input examples were identified. An input example may contain faults and misunderstandings (as in input example 5 discussed in Section 8.4.1) yet still be sufficient for identification of an acceptable solution case in the case library. Test cases produced from such an input example should not pass the verification and the input example should be refined to reflect the factual requirements.

c) Conditions to links that have not been used anywhere else (in nodes/links) in an input example may not be determinable when testing a test case. If additional conditions are consistent they may, by default, be added to the start node (this option has been implemented in CABS), but if they are not consistent, no test cases are produced and the input example needs refinement.

d) Missing facts for transition rules expected to trigger: If during verification, a particular transition rule, which is expected to trigger has some preconditions that have not been mentioned in any node or link in the input example, then these preconditions will also be missing in the test case and this transition rule cannot trigger. This can be resolved by adding these facts either to the corresponding link (additional conditions) or to the start node (or any other appropriate node) in the input example.

e) Identified faults in the case library: If the input example is correct and the cases tested do not pass, then the cases are not correct. The requirements designer has the choice of either modifying the matching service or making a new variation of it which meets the current requirements. If the difference is small, much of the proposed case failing the verification may be re-used.

Most of the generated test sequences identified some problems, showing that the approach of using test cases to recognise potential problems is helpful. Services specified and stored in the case library for the evaluation were assumed to be functioning properly based on simulation during the development. Even so, a number of problems were identified when verifying test cases. This shows that during the development of new services (not previously specified and stored in the case library), the use of test cases will be useful. Test cases are also valuable when new services are integrated with previous existing services (added value services such as *call_waiting* and *three_way_calling* have much interaction). Also, if a new service accidentally alters some of the behaviour of a previously formalised service, this will be identified by the test cases if the previous unaltered behaviour that has accidentally been changed is included in the input examples/test cases. If test cases identified problems, the necessary corrections in the input examples or cases in the case library have to be carried out by the requirements designer until the test cases pass. This correction/refinement was carried out for some of the input examples and cases during the evaluation, but not for all of them, since this effort does not contribute to the evaluation itself. Problems of class a, c and d are all classified as refinements of the input examples and are often trivial (less than twenty minutes work for most input examples).

Input example		Generated test cases	Correctly identifying problems(a-e)/passed(p)
1.	a_banking_example	1	a,d
2.	a_barring_example	1	p
3.	a_basic_behaviour_example_0	1	p
4.	a_basic_behaviour_example_1	1	p
5.	a_basic_example_0	3	b
6.	a_basic_example_1	3	p
7.	a_busy_example	-	-
7b.	6 & 7	5	p
8.	a_call_back_example	2	b
9.	a_call_last_caller	1	b,e
10.	a_call_reminder_example	2	b
11.	a_call_return_example	2	b

12.	a_call_waiting_example	1	e
13.	a_charge_advice_example	1	a,b,e
14.	a_divert_example	2	b,e
15.	a_multi_call_example	1	e
16.	a_pickup_call_example	1	e
17.	a_queue_example	1	e
18.	a_show_number_example	1	b
19.	a_voting_example	1	e
20.	a_wake_up_example	1	b,d
21.	an_emergency_example	1	d,e

Table 8.4: Generated test cases and their success rate

8.5.1 Reducing the Need for Refinement

Refinements of type a, c and d may prevent test cases from passing even if there is potential for the test case to pass. The effort required from a user in refining these by replacing variables with constants and including necessary facts to start node/conclusions could be reduced when generating test cases. This is possible because when the test cases are produced and verified, the user has selected a solution case. This information can be used to refine the input examples and fill in missing details or make over-generalisation specific enough to produce test case which less or no need for refinements of type a, c and d.

Refinements of class *a* always originate from the use of variables in input examples. In most cases it is obvious what terms should be for a stimulus, such as user *x* lifting the receiver at time 1, *off_hook(x,1)*. Time variables do not need to be given since these can easily be determined when generating test cases. A user may exemplify how a service is invoked in a particular situation by adding a link between two nodes, *service_request(x,Service,T)*. The requirements designer cannot know the name or code for the service since it is either a new service or it is unknown which of the services it is in the case library, before matching.

However once a solution proposal has been accepted, the service is known to be *transfer_call* and so CABS could instantiate these variables and generate a test case with less variables (CABS can handle variables if there is only one choice, then during the verification of the test case, the variable is instantiated to the only possible value).

Refinements of class c and d are often required because of missing facts in the initial state. In the input examples, nodes are a conjunction of facts that are required to be true, and the node denotes all states that have these facts true. When generating a test case, a proper start state is required. Since nodes are expected to be predefined (often by some more experienced requirements designer) and it is expected that input examples can be created by selecting nodes from a set provided, the start node can often be (and is for the evaluation) so well defined that it can be used as a start state for verifying test cases. If each case in the case library has a proposed start state (or required facts for any a start state), for simulation and testing, this could be merged with facts in the start node in the input example. If there are contradictions it may be relevant to report these. Some variables that have been used in terms occurring in nodes and links in the input example could be bound to constants and missing facts could be added, reducing the need of refinements of type a, c and d.

8.5.2 Conclusions for Verification

Generating test cases from input examples to verify that the behaviour of the test cases are included in the solution, has been shown to enable the user to improve the standard of the input examples and of cases that are under development. In most situations, refinement of input examples is trivial and was usually achieved by adding (or removing) a term in a node or link. The value of these automatically generated test cases is also obvious if changes are made in requirements or when new services are integrated with other services, since all previous test cases can be re-evaluated in order to confirm that no accidental change of behaviour in other services has occurred by integrating a new service in a communication system. This is a major issue in any specification of a large system that is modified and extended.

8.6 Summary of Evaluation Results

CABS can, using input examples, identify similar cases and also use the input examples to identify differences between the behaviour outlined in the input examples and the selected case. An improvement to the ranking of cases with the same number of matching links is proposed: by using the ranking from the matching link and the transition rules, the ranking for each case would better reflect the link/transition rule match (this is a small extension). If there is no matching case in the case library for an input example, the input examples can be used as a starting point to construct a new case which is most likely to be more efficient than formalising a service from scratch (although no tests have been carried out on this). The approach is also robust because it is not necessary for the solution to be the highest ranked as the requirements designer can make the final selection from the proposed solutions. The test cases generated from the input examples identified problems in both the input examples and the cases in the case library, and so they proved to be of use.

