Verification of LOTOS Specifications using Term Rewriting Techniques

Carron Elizabeth Kirkwood

Submitted for the Degree of Doctor of Philosophy.

Research carried out in the Department of Computing Science,
University of Glasgow.

Thesis
9799
Copy 1

Abstract

Recently the use of formal methods in describing and analysing the behaviour of (computer) systems has become more common. This has resulted in the proliferation of a wide variety of different specification formalisms, together with analytical techniques and methodologies for specification development. The particular specification formalism adopted for this study is LOTOS, an ISO standard formal description technique. Although there are many works dealing with how to write LOTOS specifications and how to develop a LOTOS specification from the initial abstract requirements specification to concrete implementation, relatively few works are concerned with the problems of expressing and proving the correctness of LOTOS specifications, i.e. *verification*. The main objective of this thesis is to address this shortfall by investigating the meaning of verification as it relates to concurrent systems in general, and in particular to those systems described using LOTOS. Further goals are to automate the verification process using equational reasoning and term rewriting, and also to attempt to make the results of this work, both theoretical and practical, as accessible to LOTOS practitioners as possible.

After introducing the LOTOS language and related formalisms, the thesis continues with a survey of approaches to verification of concurrent systems with a view to identifying those approaches suitable for use in verification of properties of systems specified using LOTOS. Both general methodology and specific implementation techniques are considered. As a result of this survey, two useful approaches are identified. Both are based on the technique of expressing the correctness of a LOTOS specification by comparison with another, typically more abstract, specification. The second approach, covered later in the thesis, uses logic for the more abstract specification. The main part of the thesis is concerned with the first approach, in which both specifications are described in LOTOS, and the comparison is expressed by a behavioural equivalence or preorder relation. This approach is further explored by means of proofs based on the paradigm of equational reasoning, implemented by term rewriting.

Initially, only *Basic* LOTOS (i.e. the process algebra) is considered. A complete (i.e. confluent and terminating) rule set for weak bisimulation congruence over a subset of Basic LOTOS is developed using RRL (Rewrite Rule Laboratory). Although fully automatic, this proof technique is found to be insufficient for anything other than finite toy examples. In order to give more power, the rule set is supplemented by an incomplete set of rules expressing the expansion law. The incompleteness of the rule set necessitates the use of a *strategy* in applying the rules, as indiscriminate application of the rules may lead to non-termination of the rewriting. A case study illustrates the use of these rules, and also the effect of different interpretations of the verification requirement on the outcome of the proof.

This proof technique, as a result of the deficiencies of the tool on which it is based, has two major failings: an inability to handle recursion, and no opportunity for user control in the proof. Moving to a different tool, PAM (Process Algebra Manipulator), allows correction of these faults, but at the cost of automation. The new implementation acts merely as computerised pencil and paper, although tactics can be defined which allow some degree of automation. Equations may be applied in either direction, therefore completion is no longer as important. (Note that the tactic language could be used to describe a a complete set of rules which would give an automatic proof technique, therefore some effort towards completion is still desirable. However, since LOTOS weak bisimulation congruence is undecidable, there can never be a complete rule set for deciding equivalence of terms from the full LOTOS language.) The composition of the rule set is reconsidered, with a view to using alternative axiomatisations of weak bisimulation congruence: two main axiomatisations are described and their relative merits compared. The axiomatisation of other LOTOS relations is also considered. In particular, we consider the pitfalls of axiomatising the **cred** preorder relation.

In order to demonstrate the use of the PAM proof system developed, the case study, modified to use recursion, is re-examined. Four other examples taken from the literature, one substantial, the others fairly small, are also investigated to further demonstrate the applicability of the PAM proof system to a variety of examples.

The above approach considers Basic LOTOS only; to be more generally applicable the verification of properties of *full* LOTOS specifications (i.e. including abstract data types) must also be studied. Methods for proving the equivalence of full LOTOS specifications are examined, including a modification of the technique used successfully above. The application of this technique is illustrated via proofs of the equivalence of three variants of the well-known *stack* example. The proofs are carried out by hand as neither of the implementation tools used above are able to handle data types. The approaches of other authors to verification of full LOTOS specifications are also described and illustrated by examples in order to propose an approach to verification comprising several complementary techniques.

Finally, the verification of LOTOS specifications where the abstract requirements are expressed using temporal/modal logic is briefly considered. Specific reference is made to the existing linear temporal logic used in conjunction with LOTOS and also to the use of HML (Hennessy-Milner Logic) in conjunction with CCS. The possibility of using HML with Basic LOTOS is discussed at length, with examples drawn from earlier in the thesis. Also considered is the possibility of extending the logic for use with full LOTOS. Both of these proposals require further investigation.

Contents

Chapter 1

Introduction

1.1 Background

The last few years have seen an increasing interest in the use of *formal methods* in the design and analysis of computer systems. By formal methods, we mean the application of mathematical concepts to the modelling of a real world problem. Most commonly this means the use of a formal specification language, i.e. one which has a formal mathematical semantics, to describe the system. The use of formal methods allows us to build a (simplified) mathematical model of a real world phenomenon which can then be analysed using mathematical techniques. The results gained can then be used to deduce results about the properties of the real world system. Of course, the applicability of these results to the real world system is highly dependent on the accuracy of our mathematical model.

Use of formal methods can aid practitioners in two main ways. Firstly, by virtue of their mathematical basis, formal specifications allow clear, precise and unambiguous descriptions of a system; moreover, descriptions can be written without reference to implementation issues. Secondly, the existence of formal semantics for the specification language makes rigorous mathematical analysis, i.e. *verification*, of the specification possible. Such analysis can improve confidence in the correctness of the design and/or implementation. It may also lead to a better/deeper understanding of the system, especially in cases where the analysis detects some (non-trivial) error in the specification.

A variety of different specification formalisms are in current use in many different areas of application, e.g. VDM, Z, denotational semantics, petri nets, finite state automata, process algebras, ... Although the choice seems endless, it may be narrowed down by the requirements of the area of application. We are interested in the verification (and therefore the specification) of *concurrent systems*. For this application area, a popular means of specification is *process algebra*. Prominent examples of process algebras include ACP [BK84], CCS [Mil80, Mil89b], CIRCAL [Mil85], CSP [Hoa85], SCCS (Synchronous CCS) [Mil89b], and MEIJE [AB84]. They are prominent because of *popularity*, i.e. used in many applications and studies, both academic and industrial, or *theoretical significance*, e.g. it has been shown in [dS85] that SCCS and MEIJE are *universal process algebras*, in the sense that all other process algebras can be described in terms of SCCS or MEIJE.

It should be realised that universality is not necessarily a desirable feature of a language. While a universal process algebra is able to describe every sort of system, it may be that these descriptions are clumsy. More importantly, because it is so general, the theory for verification may be less rich than if we used a language which has a smaller scope and is perhaps specifically designed for our purpose. For this reason neither of the universal process algebras mentioned above are considered further here. The matter of why variety is important in process algebras is further discussed in [BBH$^+$91].

Another consideration not mentioned above is *standardisation.* Although the formalisms above are of *academic* significance, they have not been widely adopted for industrial use. For internationally standardised specification formalisms we must look to ISO (the International Standards Organisation).

One of the tasks undertaken by ISO is the development of *open systems* which will provide a uniform framework for communication throughout the world. This means that the protocols controlling communication must behave in exactly the same way whenever and wherever they are executed, regardless of local variables such as implementation language, machine architecture and physical location.

The focal point of the standardisation process is the seven layer Reference Model [ISO74] which describes the interaction between systems in an abstract, implementation independent way. Obviously, given the aims of ISO, the first essential is to have a specification which cannot be interpreted in any way other than that intended by the specifier, implying the use of a formal specification language. With this in mind, ISO have introduced three internationally standard formal description techniques (fdts) to support the Reference Model: Estelle [ISO90], SDL [CCI88] and LOTOS [ISO88]. These are to be used to give formal specifications of the services and protocols which make up the Reference Model. [Vis90] gives a historical overview of the development of these languages and related standards, and [Tur93] gives an introduction to, and comparison of, the languages.

Support for the formal description techniques is provided in the form of European research and development projects. In the past these projects have resulted in development methodologies and analytical tools for the fdts. Initially the projects were joint ventures, encompassing all three fdts, for example, the SEDOS project [vEVD89]. More recently, as interest in the different languages has grown, the projects have been specific to particular fdts, e.g. LOTOSPHERE [vSPV92]. Following this lead, we also concentrate on just one of the fdts, LOTOS, for the following reasons.

The LOTOS language has two parts: process algebra, which can be used to specify the control aspect of a system, and abstract data type, which can be used to specify the data manipulated by the system. The process algebra part was developed from the formalisms CCS and CSP; this allows us to link our study to both academic and industrial concerns. The theory behind CCS and CSP is now well developed, and there is a wide literature on many aspects of analysis of CCS and CSP specifications. Due to the close relationship between these formalisms and LOTOS it may be possible to transfer results obtained for these formalisms to the LOTOS setting. Our main reason for choosing to study LOTOS rather than CCS or CSP is its status as an international standard. We cannot investigate all aspects of the definition and application of LOTOS in one thesis, therefore we concentrate on *verification* of properties of LOTOS specifications.

The fdts of ISO were all designed with specification in mind; therefore the expressivity of the language was given more importance than the simplicity of the semantics. While specification in itself is a valuable undertaking, providing clear descriptions and perhaps a better understanding of the system, ideally, from our point of view, more emphasis should be put on methods for checking the correctness of a specification. The more expressive a language is, the more complex the semantics, and the harder it becomes to verify correctness of specifications in that language. Verification really needs to be considered right from the first stages of specification, rather than tackled after specification is completed; this point is made in [HJOP89]. This conflict between the expressive power of the language and the simplicity of its semantics is particularly relevant to LOTOS, which is encumbered by a verbose syntax, making analyses longer and more tedious, and also obscuring the simplicity of the underlying system.