8.7 Computational Time for the Match

One of the advantages of the CABS approach is that it has a fast matching algorithm enabling the identification of similar behaviour. The matching is performed in two steps: firstly all links in the input example are matched against all transition rules and then all cases are ranked by inspecting their transition rules matching result and by making a ranking of each case in the case library. It is expected that a common size of an input example contains 5-8 links. As described earlier matching of each link is based on comparing sets of terms. The computational time used for this is linear in the size of the sets. In the current case library the number of terms in a transition rule is between 5 and 35 terms and a link has between 5 and 15 terms. Once the matching result is calculated, it is stored with the link (a ranked list of the best matching transition rules for the link).

Once all transition rules have been matched against all links, each case is matched against the input example. This is done by taking all transition rules belonging to a case and giving the case a numerical value representing the number of transition rules that match with any link from the input example. Hence matching and ranking all cases is a linear algorithm and directly proportional to the number of transition rules in the case, the number of cases in the case library and the number of links in the input examples. This enables an implementation of a very fast matching algorithm. For a realistically large case library, containing some hundred cases and some thousand transition rules with an optimised implementation of the matching algorithm, the response time, for matching an input example of normal size (5-8 links), could be guaranteed to be below a second. Some time measurements where the time scale is irrelevant[31] ensures that the prototype implementation performance is in accordance with the matching algorithm (see Figure 8.4).

[31] The implementation is made in an interpreted Prolog. Implementation has been made with no efficiency considerations and an object oriented layer that at least triples each access time to links, cases and transition rules has been used. The Prolog used is written for the 68000 processor emulated on a PowerPC. External interface to C++, efficient data storage available in LPAProlog and partial compilation mode (this requires declaration of what parts of the program are static and what parts are dynamic, which would take considerable time in a prototype system often changed and modified) have not been used in the prototype implementation. A re-implementation of the matching taking these factors into consideration and using a faster computer (5-10 times faster computers are available) should be sufficient to increase the matching performance by two to three orders of magnitude. Hence the fact that the time scale in the tests are seconds is irrelevant.

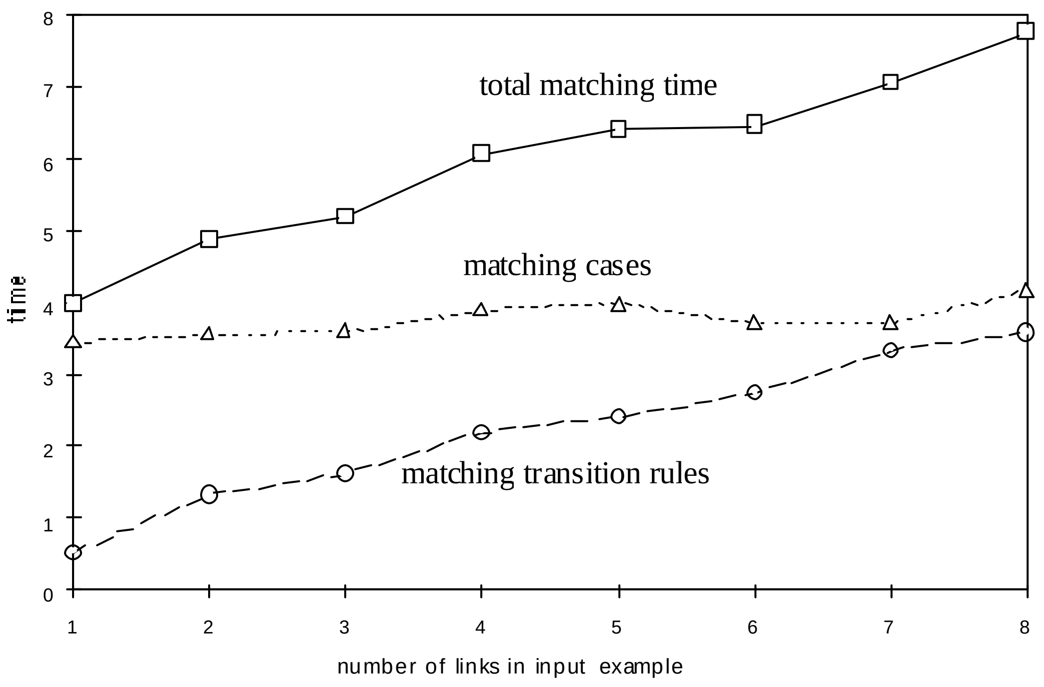


Figure 8.4: Matching time measurements, 32 cases, 225 transition rules

The variation reflect the different sizes of links, transition rules and cases. Some additional tests with different sizes of case libraries (smaller than 32 cases) showed that it is likely that the total matching time in the implementation also is linear to the size of the case library (ca. 4:1, every additional case increases the time consumption with 0.26 time units for an average sized input example, 6-8 links) in accordance with the matching algorithm. For more on optimisation strategies for matching see [Althoff, Auriol, Barletta, Manago 95]

Chapter:

9. Further Work and Extensions

In this chapter, some suggestions and ideas for further work and extensions are given. They are not presented in any particular order. Some of the proposals are minor implementation issues, which would have been implemented in the CABS system if there had been more time. Others may be seen as challenging ideas, maybe PhD projects in themselves, which I wish to document in this context to ensure that they are not lost.

9.1 Using Icons for Terms and Situations

In the links and nodes, the names of the terms and arguments provide the main means for a requirements designer to remember their meaning, which is informally described in the case library. For an alternative representation, a suggestion is to use icons (experiments with use of icons for telecommunications services have been made by [Preifelt, Engstedt, 92]). Terms or conjunctions of terms and nodes which are conjunctions of terms and links (which have the originating node as preconditions and the terminating node as conclusions) could be assigned icons. Figure 9.1 shows an idea of how a link could be represented by icons instead of terms, nodes and links. The node *all subscribers idle* in Figure 4.1 is represented by the icon in the upper right corner in Figure 9.1. When clicking on this icon, a details window could be shown (as exemplified in Figure 4.4 for the node a calling b). The next node, *dial tone* a is in the middle right and the link is represented by an icon symbolising that the receiver is lifted. In the bottom right corner is an icon representing the node *a calling b* and the link (stimulus dialling) connection the two nodes *dial tone* and *a calling b* is shown beside the arrow pointing to this node. Choosing and designing icons would be highly application domain specific. If the mapping is a direct mapping between sets of terms, links and nodes, adding such a graphical representation is a matter of implementation (but with plenty of interesting possible extensions and improvements that may be small or large research projects: graphical simulation where the output from a simulation is presented in icons representing the terms may be one of the larger ones).

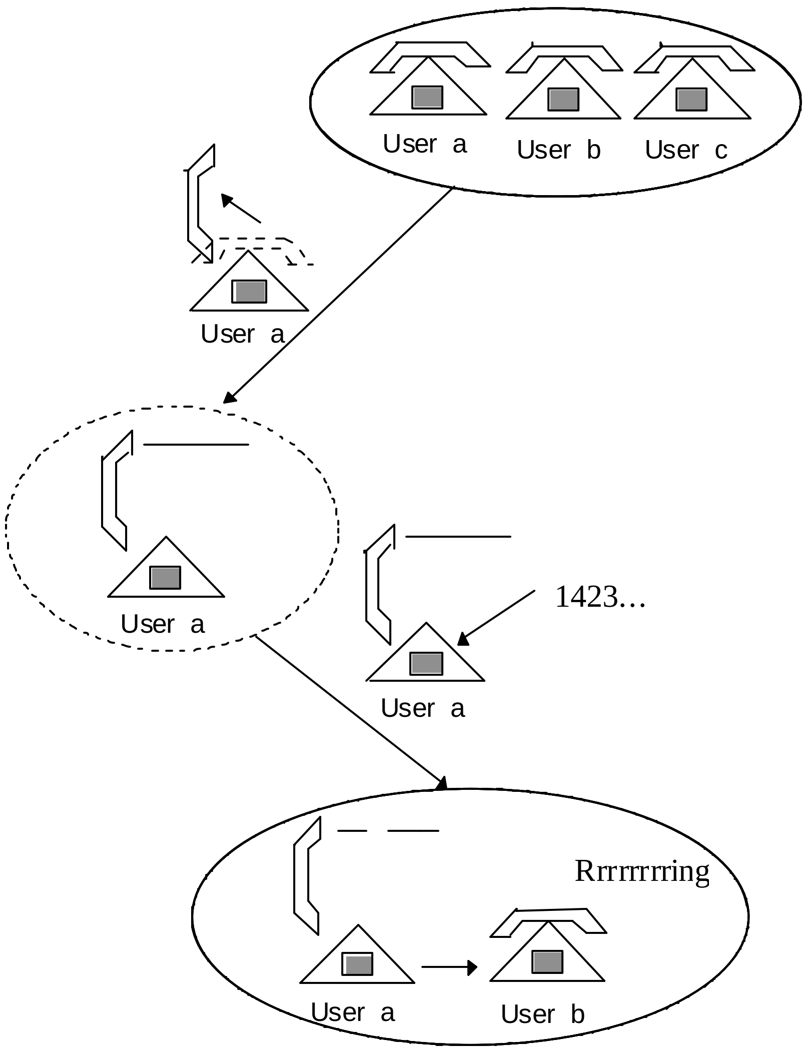


Figure 9.1: Idea of graphical representation of terms/nodes/links

9.2 Mapping Specification Against Design Objects

Most approaches to formal methods for specification have a weak connection between the specification and the actual design. Usually the specification is used for guiding the design and programming, at best the test cases are generated from the specification which may be used in a method to verify the implementation.

In large systems one of the main tasks is to update and modify the system (and hence the corresponding requirements and specifications) to meet new demands. With the weak connection between specification, design and program, the question arises of whether it is worth the effort to keep the specification up to date with changes in the system. In industry, requirements are not often maintained, which is sometimes suspected to be one of the reasons, that some years after they are written, systems start to get more and more difficult to modify and maintain.

By choosing the same formalism for the design of the different components and objects of the design, and the specification, we may use this in a mapping process. Given a new or modified specification we generate a design where we know which design components corresponds to which part in the specification. If the complete specification can be mapped in such a way that all parts of the specification correspond to design components and objects, then we have a design which can be realised.

An even stricter approach would be to only allow a specification with already designed and implemented components and objects[32]. If all the components and objects are already

[32] An analogy to this would be to let an architect only use a given set of ready made symbols in the production of a plan for a building. These symbols correspond to pre-manufactured components such as ready made walls, electricity and water pipes, floors with a ready made finish, all with a given specification. Contrast this with a plan where all walls, electricity, placing of windows and water pipes have to be worked out uniquely for each design and the building has to be built with

implemented in software or hardware then there may not be any need for programming or construction of new hardware. On a lower level, some integration and adaptation of the objects and components may still be needed. Test cases (generated from specifications, in a similar way as test cases are generated from input examples may be adopted by breadth first expansion of possible stimuli/response sequences to a chosen depth) may be used to verify that the implementation meets the requirements. An interesting question is whether it is possible to map specifications onto Service Independent Building Blocks (Sib's), as standardised and specified by the International Telecommunications Union (ITU) as part of the Intelligent Network Recommendations. If terms in a specification could be mapped against functions in a functional language (such as the concurrent programming language Erlang, [Armstrong, Virding, Williams, 1993] which is used for implementing telecommunications services), an implementation could be generated from a specification.

9.3 Using CABS for Other Application Domains

Application domains which, for practical reasons, are too large for explicitly state based approaches may be considered as potential application domains for CABS. If an application domain has a fairly simple interaction with its environment, where the connection between response and stimulus is not too complicated, but contains large numbers of states, and where it is of value to explore in detail the behaviour to show that the system will have certain properties and lack other properties, CABS may be considered. Also domains such as train signalling systems, safety systems in cars, aeronautics, power plants, computerised medical equipment (dialysis machines, scanners, etc.) may be potential application domains.

9.3.1 Object Oriented System Specifications

A similar approach to CABS may be potentially useful for requirements capture of software objects in an object oriented system. In object oriented methods it is popular to include some

bricks and concrete by highly skilled craftsmen.

state based formalism describing behavioural requirements on objects. Each object would be seen as having a closed behaviour. Stimuli and responses need to be classified as belonging to the environment of the system, or as belonging to another object in the system. Structuring the system in this way will result in some limitations in validation and verification, since CABS does not incorporate the overall validation and verification of communicating objects (but the formalised requirements in logic may be used in some theorem prover able to do validation and verification of sets of communicating objects). If behavioural requirements used in object oriented methods are similar enough to the one used in CABS, similar behaviour could be identified.