Several forms of analysis can be considered for LOTOS, including debugging, simulation, testing and verification. While projects such as LOTOSPHERE have been successful in providing a structured development methodology together with some tool support, other aspects of the development process have been largely ignored. In particular, although much effort has been expended on, for example, testing techniques, relatively little has been directed towards the problems of verification, particularly verification of properties of *full* LOTOS specifications. A desire to rectify this imbalance is the main aim of this thesis; more specific aims and objectives are detailed in the next section. We are interested in verification because it provides rigorous mathematical analysis of a system. While verification cannot provide 100% certainty that a system is correct, it can greatly increase our confidence in the general correctness of the system, and the correctness with respect to particular properties (assuming the properties are expressed accurately). Verification can give more confidence in the system than a testing method can.

Another aspect of verification is *automation*. Even for small systems, analysis can be tedious and error prone, therefore it is vital to have some sort of machine assistance. Although full automation of any analyses is preferable, in practice we may have to settle for partial automation.

A wide variety of methods for automation exist for different aspects of verification. We consider a general method of proof and automation, the paradigm of equational reasoning implemented by term rewriting, and consider how two forms of analysis may be carried out by equational reasoning and automated by term rewriting. Process algebras are all associated with sets of laws or axioms corresponding to different notions of equivalence, making equational reasoning a natural proof technique to use.

The next section details our aims and objectives in the study of verification of properties of LOTOS specifications.

1.2 Aims and Objectives

The aims of this thesis are:

- to investigate the verification of concurrent systems described using the formal description technique LOTOS,
- to apply this knowledge by developing proof methods for verifying the correctness of LOTOS specifications. These methods should make use of existing proof tools rather than necessitating the implementation of new software. The proof technique used will be equational reasoning, automated by term rewriting.
- to make the results (both theoretical and practical) of this investigation accessible to LOTOS practitioners.

These aims will be achieved by the following objectives:

- to build knowledge of LOTOS, the LOTOS related process algebras CCS and CSP, and their associated approaches to verification,
- to use this knowledge in defining what verification means for concurrent systems in general, and more specifically for both Basic LOTOS *and* full LOTOS,
- from the above study, to identify an approach to verification which can be automated using equational reasoning and term rewriting,
- to develop a proof system (based on existing tools) implementing that approach to verification,
- to demonstrate the usefulness of the proof system through examples.

We believe these aims and objectives have been largely achieved in the thesis. Although we do not claim to have completely investigated verification for LOTOS (we have only briefly covered some aspects of verification, such as the use of temporal/modal logics in specifying the requirements of the system[1]), we have thoroughly explored one aspect of verification, namely that of proving two specifications (both described using LOTOS) are related by a behavioural equivalence or preorder. This has been achieved through theoretical and practical investigations, the practical work being carried out using equational reasoning and term rewriting tools. The proof technique we develop in the thesis is illustrated through several examples, some small, others medium sized.

The following section gives a more detailed account of the work carried out.

1.3 Overview

The thesis is organised around the main topic of verification; particular questions addressed are: what is meant by verification, what kind of verification can be carried out on LOTOS specifications, how can the proofs of verification can be automated, and what do those results tell us about the system under examination?

Chapter 2 contains an informal discussion of what is meant by *verification*. The view taken here is that verification is the formal, mathematical expression and proof of the correctness of a concrete description of a system with respect to some set of (formal) requirements. Of course, the requirements also constitute a description of the system, at a more abstract level. As the two descriptions need not be expressed using the same formalism, two main cases are considered; the bulk of the thesis is concerned with the case in which both descriptions are expressed using LOTOS. An alternative case in which the concrete description is expressed in LOTOS and the requirements are expressed using a temporal or modal logic is considered in chapter 11.

What does verification mean in relation to systems specified using LOTOS? In order to consider this question a good working knowledge of the semantics of LOTOS and the ways in which LOTOS specifications can be compared is required. It is also helpful to have the same knowledge for CCS and CSP, since LOTOS was developed from these formalisms. Since both CCS and CSP have a rich literature, it may be possible to adapt proof techniques from either of these formalisms for use with LOTOS. The three formalisms, LOTOS, CCS and CSP, are presented in some detail in chapter 3, including a summary of their main differences (and similarities). At this point we consider only Basic LOTOS, which aids the comparison with CCS and CSP. Full LOTOS is considered in chapter 10.

The survey of the three process algebras is followed by the first detailed section on verification,

[1]The investigation of temporal logic in conjunction with LOTOS is the basis of a SERC funded project, "Temporal Aspects of Verification of LOTOS Specifications", which will run over the next two years.

chapter 4. The technique of comparing two descriptions of a system where both are written using the same specification language is one which is commonly used in the process algebra literature. Many proof techniques and tools based on this approach are currently in use, and a wide variety of equivalence relations and/or preorders have been developed to express ways of comparing specifications. A side issue considered briefly here is the selection of the most appropriate relation for a given problem.

The available proof techniques and tools are surveyed, compared and considered for use in conjunction with LOTOS. Although there are some fast algorithms for deciding equivalence of processes (based on graph partition algorithms), in general these do not give any intuition in the case where the two specifications are not equivalent (they only answer yes or no). Such algorithms also rely on a special internal representation of the process specifications for their calculations. In this work our preference is for a proof technique in which no special intermediate forms are required, and which may give some insight into the workings of the system under consideration, especially in the case in which the two specifications are not equivalent. Equational reasoning is such a proof technique and is adopted for use in the practical work. Equational reasoning is automated by term rewriting. The basic theory of term rewriting, including Knuth-Bendix completion, is presented in chapter 5. We also discuss the discovery of an inconsistency, which we found by rewriting techniques, in the laws of weak bisimulation congruence of [ISO88].

Chapters 6 to 9 detail the various components of the practical work. The initial aim of the practical work is to form a complete (i.e. confluent and terminating) set of rewrite rules (giving a decision procedure) for LOTOS weak bisimulation congruence using the tool RRL (Rewrite Rule Laboratory). This is described in chapter 6. A complete rule set for a *subset* of the language is developed. No complete set for the full language can exist because weak bisimulation congruence is known to be undecidable. Several small examples of proofs by rewriting demonstrate the use of the rewrite rule set and also the need for rules expressing the full power of the expansion law (which allows parallelism to be expressed in terms of sequencing and choice). A set of rules to achieve this is developed. As the new set of rules is not complete, a strategy in applying the rules must be adopted, otherwise the rewriting may not terminate.

To illustrate the use of this verification technique, a case study is introduced in chapter 7. The case study has two purposes: firstly, to obtain a successful proof, using the rewrite rules developed with RRL, of the requirement that the specification of the system is satisfied by the implementation, and secondly, to discover the effect of different interpretations of this requirement on the outcome of the proof. It is interesting to note that under the initial, intuitive interpretation of the verification requirement the implementation cannot be proved to satisfy the specification. In the end the specification has to be altered (in a modular way, using the constraint oriented specification style) in order to complete the proof. As a result of this study a number of deficiencies in the original verification technique are identified; the most important is the inability to handle recursive processes, but also significant is the inflexibility of the RRL system and lack of opportunity for user intervention.

Chapter 8 introduces our second approach to using term rewriting proof techniques for verification of Basic LOTOS specifications. A different tool, PAM (Process Algebra Manipulator), which *can* perform proofs on specifications incorporating recursive processes, is adopted. This new power is balanced by the fact that PAM cannot perform proofs automatically; the user must guide every step. However, a number of tactics describing patterns of rule application may be defined, allowing some limited form of automation.

An important decision in setting up PAM is how to express the LOTOS laws, and indeed whether all laws are necessary for most examples. The relative merits of different solutions to this question are considered. This leads on to implementation of equivalences other than weak bisimulation congruence. Also discussed is the problem of axiomatising a preorder relation.

In chapter 9 the case study example is repeated (this time with recursive processes). In order to further show the utility of the system, a number of other examples are also presented. These are a simple radiation machine, the reader/writer problem, a nondeterministic candy machine and the scheduler. Of these, the most significant, and largest, is the radiation machine study. Proofs are presented of the safety (or not) of several variants of the machine; the most interesting are the proofs of safety. These could not be completed using PAM, as reasoning external to our proof system had to be employed. All of the examples are taken from the papers of other authors.

Until this point only one half of the LOTOS language has been considered; namely the process algebra part, Basic LOTOS. However, full LOTOS also incorporates an abstract data type language, ACT ONE. In chapter 10 the proof technique used so far is reviewed and the question of how the inclusion of data types might alter the verification process is considered. The modified proof technique is illustrated by means of an example. Three descriptions of the stack, each with varying emphasis on the process algebra, are compared using weak bisimulation congruence; the proofs are carried out by hand. Hand proofs are normally tedious and error prone: these are no exception. This "feature" is only exaggerated by including data types. The technique seems of limited value without automation (which is not possible due to the limitations of current tools) so approaches by other authors to the problem of verification of full LOTOS are also surveyed. The two main approaches we consider both work on the principle of removing the abstract data types from the specification to obtain a Basic LOTOS specification, and evaluating correctness using the better understood Basic LOTOS proof techniques. The first approach provides a method for encoding the data values of a full LOTOS specification in a Basic LOTOS one, and may be varied to preserve some, all, or none of the data type information. The other approach is really a method for deriving a process algebra specification from an abstract data type specification, preserving all data type information in the derivation. We illustrate both approaches by our own examples: one using the Stack example and a hand proof, the other using the radiation machine study of section 9.2 and the PAM implementation to automate the Basic LOTOS proof. In the absence of one really useful and generally applicable proof technique for verification of properties of full LOTOS specifications, it seems that a composite approach may be the best solution.

In the initial discussion of verification in chapter 2, two main approaches to verification were identified. Although the main approach of proving equivalence between LOTOS specifications was demonstrated in chapter 9 to be fairly successful for Basic LOTOS specifications, chapter 10 showed that it is not as suitable for full LOTOS specifications. The second approach to verification mentioned in chapter 2 considers the situation in which one description (usually the more abstract specification) is written using some form of logic. This allows the desirable properties of the system to be described in a more abstract, less constructive manner. The current state of verification with respect to this approach is surveyed in chapter 11. Although a linear temporal logic has been developed for use in conjunction with LOTOS, it is not satisfactory as the equivalence induced by the logic is the rather weak trace equivalence, meaning that deadlock properties are not preserved. We conjecture that a variant of HML (the logic commonly used with CCS) might be adapted for use with Basic LOTOS; we present the logic, outline the proof technique and give some re-specification, in logic, of earlier examples. A natural progression is to consider what sort of logic would be required for use with full LOTOS; we discuss this topic, illustrating the discussion by examples, but the possibility is not pursued. This work will be the subject of a future investigation, as mentioned earlier.

Finally, chapter 12 concludes our study with a discussion of what has been achieved, how far our work has gone towards meeting the original objectives, open problems and further work.

Four appendices are attached: appendix A consists of a survey of existing tools for verification of specifications written using process algebras and tools for LOTOS (proof tools and otherwise), appendix B presents the LOTOS syntax and semantics, and appendices C and D give the input files used in RRL and PAM respectively.

Acknowledgements

My interest in the topic of verification of LOTOS specifications began when I was employed on the SERC project "Verification Techniques for LOTOS" [VTL93]. I want to thank my supervisor Muffy Thomas for getting me involved in all of this, providing the stimulus for much the work described herein and for patiently proof-reading countless drafts of bits of this thesis.

During the course of the project I had many illuminating conversations with colleagues involved in the project from Royal Holloway and Bedford New College and Rutherford Appleton Laboratory. Amongst my colleagues, I particularly want to thank Phil Watson, who shared an office with me for three years, during which time he helped guide me through the hard maths! I also want to thank visitors to our project meetings: Jeremy Dick, who provided the case study example described in chapter 7, and friendly rivals in the field, Paula Inverardi and Monica Nesi.

The Department of Computing Science has been a great environment in which to work. I'd like to thank all my colleagues there, in particular, the members of the Formal Methods group, who probably never want to hear me talk about the case study example ever again. The University of Glasgow also played a part by funding me as a University Scholar from October 1992 to September 1993, allowing me to write this thesis.

The tools used in this work came from the following people: Deepak Kapur (RRL) and Huimin Lin (PAM). Thank you particularly to Huimin, who was always ready to answer my questions about PAM.

For moral support, I want to thank all my family; the Shanklands and the Kirkwoods. Last but by no means least, thanks to my best friend Derick, who supplied endless cups of tea, never stopped believing in me, and generally put up with my ranting and raving throughout the process.

Chapter 2

Verification Requirements I

In this chapter the meaning of the term *verification* is discussed, initially in the general setting of concurrent systems. As there are many different uses of the term in the literature, we try to identify what exactly we mean by verification: what it is, what it is not, and at what point in the development of a system verification techniques may be applied. We feel it is important to clarify from the outset what we understand verification to mean so that it is clear to the reader what we are trying to achieve in the wider aims of the thesis. The discussion of the meaning of verification can of course be applied to verification of properties of systems described using LOTOS, and, where appropriate, specific examples from the LOTOS literature are used to illustrate certain forms of system analysis.

2.1 What Do We Mean By "Verification"?

As mentioned in the introduction, there are several different approaches to the analysis of specifications, some formal, some informal. For example, we may carry out syntactic analysis of the specification to ensure expressions are well formed and well typed. This sort of analysis can be easily carried out by machine. Indeed, machines can perform such tasks with greater accuracy than humans, because by hand they become tedious and errors are then easily missed.

Another form of analysis involves the use of a simulator to try to detect invalid sequences of events by "executing" the specification. A similar function may be carried out by performing a test process describing the sequence of events in parallel with the specification, synchronising on all events. If the test process reaches a special "test passed" event, then we know that the specification can perform that sequence of events. Using these simulation methods we can check for good or bad behaviours in a system, and correct any errors found. These forms of analysis involve only syntactic manipulation of the system, and have been automated for LOTOS, e.g.

various components of the LITE toolkit [LITE]. Appendix A.4 provides a brief survey of such tools.

Unfortunately, due to the non-exhaustive nature of simulations and tests, all errors in a specification may not be discovered by such methods and tools. While the presence of errors in all but the simplest of specifications is unavoidable, there are some further analyses which may further reduce the number of errors left undetected. This brings us to *verification.*

In the literature there seems to be a great confusion over exactly what is meant by *verification*: everyone has their own, slightly different, interpretation. In particular, *verification* is often confused with *validation.* Our understanding of these terms, supported by various sources [CR90, vG90, Bri88b], is as follows:

verification Formal, rigorous proofs of properties of the system by manipulations of axioms and known truths.

validation A convincing demonstration of conjectures. Proof by experiment.

For example, verification may involve formal proofs of the equivalence of two specifications, or that a particular property holds of a given system. Validation, on the other hand, tends to be less formal, and includes, for example, the application of tests to an implementation, or simulation of a specification, until a "sufficient" number of test have been passed, or "enough" behaviour has been observed in the simulation. Of course "sufficient" and "enough" are highly subjective evaluations. Since both analyses are typically non-exhaustive, we use validation techniques to increase our confidence in the correctness of the system, but we can never be sure that we haven't missed some important test. Therefore, although validation is useful, especially in the early stages of design/specification/implementation, because it is usually less time consuming than verification, we are more interested in verification because of its wholly formal basis.

An alternative definition of validation, found in, for example, [HJOP89], includes verification as a *subclass* of validation, i.e. the definition of validation is similar to that above, but validation activities consist of testing, simulation *and* verification. This definition is compatible with our own because verification retains the same meaning.

2.2 System Development and Verification

Having settled on a definition of verification, the next question to be considered is "at which point in the development of a system can/should verification techniques be applied?". To try to answer this question, we consider the flow of system development as put forward in figure 2.1.

We assume the `Requirements` are the informal requirements of the customer, written informally in a natural language, the `Specification` is given formally, written in any of a variety

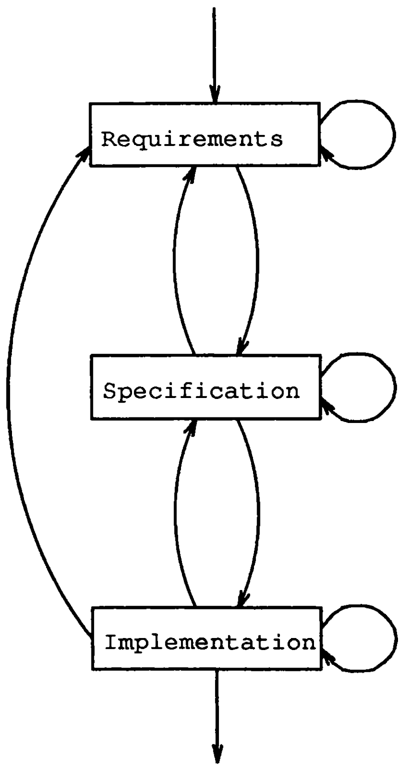


Figure 2.1: The Flow of System Development

of specification languages, and the `Implementation` is the final code of the system, written in a programming language.

What sort of verification can be carried out in each area?

2.2.1 Requirements

Since the `Requirements` are assumed to be informal and written in a natural language such as English, they are therefore ambiguous and possibly even inconsistent, so there is little to say about them in relation to formal proofs, i.e. we cannot rigorously and formally compare a formal specification with informal requirements. However, we can use *formal* requirements, which are then a form of specification, in such a proof. These formal requirements do not appear out of thin air; they must have their beginnings in informal thoughts about the system. We discuss this formalisation process below.

When the informal requirements are expressed, we expect that the user has in mind some aspects of the final implementation, therefore the informal requirements could be said to relate to a *class* of implementations. We then go through a process of trying to express the informal requirements in formal terms, thereby reducing the set of acceptable implementations by removing the ambiguity inherent in an informal description. Typically, the informal requirements may be interpreted in several different ways; therefore the formal requirements use the information in the informal requirements under one of those interpretations. One of the problems of verification lies in trying to interpret our informal notions of the correctness of a system. We look at this problem with respect to a particular example, the Login Case Study, in section 7.3. The particular informal requirement considered there is: "the implementation (of the system) satisfies the specification (of the system)". Other common informal requirements include "completeness" and "freedom from deadlock", which may also be transformed into formal statements; see the section on `Specification` below.

The derivation of formal requirements from the informal ones, i.e. requirements capture, is being studied by others, but this field lies somewhat beyond the scope of the current document.

2.2.2 Specification

The most appealing area of system development, in terms of possibilities for verification, is that of `Specification`. Given the possibility of iterations in the specification process it is useful to assume the existence of a sequence of specifications:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$$

The arrow indicates a temporal ordering of the specifications (i.e S_0 is the first specification and S_n the last) but no other relationship is assumed, i.e. S_{i+1} not necessarily derived from S_i. We expect that S_0 will be the most abstract specification, detailing *what* the system must do without saying anything about *how* these actions are to be performed, while S_n will be much more concrete, possibly providing implementation information. S_0 will probably be the first attempt at formalising the informal requirements, as described above, while S_n is the specification from which we might attempt to derive program code (depending on available techniques).