9.4 Simulation with Connected Telephones

Simulation by providing stimuli sequences in order to explore the behaviour is useful in order to explore a telecommunications service. If presenting the functionality to customers, end user or to evaluate a services popularity with telephone users before implementing the service, a simulation with real end user equipment may be useful. An interface between the simulation tool in CABS and telephones could be written. A number of phones could be connected to a PC and then the service could be tried out before ordering it, if the receiver of telephone *a* has been lifted, the stimulus *off_hook(a,1)* is sent to the simulator in CABS. The response dial_tone(a) needs to be translated by the telephone driver and a dial tone is sent. Time response for simulation of the formalised services may be sufficient if a small number of telephones/terminals are used (even if the simulator is fast, a couple of hundred telephones/terminals is to be likely a maximum if response times must be below a second).

9.5 Adding a Theorem Prover to CABS

One of the benefits of using a formal notation for requirements specification is that it enables the requirements designer to reason about the specification. This is the main advantage of a logical formalism over many other specification and programming languages [Bundy 92]. The kinds of reasoning we wish to do are:

Verification (whether the specification implements the required behaviour).

Synthesis (of specifications into a new specification).

Transformation (transform the specification into a representation using less memory and/or time when simulated).

Termination (show that no deadlocks exists).

Abstraction (abstract information about the type of its input/output etc.).

Consistency checking (prove that there are no contradictory statements in the specification).

CABS partly tackles 1 (test cases), 4 (restriction in language, see Appendix A) and 6 (a program identifying potential inconsistency between transition rules has been implemented, but not integrated). Adding a theorem prover would greatly increase CABS abilities in these areas. At the moment, there are a number of advanced theorem provers available that could be used.

9.6 Analysing Interaction Between Modules

As mentioned earlier, the condition and conclusion part in transition rules can be cross-referenced. This gives valuable information on relations between transition rules and cases. For example, if a transition rule R1 belonging to case C1 has a conclusion term T and a transition rule R2 belonging to case C2 has the term T as a condition, then we can conclude that case C1 may influence the behaviour of case C2 in one step. More obvious analyses can be made: for example, if a term only occurs in conclusions of transition rule, and is not used in any condition part of a transition rule, then the conclusion of this term is redundant. A wide variety of such analyses can be performed with straight forward cross-references between transition rules. These may be helpful in the requirements capturing process and aid the understanding of cases and their interactions, and relations.

9.7 Generating Code from State-Based Requirements

Statecharts [Harel, Naamad, 87] is part of a semi-automatic method that supports stepwise refinement to produce C, Ada or VHDL code. Formal methods for requirements specification and for program specification often have similarities, especially if the requirements specification is executable. Code is automatically generated from formal specifications, such as RSML [Heimdahl, Keenan, 1997] and non-instantaneous state transition assertions (NSA) [Gordon 86]. The code produced from RSML is 5-10 times slower than manually produced code from the state machines but if the transformations producing the code are correctness preserving, the code will have the same properties as the specification. Since both Statecharts and RSML reduce the complexity of large state transition diagrams by using *substates,* and if *substates* and CABS terms in transition rules can be mapped onto these, the approaches may potentially be combined. If combined, RSML, NSA and Statecharts would be able to apply a CABS approach to re-use and CABS would benefit from generating code from requirements. The same reasoning may be relevant for UML (Rumbaugh, Booch and Jacobson), OOA (Shlaer-Mellor) and JSD (M. Jackson) which all have graphical notations and may be extended with a re-use approach based on similar behaviour (an object with similar behaviour could be identified and proposed for re-use).

9.8 Re-Use of System Development Processes

Ericsson has a large number of detailed descriptions of system development processes that have been tailored for different projects (hardware and software) and to meet specific requirements (ISO 9000, toll-gates, milestones, well specific input/output information for different process steps). The processes are currently stored simply as pictures and text. A preliminary analysis of these processes suggests that the formal notation used in CABS might be used to describe them. It might then be possible to identify similar processes or parts of processes that can be re-used. Identifying similarities and differences can also be used to compare the solution processes to some master or standard process to identify and point out differences and suggest improvements. This possibility is being investigated with Ericsson and QLabs.

9.9 Re-Use of SDL

Re-use of SDL (se Section 2.4) diagrams form previous program implementations. SDL is more expressive than the formal notation used in CABS. Even so the graphical parts may be used as a skeleton for re-use and the formal notation in CABS may be extended to be more expressive. Since SDL is a graphical programming language that is being used more widely and outside traditional telecommunications applications, identification of similar behaviour in SDL diagrams is interesting.

Chapter:

10. Summary and Conclusions

As described in Chapter 1, formal notations can be used to formalise coarse grained telecommunications service requirements at a high level of abstraction. Formal methods for requirements have a number of advantages over informal methods, as discussed in Chapter 2.1 and 2.2. Even so, formal methods are not routinely used for telecommunications service requirements specifications. Previous research projects by Ericsson aiming at the use of formal requirements for service specifications suggest that the main reasons for this is that a number of issues have not been sufficiently addressed and solved (repeated from Section 1.1.1):

Re-use and modification of previously specified services or parts of services. The most frequent situation in the domain of telecommunications service specifications is the specification of services similar to previous ones.

The issue of iteratively refining and incrementally extending requirements that were originally sketchy, incomplete and contained errors.

End users with background in systems design and programming did not accept the idea of using the formal notation to specify services at Ericsson. Their interest in formal methods was high until they where confronted with logical axioms. Even showing slides with logical or mathematical notations drastically reduced any interest shown earlier.

These factors contributed to the cancellation of a large formal methods project and currently there is no active work at Ericsson to bring formal methods to broader use in requirements specifications for telecommunications services.

10.1 Summary of Work

In this research, the main focus is on issue 1 in the previous list and a different use of formal methods for requirements specification is proposed. Traditionally, state based formal methods for requirements specifications are used to describe the precise behaviour of all the requirements. This detailed modelling is difficult for more realistically sized problems. However, formal "sketches" of the required behaviours can be produced. The formalised service sketches are not intended to capture all the required behaviour and exclude all the unwanted behaviour, but are merely intended to sketch the key features of the behaviour required. These features are used to identify and suggest similar existing services in a case-based reasoning approach.