Given such specifications, the sorts of verification which can be carried out can be split into two groups:

- formally comparing two specifications, and
- proving properties of an individual specification.

We examine each of these case in more detail below.

Formal Comparison of Two Specifications

We also refer to this approach to verification as *satisfaction* because usually we want to prove that one specification *satisfies* another. A different formalism may be used for each description, for example a logic for the "specification", i.e. the more abstract description, earlier in the sequence of specifications, and a process algebra for the "implementation", i.e. the more concrete description, coming later in the sequence of specifications. In this case we need a proof technique relating the semantics of the logic to the semantics of the process algebra. This approach to verification is discussed in chapter 11. Note that the term "implementation" is used here in a different sense to its use earlier, in figure 2.1, where it denoted system code. The language used for the implementation here will be a formal specification language, and not a programming language, but we expect that the style of specification will be more concrete than that used for the other specification.

Alternatively, we might use the same language for both descriptions, but at different levels of abstraction, again allowing one to be viewed as the "implementation" of the other. Given two process algebra descriptions, there are many relations based on the observable behaviour of a system which can be used to express their equivalence, or perhaps that one is a refinement of the other. This approach to verification is discussed in detail in chapter 4.

Proving Properties of Individual Specifications

Although it is more likely that we will wish to express the correctness of a LOTOS specification with reference to some other specification, occasionally we want to prove particular properties hold of an individual specification. We note that this provides a link with the informal requirements as typically these properties will be expressed informally.

An example of a property we may desire a specification to have is *internal consistency.* For example, a change may have been made to the specification and we wish to ensure that this change does not conflict with, or contradict, the existing parts of the specification.

Another property we may desire of an individual specification is *completeness*, i.e. the specification describes all the things we want it to describe. This can be hard to express because we are asking if the specification matches an informal intuition about the system; however, we can at least put forward some guidelines to help.

Above we mention that logic may be used to specify a system. Although this will not be fully discussed until section 11, the properties which can be described by the logic may be partitioned into two classes: *safety* properties and *liveness* properties. This classification can help express the completeness of a specification because we may have some informal criteria such as "the specification should satisfy safety *and* liveness properties". Then, if the specification says nothing about liveness, this is probably an omission, since it *should* say something about liveness.

For example, the trivial process which does nothing satisfies the liveness property of termination, but obviously does no useful work. Similarly, the divergent process satisfies all safety properties of the form "when the process terminates, P holds" because it never terminates. Only the liveness property of termination forces the specification to describe useful work.

Completeness of a specification can also be partially checked by testing, e.g. to check that the specification satisfies the requirement that all classes of data are correctly dealt with, test cases must include examples from each data class. Conformance testing (see the following section on implementation) can perform this function for all classes of data expected by the specification, but not for unexpected ones.

In [RS91], the completeness of a specification of a finite state machine (which is a subclass of the machines which may be described using LOTOS) is investigated by describing the machine as a set of rules and relating the completeness of the rule set, a property well understood in term rewriting theory, to the completeness of the machine. This work is described in more detail in chapter 5.

We ignore other properties which may be desired to hold of a specification, such as robustness and performance, as they cannot be expressed in the chosen formalism and therefore cannot be verified. For example, the language must be able to express time and/or space to rate performance since this is usually measured in terms of these attributes. Also ignored is *fairness*, e.g. given a repeated choice between two actions a and b, we cannot guarantee that a will ever be performed. This is a limitation of process algebra.

Obviously the choice of formalism affects the properties we can prove hold of a specification. It can also affect the verification process: if logic is used to describe the system, a proof technique such as model checking should be applied, whereas if process algebra is used to describe the system a different proof technique would be applicable. This leads on to consideration of other issues which may affect the verification process, such as the style in which the specification is written. Varying the level of abstraction is an obvious stylistic choice which has already been mentioned. In LOTOS another obvious choice is between writing an abstract data type specification and writing a process algebra specification; the problems of verifying full LOTOS specifications are discussed in chapter 10.

We now consider what sort of verification can be carried out in the implementation stage of system development.

2.2.3 Implementation

Unlike specification, there is little to say about the `Implementation`, as this will be a bottom level implementation in a language such as C, Pascal, Ada, FORTRAN etc. Note that most of these languages do not support concurrency. This is because with OSI we expect that much of the concurrency comes from executing the protocol on different machines, i.e. distributed computing, rather than from having several processes running on one machine.

The transition from specification to implementation has been more widely researched than that from requirements to specification. Below we describe two particular areas of research which are of some interest because, although they cannot be considered as verification, they are based on a formal approach.

A number of tools have been developed which "translate" a specification language into program code, assuming that the specification is suitably concrete. The method may also require that the specification is annotated by comments indicating how something should be implemented (to give the compiler some help). An example of this, drawn from the LOTOS literature, is described in [vEKvS90]. This approach uses the relationship between the resource oriented and state oriented specification styles, see section 3.5.2, to translate LOTOS into C code.

Also of interest is *conformance testing*, i.e. checking that a given implementation conforms to its specification by applying particular tests. This allows the user to confirm that the product behaves as expected, i.e. as originally stated in the informal requirements. Note that conformance testing does not test for *robustness*, i.e. correct behaviour of the system in the presence of incorrect/unexpected inputs. This is because such tests would be impossible to derive from the specification and, while the set of conformance tests is finite, the set of robustness tests can be infinite.

In [BSS87] the notions of what constitutes a correct implementation, and how conformance tests can be used to prove that an implementation is correct, are discussed. The formalisation of these notions is based on the testing equivalences and preorders of section 3.5.3. A later work, [Bri88a], considers the question of *canonical testers*. A canonical tester is a test case, or a set of tests, such that if I conforms to S then I will pass the test, and if I does not conform to S then it will not pass the test (this last part requires repeated execution of the test since a nonconforming implementation may pass the test on some occasions, but not on every occasion). In [Bri88a], canonical testers are shown to always exist, and a method for deriving such testers from the (Basic LOTOS) specification is given. This result is only possible because of the exclusion of robustness tests. This and other methods, also for specifications in Basic LOTOS, have been implemented as tools [Wez90, Ald89].

There are situations in which conformance testing, a form of validation, is more appropriate than a verification technique. Firstly, in an industrial setting, the source code of an implementation may not be available (to check it against the specification), therefore testing is the only way to determine if an implementation meets its specification. Secondly, although verification may have been carried out at an earlier stage in the system development, errors may have crept into the final code when transforming the formal language into a programming language. Thirdly, the system may be too complex, or too large, to make exhaustive analysis practical.

The reason that techniques such as conformance testing do not make verification redundant (even though it is typically harder to analyse a specification formally than it is to apply tests to a finished implementation) is that any errors are discovered relatively late in the development of the system, by which stage correcting the error may be very expensive. Obviously, given that we *can* examine the specification of the system, it would be better if verification techniques could be applied early in the design process, catching errors before they have serious consequences. Ideally, a combination of verification and validation techniques should be used in system development.

2.3 Summary

In this chapter we have given a general definition of verification as the formal proof of correctness of a system, and have identified two broad approaches to verification of the correctness of LOTOS specifications. In the remainder of this thesis we concentrate on the former approach, i.e. expressing the correctness of a LOTOS specification by comparing it with another specification (either in LOTOS or in a modal or temporal logic). The second approach to verification, of proving properties of an individual specification, is rejected as it relies on some degree of informality in specifying the requirements.

Since this type of analysis may only be carried out on formal specifications of the system, the formalism to be used, i.e. LOTOS, must be introduced before verification techniques can be discussed formally and in more detail. This is the purpose of the next chapter.

Chapter 3

Concurrency and Process Algebra: A Survey

3.1 Introduction

Here we introduce Basic LOTOS by way of a general discussion of the features of process algebra and presentation of the related process algebras, CCS [Mil89b] and CSP [Hoa85]. This helps to give a context for LOTOS within the wider spectrum of process algebras. The reason for presenting CCS and CSP is that in order to verify a system, we need to have a firm grasp of what the meaning of the system is, i.e. its *semantics*. Presenting CCS and CSP in detail achieves two goals; the first is that their semantics are clear and simple, and therefore provide a much better introduction to the concepts of process algebra than LOTOS, which is made more complex by considerations of OSI protocols. The second reason is that a rich literature on verification in CCS and CSP exists, and, bearing in mind the close connection between these languages and LOTOS, it may be possible to adapt verification techniques developed for use with CCS or CSP for use with LOTOS instead. This latter goal in particular should be borne in mind when reading this chapter.

The reader who is familiar with process algebra can safely ignore much of this chapter. We recommend reading sections 3.3.3 and 3.4.4 which deal with proof techniques for CSP and CCS respectively, and also section 3.5 which introduces LOTOS and its associated proof techniques.

3.2 Process Algebra

In this section a particular kind of concurrency formalism is introduced, *process algebra*. Process algebras are based on the concept of *observable behaviour*, where the behaviour being observed is typically a sequence of *events*. Events are usually atomic and without duration, i.e. we abstract away from duration, therefore it is impossible to distinguish the start (or end) of an action. There may of course be intervals between occurrences of actions, so *processes* are not without duration.

The systems being observed display patterns of behaviour which are normally called *processes*. This term can be used to refer to the systems themselves, associating an object with its behaviour. The *observer* is usually taken to be some entity with a means of recording events who has the power to perform experiments on processes, either by controlling the environment in which they operate or by interacting with them.

For example, consider a vending machine[1] which can take money and dispense bars of chocolate. This can be viewed as a process which orders the events *in10p* and *outchoc* in some way, where *in10p* denotes the event of putting ten pence into the machine, and *outchoc* denotes the event of a chocolate bar appearing in the dispensing tray. Of course, other events may also occur in the life of the vending machine (e.g. refill with chocolate, empty cash box), but we have the ability to ignore events which we do not find interesting, or which are not relevant to our point of view.

The three process algebras to be discussed here all use the same (or similar) notions of events, processes, operators and environments, although these basic concepts may have different representations in each formalism. We begin by looking at some of the basic ideas common to all of the languages considered.

3.2.1 Basic Concepts and Operators of Process Algebra

In a process algebra, each process is defined inductively (or algebraically), with one or more special processes forming the base case, and a variety of operators providing the means to construct new processes. The following list of basic ideas is therefore in two parts; first the base elements of process algebra, followed by the constructors.

Events (Also called *actions*). These are the basis of the observable behaviour of our systems. Events may be specified as required for a particular task e.g. *in10p, outchoc* for a vending machine, *send_m, receive_m* for a communications protocol, and so on. They can be simple or complex, depending on the level of abstraction in the specification, e.g. the event *buychoc* could model the whole vending machine operation described above.

In addition to user defined events, some formalisms have special events built into the language. The most common special event is one which denotes internal, or unobservable, action. Internal actions can be used in specifications for a number of reasons; often they are used to explicitly introduce *nondeterminism* (see below), but they may also result from the application of other operators such as *hiding* (see below).

[1] We acknowledge a debt to Tony Hoare, who was the first to use vending machines to illustrate process algebra.

To continue with the vending machine example, an internal action may denote the coin dropping down inside the machine, or a gate opening to release the chocolate, neither of which are of particular significance to the customer of the machine and are therefore unobservable.

Other common special events include successful termination (although in some languages this is a process rather than an event, see below) and clock tick (used in languages with *time*, see section 3.2.2).

Processes As with events, processes can be user defined by associating a name with a particular behaviour, or they can be special processes, built into the language. Examples of special processes are: the broken process (also known as *deadlock*) which can do nothing, the process which can terminate successfully and then do nothing, or the process which can always do everything.

Another important feature of execution is *divergence*, i.e. a process never terminates but continues doing useless work for ever. Useless work is characterised by endless repetitions of the internal event. Although we usually do not want such processes in our specifications, divergence should appear in our concurrency theory so it may be identified and avoided. Some formalisms give this process a special name, others do not.

Another special process is the *environment*. The environment of a process can determine which events the process may perform, and can be viewed as another process interacting with the process of interest. Typically the process chosen to represent the environment is one which will allow all actions, and not hinder the progress of any other process.

In addition to the base case elements above, all process algebras include some operators, or constructors, which create new processes from the building blocks of events and special processes. The common operators are given below.

Event Prefixing This is the most basic way to construct a new process. The operator for this takes two arguments, an event, x and a process, P. The new process can be described by "do event x, then behave like process P". This operator allows the construction of sequential processes, i.e. events occur one after the other.

Choice This describes the point in a behaviour where we want to say "behave either like A or like B". A and B are called the *branches* of the choice. There are two different sorts of choice:

deterministic choice The current environment determines which branch of the choice is to be taken, depending on the initial events of A and B. Note, the initial events of A and B must be distinct. When the initial events are both allowed by the environment weak determinism results. This means the environment *observes* the choice made, but cannot influence it.

If only deterministic choices are made in a process, then, given the same environment and the same sequence of events, the process will always end up in the same state.

An example of deterministic choice may be found in the vending machine which sells chocolate and toffee. After inserting 10p, the customer, who forms part of the environment of the machine, presses one button to choose chocolate and another to choose toffee. The machine then delivers chocolate or toffee as appropriate. The machine is willing to allow either button to be pressed, and the choice to deliver chocolate or toffee is only taken after the button is pressed.

nondeterministic choice The choice between A and B is random. Given the same environment, the same choice may or may not be made again. Continuing the vending machine example, assume that instead of having buttons to allow choice between chocolate and toffee, the machine decides which to supply, based on some internal decision making procedure, which is sometimes modelled by the internal action (this makes the two branches initially look the same and the choice is made nondeterministically). To the customer, the machine has made an internal choice as to what sort of sweet to supply, chocolate or toffee; the customer does not participate in the decision making.

When implementing nondeterministic choice, it may be further classified as *angelic*, *demonic* or *erratic*. The difference between these two types of nondeterminism is best demonstrated by example.

In a choice between A and B let A be a process which takes a coin, gives a chocolate then stops (deadlocks), and B be a process which takes a coin, gives out a chocolate and returns to its initial state. Angelic choice will never pick A, always B (because A deadlocks, but B does not).

This sort of choice is also known as *external* choice, meaning that the only actions which may proceed are those allowed by the environment. In the implementation of angelic choice both branches are followed until some event causes one branch to deadlock. This forces execution to commit to the other branch of the choice (although it may deadlock at a later stage)[2]. This form of choice is in some sense stronger than deterministic choice (in the case in which both actions are possible), as it has the ability to look beyond the first action when making its choice.

Demonic choice operates in much the same way, except that the bad branch is chosen, i.e. the one which deadlocks. Demonic nondeterminism is also known as *internal* choice since it depends only on the process — the environment has no influence.

[2]In an environment where neither branch leads to deadlock we must rely on some other implementation of nondeterminism, e.g. tossing a coin!

Obviously, both forms of choice lead to inefficient implementation as extra computations must be carried around when there are unresolved choices. A more efficient alternative is erratic choice.

In the implementation of erratic choice the decision of which branch to follow is made straight away, with no reference to the events in each branch, initial or otherwise. The choice is random, and each branch is equally likely to be chosen. In the example above, there is a 50% chance that the deadlocking branch A will be chosen.

Some notations have separate operators for deterministic and nondeterministic choice, while others have only one which can behave in either way, depending on the circumstances. Typically, such general choice operators behave deterministically when the initial events of the choice are distinct, and nondeterministically when the initial events of the choice are the same, or one is the internal event.

Parallelism One of the most important features of concurrent systems is that processes may be observed interacting. In order to achieve this, the processes must be executed at the same time, i.e. in parallel. There are two sorts of parallelism: *true parallelism* (events may occur simultaneously) and *interleaving parallelism* (only one event occurs at any point in time, but the order in which the events occur is unknown). Although true parallelism is the more powerful of the two approaches (since any the result of any computation obtained under interleaving semantics may also be obtained under true parallelism semantics), interleaving is a more commonly used semantics for process algebra. This is because interleaving is simpler, or more easily understood, than true parallelism and has nicer algebraic properties (such as being able to express parallelism in terms of event prefixing and choice), while retaining some level of nondeterminism in the ordering of events. Some languages may have operators which *specify* interleaving, taking precedence over the true parallel semantics if necessary.

To illustrate the difference between true parallelism and interleaving parallelism, consider the following example. Take two processes A and B. Let A be the statement $y := y + 1$, and B be the statement $y := y - 1$. The initial conditions are $y = 1$. If A and B are performed in parallel there are two possible results depending on the kind of parallel semantics employed. With interleaving semantics, the result will be $y = 1$, since A before B gives $y = 1$, and B before A also gives $y = 1$. On the other hand, true parallel semantics will yield a *set* of results $y = \{0, 1, 2\}$. This is because, in addition to the execution scenarios given above for interleaving semantics, it is possible that A and B begin execution at exactly the same time. Each process may read the value of y at the same time ($y = 1$), but the result depends on which statement overwrites the value of y last. Obviously true parallelism is more powerful than interleaving semantics, since the result set obtained by interleaving semantics is a subset of the the result set obtained by true parallelism. True parallelism may also give a more realistic model of the world than interleaving semantics. However, formalisms are often a simplification of real world situations, as the full complexity of the situation may be too great to allow analysis.

Another variation of parallelism to consider here is whether the actions of the language occur *synchronously* or *asynchronously*, i.e. actions of parallel processes are performed in lockstep, or actions of parallel processes are allowed to start and stop at different times. Synchronous calculi are interesting because they are more powerful than their asynchronous counterparts (asynchrony can be expressed by introducing a wait operator to the synchronous calculus), but they are less used because the idea of every process executing according to some global clock does not intuitively relate to our notion of distributed systems, where each part proceeds at its own speed. This problem, the conflict of power and popularity, was mentioned in section 1.1 when we discussed which formalism to adopt for our study.

Now we can express two processes executing at the same time, we also need to consider different forms of interaction, or *communication.*

Communication Communication is a simple phenomenon which, like parallelism, comes in several varieties. Given two processes A and B which wish to communicate, the most obvious form of communication is *message passing.*

Message passing provides a means to exchange information between processes; rather like input/output. Events which perform message passing are usually called *channels.* Message passing can be synchronous or asynchronous. These two forms of communication can be illustrated by simple, everyday, examples. Synchronous communication is like a telephone conversation (both parties engage in the communication at the same time), while asynchronous communication is more like writing a letter (each party is active at different times, but not necessarily so). Note that synchronous message passing does not necessarily entail a synchronous calculus, similarly for asynchronous message passing.

The difference between these two forms of communication may be understood by considering the type of communication medium being modelled in each case. Synchronous message passing models communication over a wire, i.e. without a buffer, while asynchronous message passing models communication over a buffer. In synchronous message passing, if one process is ready to send a message it has to wait until the other process is ready to receive that message. Asynchronous communication allows the sending process to deposit its message in a buffer and continue working; the receiving process may come and get the message when it is ready. Obviously the reverse is not true; the receiving process cannot pick up a message before it is available.

A special case of synchronous message passing occurs when the messages are empty. For examples, if both processes can perform event x, say, we require that they perform x at the same time, with the occurrences of x being somehow merged. Hence the observer sees just one occurrence of x, but it was performed by both A and B. This sort of communication serves only to synchronise the progress of the processes involved, i.e. if process A is first to reach the point where it should execute x, it is forced to wait until B is also ready to execute x before it may proceed. Communication of this type is referred to as *synchronisation.*

Some process algebras restrict communication to be between two processes only; however, communication can be generalised to multi process interaction. When more than two processes are involved, message passing is known as *broadcasting*, or *multi-way synchronisation* (for empty messages).