The similar services proposed may be adapted to the users' needs and can be validated and verified against the initial service sketches. The chosen application domain of telecommunications services is non-trivial and seventeen services often used in evaluation of service specifications have been specified and used in the evaluation. Matching is the core component of a case based reasoning system and has been the main focus of this research. In order to evaluate the matching, subsidiary components for the CBR system have been implemented: a graphical input editor where input examples can be produced and refined, a simulator to simulate the proposed and chosen solution and a verification component that generates test cases from the input example and verifies that the final solution contains this behaviour. The matching component and these subsidiary components have been implemented in the CABS system enabling the user to sketch desired behaviours of a telecommunications service, for which the CBR system proposes similar solutions from the case library that may be re-used in whole or in part. The input examples and the simulator/verification component are used to evaluate the matching algorithm. See Figure 3.1

for the different parts in CABS. Both the matching and the re-use of test cases have been put in context with an iterative requirements development method as shown in Figure 7.1.

CABS performs matching on two levels. Firstly each link in the input examples with the corresponding originating and terminating node are translated to transition rules which will only be used for matching. These input transition rules are then matched against all transition rules in the case library to identify transition rules that capture "similar" behaviour as defined in Chapter 6. Transition rules in the case library are grouped in services and the result from the transition rule matching is used to identify which of the services in the case library have a similar behaviour to the input examples. To evaluate the matching, a case library with seventeen services and twenty-one input examples of services have been used. All the input examples were very rudimentary and only captured a coarse grained sketch of a small part of the total behaviour of the corresponding service in the case library. Even so, the matching successfully identified the corresponding services (including some where the input example and service did not captured exactly the same behaviour) as evaluated in Chapter 8.

To test the proposed solution, the input examples were used to generate test cases which were automatically tested against the selected service with a batch mode of the simulator. Since the solution was known to each input example, no problems were expected in the verification, but more than half of the test cases did not pass. By analysing these, a number of mistakes were found in the input examples and in services in the case library, which shows that the verification process was useful under these circumstances. So many errors in the case library would not be expected under real conditions since all the services in the case library would already have been successfully integrated and fully implemented, and many mistakes should have been corrected during this process.

Input examples and test cases also play a role when completely new services have to be specified and there is no similar service in the case library. The input examples are translated to a set of transition rules when used in the matching, and these transition rules can be used as the starting point for a new service. During refinement of the new service, the test cases will identify where the service differs from the input examples, and the requirements designer has to either change the input examples or the service requirements.

10.2 Limitations

The formal notation used in CABS is constrained to suit a particular (efficient) matching strategy and visualisation, in this sense its simplicity is a virtue. However its limited expressiveness makes CABS unsuitable for more complex behaviour including concurrency, timing constraints, communicating processes and simultaneously occurring events, which would have been possible if a more expressive formal notation had been chosen (for example Petri nets).

If requirements specifications and formal methods are used for tasks where new requirements bear little similarity to previous requirements, more traditional use of formal methods may be preferred, i.e. mathematicians develop the formal requirements directly in a suitable formal notion using logic or algebraic notation. The proposed method is aimed at applications where re-use and modification are central issues. Using a system such as CABS would be unnecessarily limiting for problems where re-use and modification of specification is less relevant.

10.3 Future Work

This research will be continued by identifying commercially interesting areas where identification of similar behaviour is of value and where a case library with formalised cases exists or can easily be produced. By producing prototypes for this new application domain, further insights to the problem of identifying similar behaviour will be achieved.

The hope is that this result can be transferred to other application domains where comparison and re-using of similar behaviour is relevant. Some potential application domains where the identification of similar behaviour is of interest have already been identified: re-use of system development processes and re-use of SDL diagrams (SDL is briefly described in Section 2.4) as mentioned briefly in Section 9.8 and 9.9.

11. Bibliography

Aamodt A. (1993). *A Case-Based Answer to Some Problems of Knowledge-Based Systems*. Scandinavian Conference on Artificial Intelligence. E. Sandewall, C.G. Jansson (eds.), IOS Press, pp 168-182.

Aamodt A., Plaza E. (1994). *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* AI Communications, vol 7 no. 1, pp 39-59.

Acharya A. (1994). *Scaling up production systems: Issues approaches and targets*. The Knowledge Engineering Review, vol 9:1, pp 67-72.

Addis T.R. (1993). *Knowledge Science: A Pragmatic Approach to Research in Expert Systems.* ES93, pp 321-339.

Addis T.R., Gooding D.C., Townsend J.J. (1993). *Knowledge Acquisition with Visual Functional Programming*. Knowledge Acquisition for Knowledge Based Systems, 7th European Workshop, EKAW ‘93, Lecture Notes in AI 723, Springer Verlag, pp 379-406.

Allen J.F. (1983). *Maintaining Knowledge about Temporal Intervals*. Communication of the ACM, November, vol 26, Nr 11, pp 832-843.

Althoff K.-D., Auriol E., Barletta R., Manago M. (1995). A Review of Industrial Case-Based Reasoning Tolls. AI Intelligence, Oxford.

Armstrong J.L., Elshiewy N.A., Virding R. (1986). *The Phoning Philosopher's Problem or Logic Programming for Telecommunications Applications*. IEEE, pp 28-33

Ask G. (1994). *Delphi-generated TTCN Test Suites- usage in certification*. Internal Document JT/V-94:247, Ericsson, Sweden.

Ahtianen A., Chatras B., Hornbeck M., Kesti S. (1994). *Experience With Octopus Automated TTCN Translation Tools Applied to GSM/SS7*. In protocol Test Systems, vol VI, Elsevier Science.

Atkinson W., Cunningham J. (1990). *Proving Properties of a safety-critical system.* Imperial College Research Report Soc 90/28.

Bardasz T., Zeid I. (1992). *Dejavu: A Case-Based Reasoning Designer's Assistant Shell.* Artificial Intelligence in Design '92, J.S. Gero (ed.), Kluwer Academic Publishers, pp 477-496.

Barroca L.M., McDermid J.A. (1992). *Formal Methods: Use and Relevance for the Development of Safety-Critical Systems*. The Computer Journal, vol 35, No 6, pp 579-599.

Ben-Abdallah H., Leue S. (1996). *Architecture of a Requirements and Design Tool Based on Message Sequence Charts*. Technical Report 96-13, University of Waterloo, pp 1-19.