Hiding/Restriction [3] Hiding takes place implicitly when the observer decides which events are important/to be observed (all others are hidden), but the concurrency formalism may also provide a mechanism for explicitly hiding certain events from the observer. Hiding of events is typically achieved by transforming the events to be hidden into occurrences of the internal event, which is unobservable. Hidden events may proceed instantaneously.

An alternative to hiding, *restriction* specifies a list of events which may *not* occur, rather than a list of events which occur but cannot be observed. Restriction is often used to force processes to communicate with each other by refusing to allow communication with any other processes in the environment.

Recursion This allows us to describe repetitive behaviour patterns that may continue indefinitely.

Above we have described some of the most common process algebra operators. There is one more fundamental concept which has yet to be discussed; this is the area of *Observations, Semantics and Equivalence Relations.*

The concept of observable behaviour is fundamental to process algebras. As described above, the semantics of a system is given by the actions it takes, and patterns of those actions. It is also important to realise that the semantics of a system may change depending on our notion of what can be observed. This may allow us to identify more processes (to distinguish fewer), or to identify fewer processes (to distinguish more processes). This is appropriate because the modelling of different systems may call for different notions of what is important and needs to be observed, and what can be safely ignored.

[3]This is a rather simplistic view of how hiding and restriction work. For more details see section 3.3.1 and section 3.4.1 respectively.

Another way of altering which processes are identified is to define equivalence relations over the structure of the processes. This is just another method of saying which aspects of process behaviour are observable. Usually, each process algebra has a particular equivalence relation with which it is most often associated, although it may also be associated to a lesser extent with other equivalences. The topic of equivalence relations will be discussed in more detail in sections 3.3, 3.4 and 3.5 where reference can be more easily made to specific relations and process algebras. We also discuss the relationships between different equivalence relations in section 3.6.2.

Those features of concurrency which are common to many process algebras have been given in this section, but each process algebra will have its own interpretation of the operators, and probably some special features not discussed above. There is a large body of work on extending the capabilities of process algebras to make them capable of specifying real world phenomena more accurately. Of course, this also has an effect on verification in terms of being able to express different properties, e.g. a timed language can express properties relating to performance and efficiency. We may also have to develop new verification techniques to allow us to determine whether or not our system possesses these properties. The next section briefly discusses the sort of extensions which have been proposed in the literature.

3.2.2 Extensions to Process Algebra

Extensions to process algebras fall into two broad classes: those which are merely notational, and are intended to simplify the specification of complex systems by giving the specifier more operators, and those which are more fundamental, requiring an extension of the underlying model. An example of the first type of extension (although it is not generally viewed as an extension) is the parallel operator in an interleaving semantics. Although we presented this operator as a basic constructor of process algebras, it is actually redundant, as any expression using parallelism can be rewritten using action sequencing and choice. It should be noted however that the huge specifications resulting from the removal of parallelism would be almost impossible to read and understand mainly because of their size and lack of structure. Other examples of the notational type of extension may be found as part of the language descriptions in sections 3.3.1 and 3.5.1.

These forms of extension only affect verification if we use a proof technique which relies on the syntactic form of the process. More often we will use some form of pre-processing which reduces the process expression to its simplest form, using only basic operators, and apply the proof technique to that expression. A far greater impact may be made on the verification process by the introduction of the second type of extension which can alter the underlying model of the language.

The second type of extension lies somewhat outside the scope of this survey, but it is important to know of the existence of such developments and of the form they may take. To this end, some possible extensions to process algebra are given below. We present only those extensions which have been applied to LOTOS, giving LOTOS specific references in each case. Other extensions, such as priority weighting for choice, which do not have LOTOS applications have been ignored.

Time The formalisms considered later in this chapter order the occurrence of events but do not include an explicit notion of time. However, the specification of real-time systems requires a more complex model of time.

There are several different approaches to the introduction of time: some languages introduce a special operator which signifies the passing of time, others introduce a global clock. A third method of adding time to a language is to make actions more complex by adding a time parameter to the actions, e.g. action a takes 0.2 seconds. An example of a language which uses time explicitly in this way is Timed LOTOS [QAF90, QFA89, RvB91] which adds date stamps to its actions.

Data Constructs Most existing process algebras deal only with simple data types. Obviously to describe more complex concurrent systems more complex data types may be required. The ability to describe such data types has been added to some languages by the introduction of an abstract data type sublanguage to describe the actions. For example, full LOTOS is obtained by adding the ACT ONE data type language to the process algebra Basic LOTOS.

Mobility A different approach to the introduction of data types to a process algebra is to extend the sort of messages which can be passed between processes to include channel names. This allows dynamic reconfiguration of processes, giving the ability to model complex data types and higher order process algebra. This approach is basic to the π-calculus [MPW92]. This extension has been modelled in LOTOS in [FO91].

Probability The branches of a choice can be weighted by adding probability to the language. For example, in a choice between a and b, we can say action a is more likely to occur than action b. For LOTOS, this extension has been described in [RvB91] and [MFV89].

Further examples of extensions and approaches to extensions both generally, and more specifically for LOTOS, can be found in the proceedings of conferences such as CONCUR, e.g. [CONCUR], PSTV, e.g. [PSTV] and FORTE, e.g. [FORTE].

The additions of such extensions can have important repercussions in the semantics and proof theory of a language, and this in turn affects verification. Typically proofs become more complicated; we shall see examples of this in chapter 10 where proofs in full LOTOS, i.e. with data types, are considered. Initially, to avoid such complications, we concentrate on the simpler Basic LOTOS, i.e. with no extensions other than the notational kind.

3.2.3 Properties of Specification Languages

In section 3.2.1 the features specific to concurrency formalisms were considered; such languages should also have the features we desire of *any* specification language.

In [Bri88b] some basic criteria are presented which a good formalism should satisfy. A formal description technique (fdt) should be clear and consistent, have appropriate mechanisms for providing the specification with structure, be able to specify all aspects of the system under consideration (at a sufficiently high level of abstraction) and should encourage the specifier to write unambiguous specifications (of course, the language's own semantics should also be without ambiguities). These basic criteria can make a specification easier to reason about.

A concurrency specification language may also have special needs as regards verification. We may want to analyse any of the following properties: freedom from deadlock/livelock, fairness, correct allocation/deallocation of resources, mutual exclusion, equivalence between processes etc. Our modelling of the concurrent system should facilitate specification and verification of these properties. It is possible that such properties may be better checked by some formal system which can be used in conjunction with our process algebra, rather than by using a more complex version of the process algebra.

We move on now to present the three process algebras CCS, CSP and LOTOS in more detail. Although LOTOS was developed from CCS and CSP there are many differences in approach between the formalisms. LOTOS was developed for industrial use, whereas CCS was developed for research purposes. CSP has a foot in both camps as it was intended to be a usable language for large scale development (the programming language occam™ is based on CSP) but is also commonly used for research.

A further contrast between the languages chosen is evident in the approach to their definitions. We can split process algebras into two groups: those which are based on a particular model and are presented with laws which are true in that model (e.g. CSP and LOTOS), and those which are calculi of rules and axioms and can be presented independently of any mathematical model (e.g. CCS).

The languages chosen are discussed individually in the following three sections. In each section we present the basic ideas behind the design of the language, the operators which express that language's particular flavour of concurrency, the form of semantics most commonly associated with each formalism (but not the full language semantics), and some of the proof techniques which may be used in that semantic framework. Specific implementations of techniques and tools for particular languages are *not* discussed here; automated proof techniques are considered in chapter 4, and a survey of verification tools, mainly those for CCS, and of LOTOS tools in general, may be found in appendix A.

We begin by presenting the process algebra CSP. Alphabetically, CCS should be considered first; however, CSP has a simpler mathematical model.

3.3 CSP

The most important design decision behind CSP [BHR84, Hoa85] was to have a single, simple model in which as many processes as possible were identified. This decision arose from the following aims: the language must

- be able to describe a wide range of applications,
- admit efficient implementation,
- give programmer support in all stages of development, i.e. specification, design, implementation, verification and validation.

These aims result in a large number of operators, each corresponding to one concept in concurrency theory, giving the programmer flexibility. The designers also adopted the principle of indiscernibles: only observably different processes are distinguished, all others are identified. This model yields a rich set of algebraic laws, allowing flexible transformation and optimisation of CSP processes.

3.3.1 Operators of CSP

A core set of CSP operators are given in figure 3.1, in which a and b denote events, c a channel (a communication event), P and Q processes, and A and C sets of events. We use x to range over events, $P(x)$ for the set of processes parameterised by x, v and w to range over some data set e.g. integer, X to range over processes and $F(X)$ to denote a guarded expression containing the process variable X.

Features of CSP concurrency to note are:

alphabet An important part of CSP is the *alphabet* of a process. The alphabet of a process is the set of events in which the process may engage. This may be explicit (as a subscript to the name of the process, or defined separately), or implicit (can be deduced from the process description).

environmental choice This form of choice is intended to be deterministic, so we require $a \neq b$ in the expression $(a \rightarrow P) \mid (b \rightarrow Q)$.

general choice This is a combination of environmental choice and nondeterministic choice. If the initial actions of P and Q are distinct then it behaves like $\mid$, otherwise it behaves like $\sqcap$.

description	notation
deadlock	STOP
divergence	CHAOS
action prefixing	$a \rightarrow P$
deterministic choice based on actions (binary)	$(a \rightarrow P) \mid (b \rightarrow Q)$
(and over sets of actions)	$(x : A \rightarrow P(x))$
general choice over processes	$P \,[]\, Q$
demonic nondeterministic choice over processes	$P \sqcap Q$
parallelism	$P \parallel_A Q$
interleaving	$P \,\vert\vert\vert\, Q$
communication event (input)	$c?w$
communication event (output)	$c!42$
hiding of events	$P \setminus C$
recursion	$\mu X : A.\ F(X)$

Figure 3.1: CSP Operators

parallelism The parallel operator may have its synchronisation set explicitly specified, as in $P \parallel_A Q$, meaning that only events in the specified set, A, may interact. Otherwise, the synchronisation set of the parallel operator is taken to be the intersection of the alphabets of the processes, i.e. all possible events interact.