Borgida A., Greenspann S., Mylopoulus J. (1985). *Knowledge Representation as the Basis for Requirements Specification*. IEEE Computer, April.

Bose R. (1994). *Strategy for integrating object-oriented and logic programming*. Knowledge-Based Systems, vol 7, number 2, pp 66-74.

Bowen J.P., Hinchey M.G. (1996). *Seven More Myths of Formal Methods*. To appear in IEEE Software, pp 1-12.

Brandau R., Lemmon A., Lafond C. (1991). *Experience with Extended Episodes: Cases with Complex Temporal Structure*. Workshop on case-based reasoning, Morgan Kaufmann, pp 1-12.

Bubenko J.A. jr. (1995). *Challenges in Requirements Engineering*. Invited talk in Proceedings of IEEE International Symposium on Requirements Engineering, pp 160-162.

Buchanan B.G., Shortliffe E.H. (1984). Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley.

Bundy A. (1992). *Tutorial notes: reasoning about logic programs*. Second International Logic Programming Summer School, LPSS '92. Proceedings, G. Comyn, N.E. Fuchs, & M.J. Ratcliffe (eds.), Springer-Verlag, pp 232-277.

Callan J.P., Fawcett T.E., Rissland E.L., CABOT (1991). *An Adaptive Approach to Case-based Search*. Proceedings of the Twelfth International Conference on Artificial Intelligence.

Cameron E.J., Velthuijsen H. (1993). *Feature Interactions in Telecommunications Systems*. IEEE Communication, August, pp 18-23.

Capellmann C., Christensen S., Herzog U. (1998). *Visualising the Behaviour of Intelligent Networks.* Visual '98, International Workshop on Visualisation Issues for Formal Methods, ed. Margaria T, Posegga J.

Cleland G., MacKenzie D. (1995). *Inhibiting Factors, Market Structure and the Industrial Uptake of Formal Methods.* in Industrial Strength Formal Specification Techniques, Folorida, pp 46-60.

Cohn, A. G. (1985). *On the Solution of Schubert's Steamroller in Many Sorted Logic.* IJCAI, pp 1169-1174.

Cummins R. (1989). Meaning and mental representation, MIT Press, Bradford Books.

Cybulski J.L. (1996). T*he Formal and the Informal in Requirements Engineering.* Workshop on Requirements Engineering, Monash University, Caulfield, Victoria, Australia, pp 2.1-17.

Dalianis H. (1995). Concise Natural Language Generation from Formal Specifications, Taxonomic Representation. PhD thesis, University of Stockholm, The Royal Institute of Technology.

Davis E. (1990). Representations of Commonsense Knowledge, chapters 2 and 3. Morgan Kaufmann.

Domeshek E.A., Kolodner J. (1992). *Toward a Case-Based Aid for Conceptual Design*. International Journal of Expert Systems, vol 4, Number 2, pp 201-220.

Easterbrook S., Nuseibeh B. (1995). *Managing Inconsistencies in an Evolving Specification*. IEEE, pp 1-48.

Eberlein A.P.-G., Crowther M.J., Halsall F. (1996a). *RATS: A Software Tool To Aid The Transition From Service Idea To Service Implementation.*

Eberlein A.P.-G., Crowther M.J., Halsall F. (1996b). *An Expert System For The Development Of New Telecommunications Services.*

Echarti J.P., Stålmarck G. (1988). *A logical framework for specifying discrete dynamic systems*. Technical Report, Ellemtel Telecommunications System Laboratories.

Engstedt M. (1991). *A Flexible Specification Language using Natural Language and Graphics*. MSc thesis, University of Edinburgh.

Evertsz R. (1991). *The Automated Analysis of Rule-based Systems, Based on their Procedural Semantics*. Proceedings of the Twelfth International Conference on Artificial Intelligence.

Fencott P.C., Lockyer M.A., Taylor P. (1992). *The Integration of Structured and Formal Methods for Real-Time Systems Specification.* Proceedings: 5th International Conference on: Putting into practice method and tools for information system design, France, September, pp 313-323.

Fouqué G., Matwin S. (1993). *Compositional Software Reuse with Case-Based Reasoning.* Conference on Artificial Intelligence Applications 1993, IEEE, Florida.

Fuchs N., Schwitter R. (1995). *Specifying Logic Programs in Controlled Natural Language.* Workshop on Computational Logic for Natural Language Processing, Edinburgh.

Funk P.J. (1988). *Induction of Automata via Rules from Situation Sequences.* Technical Paper, University of Stockholm and Ellemtel Telecommunications System Laboratories.

Funk P.J. (1993). *Development and Maintenance of Large Formal Specifications Supported by Case-Based Reasoning.* Technical Paper TP026, University of Edinburgh.

Funk P.J., Raichman S. (1990). ROS, *An Implementation Independent Specification for ISDN.* Technical Report, Ellemtel Telecommunications System Laboratories,.

Funk P.J., Robertson D. (1994). *Requirements Specification of Telecommunications Services Assisted by Case-Based Reasoning.* The 2nd International Conference on Telecommunications Systems, Modelling and Analysis, Nashville, pp 160-169.

Gelfond M., Lifschitz V. (1993). *Representing action and change by logic programs.* Logic Programming, pp 301-321.

Gotel O.C.Z, Finkelstein A.C.W, (1994). A*n Analysis of the Requirements Tractability Problem.* Proceedings: International Conference on Requirements Engineering IEEE, pp 94-101.

Goel A.K. (1992). *Representation of Design Functions in Experience-Based Design.* Intelligent Computer Aided Design, Elsevier Science Publishers, pp 283-303.

Gordon M. (1986). *A Formal Method for Hard Real-Time Programming.* pp 379-410.

Grahlmann B. (1991). *Combining Finite Automata, Parallel Programs and SDL using Petri Nets.* TACAS'98, pp 1-16.

Heimdahl M.P.E., Leveson N.G. (1995). *Completeness and Consistency Analysis of State-Based Requirements.* ACM 95/1 pp 3-14.

Hall A. (1990). *Seven Myths of Formal Methods.* IEEE Software, pp 11-19, September.

Harel D. (1987). *Statecharts: A Visual Formalism For Complex Systems.* Science of Computer Programming 8, pp 231-274, Elsevier Science Publishers.