In a CSP parallel expression two or more occurrences of the same event synchronise i.e. multi-way synchronisation. Synchronisation is compulsory in that if an event belongs to the specified synchronisation set then it may not proceed independently.

interleaving Interleaving is parallelism where the synchronisation set is empty, i.e. no synchronisation/message passing at all.

communication For example, occurrences of $c?w$ and $c!42$ synchronise, assigning the value 42 to w. Message passing is treated slightly differently from synchronisation. Convention dictates that while synchronisation in general is multi-way, channels may be used for two process communication only, although this is not enforced by the semantics.

hiding The expression $P \setminus C$ means the process P with all occurrences of events in C hidden, i.e. the events may still occur, and in fact they occur automatically and instantaneously, but they may not be observed by the environment. The alphabet of $P \setminus C$ is therefore the alphabet of P minus events in C.

recursion In the definition of a recursive process $F(X)$ must be a guarded expression, otherwise the recursion degenerates to divergence. A guarded expression is one in which the occurrence of the process variable is prefixed by at least one action. This action must not be hidden.

In view of the aims of the designer (in particular, that CSP should be applicable to large scale systems), there are a number of additional operators in CSP which are more apt for specifying real protocols and applications. These include $\surd$ (the successful termination event), sequential composition of processes, interrupts, pipes and various features borrowed from imperative languages such as assignments, conditionals and loops.

3.3.2 Semantics of CSP

As mentioned in the introductory section, most formalisms have several different semantics. The following section gives the strongest and most commonly used of the CSP semantics, *Failures-Divergences* semantics. Two other semantics are frequently used in connection with CSP: *Trace* semantics and *Failures* semantics. These are also detailed below.

Standard Semantics

In CSP, a process is uniquely represented by its *alphabet*, its *failure set* and its *divergence set.* This representation of a process is known as *Failures-Divergences* semantics. The alphabet has already been mentioned. The definition of the failure set is a little complicated; first the terms *trace* and *refusal* must be defined.

A trace is a sequential record of the observable behaviour of a process. It may be viewed as a string and shares many of the standard string operations such as concatenation. In this section the following notation is used:

- $\langle\rangle$ denotes the empty trace.
- $\langle a\rangle$ denotes the trace containing one occurrence of the event a.
- $\langle a, b, c\rangle$ denotes the trace a then b then c.
- A^* denotes the set of all possible traces using events in A. We use σ to range over A^*.

Each process may have many possible traces as a result of choices in execution.

To define a refusal, let X be the set of events offered by the environment of a process, P. If P can deadlock on its first step when placed in this environment, then the set X is known as a refusal of P. We combine traces with refusals to obtain the failure set of a process, P, which contains all pairs $\langle\sigma, X\rangle$, where σ is a trace and X is a refusal, such that P may deadlock after trace σ in the environment offering events X.

The divergence set of a process is a set of traces such that, after performing any one of these traces, the process will behave chaotically, i.e. it is impossible to determine which events will occur.

The semantics of any CSP process is given by these three sets. As an example consider the two special processes:

$$\mathrm{STOP}_A \overset{def}{=} (A, \{\langle\rangle\} \times \mathbb{P}(A), \{\}) \tag{3.1}$$
$$\mathrm{CHAOS}_A \overset{def}{=} (A, (A^* \times \mathbb{P}(A)), A^*) \tag{3.2}$$

where $\mathbb{P}(A)$ denotes the powerset of A.

The first component of the triple in equation (3.1) says that the alphabet of STOP_A is A, some arbitrary set of events. Note the use of a subscript A to make the alphabet explicit in the name of the process; if unspecified the alphabet is assumed to be all events. The first component of equation (3.2) is similar. The second component of equation (3.1) says that STOP_A can refuse to do any subset of A after it has performed the empty trace, i.e. it can refuse to do everything before it does anything, while the second component of equation (3.2) says that CHAOS_A can refuse to do any subset of A after any trace, i.e. at any time it can refuse to do any event. The remaining component of equation (3.1) says that STOP_A has an empty divergence set, meaning STOP_A never diverges, while the last component of equation (3.2) says that CHAOS_A may diverge after every possible trace.

We give a further example of the Failures-Divergences semantics of a process below.

Example Consider a vending machine, *VM*, which either takes 5p, gives a chocolate and stops, or takes 10p, gives a toffee and stops. This machine can be represented by a process in the following way:

$$\begin{aligned}
&VM \overset{def}{=} (\mathit{in5p} \longrightarrow \mathit{outchoc} \longrightarrow \mathrm{STOP}_{\alpha VM} \mid \mathit{in10p} \longrightarrow \mathit{outtoffee} \longrightarrow \mathrm{STOP}_{\alpha VM})\\
&A \ \ \overset{def}{=} \{\mathit{in5p},\ \mathit{in10p},\ \mathit{outchoc},\ \mathit{outtoffee}\}\\
&VM \overset{def}{=} (A,\ \{\langle\langle\rangle,\ \mathbb{P}(\{\mathit{outchoc},\ \mathit{outtoffee}\})\rangle,\ \langle\langle \mathit{in5p}\rangle, \mathbb{P}(A - \{\mathit{outchoc}\})\rangle,\\
&\qquad \langle\langle \mathit{in10p}\rangle, \mathbb{P}(A - \{\mathit{outtoffee}\})\rangle,\ \langle\langle \mathit{in5p}, \mathit{outchoc}\rangle,\ \mathbb{P}(A)\rangle,\\
&\qquad \langle\langle \mathit{in10p}, \mathit{outtoffee}\rangle,\ \mathbb{P}(A)\rangle\},\ \{\})
\end{aligned}$$

The alphabet, A, contains the four events of interest, *in5p, in10p, outchoc* and *outtoffee.* The process may deadlock if, after doing trace $\langle\rangle$, it is only offered *outchoc* and/or *outtoffee.* Once the machine has accepted a five pence, i.e. $\langle \mathit{in5p}\rangle$, it can only perform an *outchoc* event, refusing all others, similarly for *in10p.* Finally, after the process has performed $\langle \mathit{in5p},\ \mathit{outchoc}\rangle$, it becomes incapable of any further action (similarly with $\langle \mathit{in10p},\ \mathit{outtoffee}\rangle$). *VM* never diverges.

Alternative Semantics for CSP

The semantics given in the previous section is not the only possible semantics for CSP. We now describe two other weaker CSP semantics: *Trace* semantics and *Failures* semantics.

In trace semantics a process is represented by its *trace set.* For example, let p be a process, then $traces(p)$ is defined to be the set of all possible traces of p. Two processes are equivalent if their trace sets are the same. This equivalence on processes is similar to the language equivalence used for finite state automata. It is the weakest possible semantics for CSP and generally an unsatisfactory one as it does not preserve deadlock properties, i.e. two processes may be equivalent under trace semantics even if one deadlocks and the other does not. Trace semantics is popular because it is simple, and easily described and understood. For verification purposes a stronger equivalence is generally required.

The next step up from trace semantics is *Failures* semantics. In this semantics a process is represented by its alphabet and its failure set (as described above). Two processes are then equivalent if their failure sets are the same. This semantics cannot detect divergence.

3.3.3 Proof Techniques for CSP

Proofs in CSP can be carried out using two methods: manipulation of the specification by using the algebraic laws, or proving certain properties hold of the semantics of a particular process. This has been compared with the way in which proofs in boolean algebra may be performed [BBH$^+$91].

As mentioned earlier, because of the simple model of CSP and the proliferation of operators, the algebraic theory is fairly rich. The algebraic laws can transform process algebra statements so that they are more efficient, or so that they may be more easily verified. We will not present all the laws here: a full treatment is given in [Hoa85]. Instead we mention a few of the main laws. For example, $\|$, $\sqcap$ and $[]$ are all associative and commutative, $\sqcap$ and $[]$ are also idempotent. STOP is a zero for $\|$, but an identity for $[]$, while RUN is an identity for $\|$ (RUN is the process which may do anything/everything, but never blocks an action, unlike CHAOS). We also have $\sqcap$ and $[]$ distribute over each other, and $\rightarrow$ distributes over $\sqcap$. Note that this law $a \rightarrow (P \sqcap Q) = (a \rightarrow P) \sqcap (a \rightarrow Q)$ is one of the main distinctions between CSP and CCS (it does not hold in CCS).

The alternative proof technique is to define desirable properties in terms of traces (and failures and divergences if necessary) and use a relation **sat** which relates process algebra expressions to the CSP semantics. We write P **sat** S, where P is a process and S is a specification of properties, meaning P satisfies S, i.e. all possible observable behaviours of P are consistent with by S. More formally, $\forall \mathrm{tr}.\ (\mathrm{tr} \in \mathit{traces}(P) \Rightarrow S)$. These specifications allow us to write more abstract descriptions of systems, specifying *what* should be achieved, rather than *how* to do it.

For example, in a vending machine, we may want to specify that pairs of *in10p* and *outchoc* events always match up, i.e. the machine never takes money without giving a chocolate, and it never gives chocolates without taking money. This can be done by stating that the number of *in10p* events in a trace must be equal to or one more than the number of *outchoc* events.