Harel D., Lachover H., Naamad A., Pnueli A., Politi M., Sherman R., Shtull-Trauring A., Trakhtenbrot M, (1990). STATEMATE: *A Working Environment for the Development of Complex Reactive Systems.* IEEE Transaction on Software Engineering, vol 16, no 4.

Harel D., Naamad. A. (1995). *The STATEMATE semantics of Statecharts.* Technical Report CS95-31, The Weizman Institute of Science.

Hayes P. (1985). *Some Problems and Non-Problems in Representation Theory.* in Readings In Knowledge Representation, Morgan Kaufmann Publishers Inc, pp 3-22.

Hesketh J., Robertson D., Fuchs N., Bundy A. (1996). *Lightweight Formalisation in Support of Requirements Engineering*. University of Edinburgh.

Hinchey M.G. (1993). *The Design of Real-Time Applications.* pp 178-182, IEEE.

Hirakawa M., Monden N., Yoshimoto I., Tanaka M., Ichikawa T. (1986). Hi-Visual, *A Language Supporting Visual Interaction in Programming.* in Visual Languages, Chang S., Ichikawa T., Ligomenides P. (eds.), Management and Information Systems Plenum Press, pp 233-259.

Holzmann G.J., Peled D. (1994). *An Improvement in Formal Verification.* FORTE 1994 Conference, Switzerland. pp 1-12.

Hsia P., Davis A., Kung D. (1993). *Status Report: Requirements Engineering.* IEEE Software, November, pp 75-79.

Hughes T.S., Cooling J.E. (1991). *Real-Time Systems - Animation Prototyping of Formal Specifications.* in Third International Conference on Software Engineering for Real Time Systems, Loughbourgh University, pp 51-57.

Hunt J. (1997). *Case based diagnosis and repais of software faults.* Expert Systems, vol 14, no 1, pp 15-23.

ITU I.254 Recommendation CCITT I.254 (1992). Integrated Service Digital Network, General Structure and Service Capabilities, International Telecommunications Union, Geneva, Swizerland.

ITU Z.100 Recommendation CCITT Z.100 (1994). CCITT Specification and Design Language (SDL). International Telecommunications Union, Geneva, Swizerland.

ITU X.21x Recommendation CCITT X.21x (1995). Service Definitions. International Telecommunications Union, Geneva, Swizerland.

ITU Q.1203, Recommendation CCITT Q.1203 (1992). International Telecommunications Union, Geneva, Swizerland.

Jackson P. (1990). Introduction to Expert systems, Addison-Wesley.

Jacobson I., Christerson M., Jonsson P., Övergaard G. (1993). *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison Wesley.

Jensen K. (1992). Coloured Petri Nets, Basic Concepts, Vol 1, Springer-Verlag.

Jensen K. (1997). Coloured Petri Nets, Practical Use, Vol 3, Springer-Verlag.

Johannesson P., Boman M., Bubenko J., Wangler B. (1997). Conceptual Modelling, Prentice Hall.

Johnson W.L. (1988). *Deriving Specifications from Requirements.* IEEE, pp 428-438.

Johnson W.L., Brenner K.M, (1993). *Developing Formal Specifications from Informal Requirements.* IEEE Expert, vol 8, no. 4.

Johnson W.L., Brenner K.M, Harris D.R., Sanders, (1993). *Developing Formal Specifications from Informal Requirements*. IEEE Expert, August, pp 82-90.

Karjoth G., Kooij M. (1992). *Formal Methods for the Implementation of Specifications*. pp 841-850.

Kelly V.E., Nonnenmann U. (1987). *Inferring Formal Software Specifications from Episodic Descriptions*. Sixth National Conference on Artificial Intelligence.

Kelly V.E., Nonnenmann U. (1991). *Reducing the Complexity of Formal Specification Acquisition*. Automating Software Design, M. Lowry, & R. McCartney (eds.), pp 41-64.

Ketler K. (1993). *Case-Based Reasoning: An Introduction*. Expert Systems With Applications, vol 6, pp 3-8.

Klusener S., Vlijmen B., Waveren A. (1993). S*ervice Independent Building Blocks-I; Concepts, Examples and Formal Specifications*. Technical Report P9310, University of Amsterdam,.

Kolodner J. (1991). *Improving Human Decision Making through Case-Based Decision Aiding*. AI Magazine, Summer, pp 52-68.

Kolodner J.L. (1993). Case-Based Reasoning. Morgan Kaufmann.

Kowalski R., Sergot M. (1986). *A Logic-based Calculus of Events*. New Generation Computing 4, Springer-Verla, pp 67-95.

Larkin J.H., Simon H.A. (1987). *Why a Diagram is (Sometimes) Worth Ten Thousand Words.* Journal: Cognetive Science, vol 11, pp 65-99.

Lecceuche R., Robertson D., Barry C. (1998). *Acquisition of Focus Rules for Requirements Elicitation Systems.* Submittet to ECAI 98.

Leue S. (1995). *Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach.* Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95, Chapmann & Hall, pp 19-34.

Luger G.F., Stubblefield W.A. (1989). Artificial Intelligence and the Design of Expert Systems, Benjamin/Cummings Publishing.

Maiden N.A.M., Mistry P., Sutcliffe A.G. (1995). *How People Categorise Requirements for Reuse: a Natural Approach.* Proceedings of Second IEEE International Symposium on Requirements Engineering, pp 148-155.

Maiden N.A.M., Sutcliffe A.G. (1995). *Requirements Engineering by Example: an Empirical Study*. Proceedings of IEEE International Symposium on Requirements Engineering, pp 104-111.

Malec J. (1992).*Process Transition Networks: The Final Report.* Technical Report LiTH-IDA-R-92-07, Linköping University, pp 1-31.

Mark M., Greer J. (1993). *Evaluation Methodologies for intelligent Tutoring Systems.* Journal of Artificial intelligence in Education, vol4. no 2/3, pp 129-153.

Mataga P., Zave P. (1993). *Formal Specifications of Telephone Features*. pp 20-49.

Mizuno O., Niitsu Y, *A Method of Designing Communication Service Specifications Using Message Sequence Charts*. Electronics and Communications in Japan, Part 1, vol 76, pp 1-15.

Moor D.J., Swartout W.R. (1988). *Explanation in expert systems: a survey*. Research Report ISIRR, University of Southern California, pp 88-228.

Mostow J., Barley M., Weinrich T. (1989). *Automated reuse of design plans*. Artificial Intelligence in Engineering, vol 4, no. 4, pp 181-196.

Mott S. (1993). *Case-Based Reasoning: Market, Applications, and Fit With Other Technologies*. Expert Systems With Applications, vol 6, pp 97-104, Pergamon Press Ltd.

Muggleton S.,(1990). Inductive Acquisition of Expert Knowledge, Turing Institute Press and Addison-Wesley.

Nakata K. (1992). *Behavioural Specification with Nonmonotonic Temporal Logic*. D. Finn (ed.) Preliminary Stages of Engineering Analysis and Modelling Workshop, AID '92, pp 41-45.

Nakatani Y., Tsukiyama M., Fukuda T. (1992). Engineering Design Support Framework by Case-Based Reasoning. ISA Transaction, vol 31, no. 2, pp 235-180.

Nonnenmann U., Eddy J.K. (1992). *KITSS - A functional Software Testing System Using a Hybrid Domain Model*. IEEE, pp 136-142.

Nyström J. H., Jonsson B. (1996). F*ormalization of Service Independent Building Blocks*. AIN'96 workshop, Passau. pp 1-14.

O'Leary D. (1993). *Verification and Validation of Case-Based System.* Expert Systems With Applications, vol 6, Pergamon Press Ltd, pp 57-66.

Pearce M., Goel A.K., Kolodner J.L., Simring C., Sentosa L., Billington R. (1992). *Case-Based Design Support.* IEEE, October, pp 14-20.

Pohl K. (1994). *The Three Dimensions of Requirements Engineering: A Framework and its Applications.* Information Systems, vol 19, no 3, pp 243-258.

Preifelt S., Engstedt M. (1992). *Results from the VINST Project.* In Swedish. Technical Report, Ellemtel Telecommunications Systems Laboratories.

Quinlan J.R. (1987). *Generating Production Rules From Decision Trees.* Proceedings of the Tenth International Joint Conference in AI, Morgan Kaufmann Publisher.

Regensburger F., Barnard A. (1998). *Formal verification of SDL systems at the Siemens mobile phone department.* TACAS'98, pp 439-455.

Ridley G.A. (1994). *Description of TTCN Test Suite Generation from AUC Delphi Specification.* Internal Document F94 2194, Ericsson, Sweden.

Ridley G.A., Höök H., Engstedt M., Lapins E., Lindroos L. (1997). Formal specification system ECLARE. Internal Document UR97, Ericsson, Sweden.

Riesbeck C., Schank R. (1989). Inside Case-Based Reasoning, Lawrence Erlbaum Inc.

Robertson D. (1996). *Distributed Specifications.* ECAI 96, 12th European Conference on Artificial Intelligence, Budapest, Hungary, John Wiley & Sons Ltd, pp 390-394.

Robertson D., Agusti J. (1998). Automated Reasoning in Conceptual Modelling, draft book, available from the authors at DAI Edinburgh.

Sandewall E. (1990). *Proposal for a ProArt specification platform.* Technical Report LAIC-IDA-90-TR18, Linköping University.

Segala R., Gawlick R., Søgaard-Andersen J., Lynch N. (1998). Liveness in Timed and Untimed Systems. Submitted for journal publication. Available from the authors. pp 1-52.

Schofield M. (1992). *Formal Methods: The Next Generation of System Design Tools.* Quality and Reliability Engineering International. vol 8, pp 549-555.

Semmens L.T., France R.B., Docker T.W.G. (1992). I*ntegrated Structured Analysis and Formal Specification Techniques*. The Computer Journal, vol 35, No 6, pp 600-610.

Simon H.A. (1981). The Sciences of the Artificial, The MIT Press 1969, Massachusetts, Second edition reprint.

Skoglund N. (1993). *Systemhantering med regler.* In Swedish, Internal Document, Ellemtel Telecommunications System Laboratories.

Smyth B., Keane M.T. (1994). *Retrieving Adaptable Cases*. In: S. Wess, K.-D. Althoff, & M.M. Richter (eds.), Topics in Case-Based Reasoning, Springer-Verlag.

Sommerville I. (1996). Software Engineering, fifth edition part one & five, Addison Wesley.

Sycara K.P., Navinchandra D., Guttal R., Koning J., Narasimhan S. (1992). *CADET: A Case-Based Synthesis Tool for Engineering Design*. International Journal of Expert Systems, vol 4, no. 2, pp 167-188.

Uschold M. (1996). *Building Ontologies: Towards a Unified Methodology*. Proceedings of Expert Systems 1996, Cambridge, UK,.

Vargas-Vera M., Robertson D., Inder R. (1993). *An Environment for Combining Prolog Programs*. In Third International Workshop on Logical Program Synthesis and Transformation.

Verpers K. (1991). Induction of rules from Behavioural Sequences (in Swedish). M.Sc. dissertation, Royal Institute of Technology, Stockholm, Sweden.

Watanabe L., Rendell L. (1991). Learning Structural Decision Trees from Examples. Proceedings of the Twelfth International Conference on Artificial Intelligence.

Watson I. (1997). Applying Case-Based Reasoning: Techniques for Enterprise Systems, Morgan Kaufmann.

Wenger E. (1987). Artificial Intelligence and Tutoring Systems (Computational and Cognitive Approaches to the Communication of Knowledge), pp 261-288. Morgan Kaufmann Publishers, Inc.

Wieringa R.J. (1996). Requirements engineering: Framework for understanding. John Wiley & Sons Ltd, Chichesters.

Wringa R., Dubois E. and Huyts S. (1997). Integrating Semiformal and Formal Requirements. Proceedings of the Ninth International Conference on Advance Information Systems Engineering (CAiSE'97), Barcelona, Spain.

Wing J.M. (1990). A specifier's introduction to Formal Methods. Computer, vol 23. pp 8-24.

Yang S.-A., Robertson D., Lee J. (1995). Use of Case-Based Reasoning in the Domain of Building Regulations. Topics in Case-Based Reasoning, Springer-Verlag, pp 292-306.

Zave P. (1991). An Insider's Evaluation of PAISLey. IEEE Transaction on Software Engineering, vol 17, no. 3. March.

Zave P. (1993). Feature Interactions and Formal Specifications in Telecommunications. Computer, vol 26, no. 8.

Zave P., Jackson M. (1996). Four Dark Corners of Requirements Engineering. ACM pp 1-34.