We move on now to consideration of CCS.

3.4 CCS

The aim behind the design of CCS [Mil80, Mil89b] was to provide a means of investigating different models of concurrency. Each model uses the same set of operators, but a different notion of equivalence between processes. Unlike CSP, the operator set of CCS is very compact, with each operator combining elements of different concurrency concepts. For example, deterministic choice and nondeterministic choice are denoted by the same operator. Despite the small size of the operator set, CCS has a high level of articulacy and generality, i.e. many different kinds of system may be described, at many different levels of abstraction.

3.4.1 Operators of CCS

The operators of CCS are given in figure 3.2, in which P and Q denote processes, a an event and L a set of event names. X ranges over processes.

description	notation
internal action	τ
inactive process (deadlock)	0
action prefixing	$a.P$
choice	$P + Q$
parallelism	$P \mid Q$
restriction	$P\backslash L$
recursion	$\mathbf{fix}\ X = P$

Figure 3.2: Operators of CCS

Because of the difference in approaches between CSP and CCS, there are some important differences between the operators:

- τ, the special internal action, signifies the occurrence of an internal event without giving details as to what that event is. It can be used explicitly by the specifier, but it also results from communication between processes (see below).
- There are two sorts of actions, the simple actions and their overbarred complements. Communication occurs between an action and its complement. When such a communication occurs the resulting action is τ, i.e. given a and $\bar{a}$, the result of their communication is τ and both a and $\bar{a}$ are hidden. It is convenient to view these complement pairs as input/output pairs.
- Communication and value-passing can occur between only two processes, unlike CSP which allows broadcasting.
- CCS has only one parallel operator. There is no synchronisation set associated with this operator, so it is possible for all events to synchronise, but synchronisation/communication

is not compulsory and need not take place. The restriction operator, see below, may be used to force communication between certain processes by denying communication with other processes.

- CCS has only one choice operator, which is deterministic but can be forced to be nondeterministic by prefixing each branch by the same event, or by prefixing just one branch by the special event τ.
- Restriction prevents other processes in the environment communicating with P through actions in L. Actions in L may occur within P (assuming P is a complex process composed of other processes in parallel) if they can synchronise with their complements (since synchronisation results in a τ action, which cannot be restricted). Unlike CSP hidden actions, restricted actions may only proceed by communication and may not proceed independently.

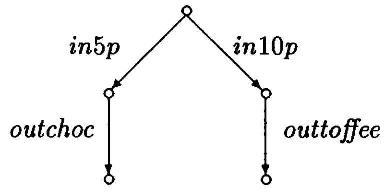
Extensions of the basic calculus introduce operators which make descriptions of real systems more convenient, but these operators introduce no extra power. They are not described here since they are similar to the extra operators introduced for CSP and will not be used in the following discussion of the equivalence relations defined over CCS.

The following sections introduce the semantics of CCS processes and discuss some of the equivalence relations which have been defined in the CCS literature.

3.4.2 Semantics of CCS (Informal)

The operators of CCS are defined in terms of *labelled transition systems.* Each process is seen as a set of states, with arcs between states representing actions which move the process from one state to another. The unobservable action, τ, moves a process silently from state to state. Loops from a state to itself are possible. A set of inference rules give an operational definition of CCS processes in terms of labelled transition systems.

Example Consider again the vending machine *VM* presented in section 3.3.2. The labelled transition system for *VM* can be rolled out into the form of a tree, which looks like this:



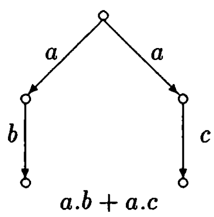
Labelled transition systems are often represented as trees, or process graphs, since pictures are usually understood more quickly and easily than a mathematical equation. The switch from one domain to the other is normally made without comment.

Now consider the notion of equivalence between processes/labelled transition systems. As with CSP, there are several different equivalences for CCS. The choice of equivalence is important in system verification (where we might want to prove equivalence between two processes) since processes can be equivalent in one model, but not equivalent in another. For more details of how the equivalences relate to each other, see section 3.6.2.

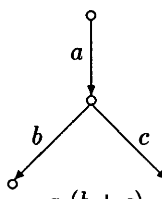
Of the many equivalences which can be defined for CCS, only four will be detailed here.

Description of CCS Equivalences

To motivate the different equivalences, we first include some examples of processes, taken from [Mil89b], which we would *not* like to be identified.



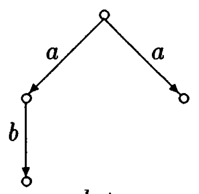
$a.b + a.c$



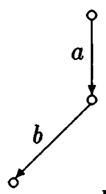
$a.(b + c)$

The reason for distinguishing these processes is that, after performing an a event, the left hand process may deadlock when offered b, i.e. the right hand branch has been taken, whereas the right hand process will never deadlock if offered b after performing a. Note that these processes are equivalent under all three of the CSP semantics. This example demonstrates the invalidity of the distributive law of . over + in CCS.

We also want to distinguish the following processes:



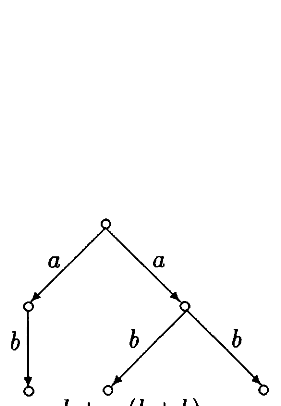
$a.b + a$



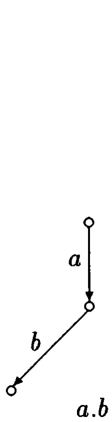
$a.b$

The left hand process may do an a and then fail when offered b. Again, this is because the right hand branch has been taken. The process on the right will always do a then b then stop.

This may lead us to the conclusion that we only want to identify processes which have exactly the same branching structure, i.e. tree equivalence, but this is too strong. As a further example, here are two processes which we generally wish to equate:



$a.b + a.(b + b)$



$a.b$

These processes cannot be distinguished by their behaviour: they both perform a then b and then stop. The criteria we use in identifying processes is that they should be equated if they exhibit the same behaviour under all tests. These tests are constructed with respect to three conditions:

1. The choice of transition at any moment is determined by the *environment* (nondeterministic choice is used in the case of ambiguity, e.g. same first event in each branch);

2. The environment has only finitely many states — at least as far as choice-resolution is concerned;

3. We can control the environment.

Essentially, these criteria mean that to test a process, a copy of the process is executed. When a branch point is reached, duplicate copies of the state reached are made, one for each possible branch, and execution continues. This means that for any process we know *all* the behaviour of that process, and have a record of the places in the process where (significant) choices are made. This last point is most important: often it is the differences between the choice points which distinguish processes. See, for instance, the first example of this section.

Two equivalences are obtained using this view of the testing:

- If τ is viewed in the same way as all the other actions, the equivalence obtained is *Strong Equivalence*, also known as *Strong Bisimulation Equivalence.* This is the strongest equivalence used in conjunction with CCS.

- If τ is given its special status as the unobservable action, it cannot be used to distinguish between processes. The equivalence obtained is *Observation Equivalence*, also commonly referred to as *Weak Bisimulation Equivalence*, or just *bisimulation* equivalence in the literature.

Although these equivalences are nice to use because of their simple definitions, they have been criticised for their artificiality, i.e. in reality we may not be able to control the environment, expect it to have a finite number of states, or be able to calculate all the states. This leads us to a third equivalence known as *Testing* equivalence in which two processes are equivalent if they "pass" the same tests (where pass can have different interpretations). On the other hand, criticism is also made of observation equivalence for making distinctions which are not truly observable. *Branching Bisimulation Equivalence* was developed to correct these deficiencies.

The formal definitions for these equivalences are given in the next section.

3.4.3 Semantics of CCS (Formal)

Before defining the equivalences of CCS, we first need to make some auxiliary definitions. All definitions, including those of particular equivalence and congruence relations given elsewhere in this section, are taken from [Mil89b] unless otherwise specified.

Definitions

- The domain of labels is called $\mathcal{L}$. This consists of names $\mathcal{A}$ and co-names $\bar{\mathcal{A}}$. We use a, b, c, $\ldots$, $\bar{a}$, $\bar{b}$, $\bar{c}$, $\ldots$ to range over $\mathcal{L}$ and $t, t_1, \ldots$ to range over $\mathcal{L}^*$.

- The domain of actions is called *Act*. This is defined to be $\mathcal{L} \cup \{\tau\}$. We use α, β, γ, $\ldots$ to range over *Act*, and σ to range over *Act**. The empty string is denoted ϵ.

- The domain of CCS processes, called $\mathcal{P}$, is the set of ground process expressions. We use P and Q to range over $\mathcal{P}$.

- A labelled transition system is a 4-tuple $(S,\ Act,\ \{\xrightarrow{\alpha} \subseteq S \times S\}, s_0)$, which consists of a set S of *states*, a set *Act* of *transition labels*, a *transition relation* $\xrightarrow{\alpha}$, one for each $\alpha \in Act$, and a starting state $s_0 \in S$. The transition relation determines how we get from one state to another, and is defined by the inference rules which give an operational semantics of CCS. The name of a CCS process is identified with the starting state of its labelled transition system by an abuse of notation.

- $P \xrightarrow{\alpha} Q$ means that process P performs the event α and then behaves like process Q. This can be extended to strings, so $P \xrightarrow{\sigma} Q$, where $\sigma = \langle a, b, \bar{c} \rangle$, is an abbreviation for $P \xrightarrow{a} P' \xrightarrow{b} P'' \xrightarrow{\bar{c}} Q$, for some intermediate states P' and P''.

- $P \overset{\alpha}{\Longrightarrow} Q$ means that P performs some string of events $\tau^i \alpha \tau^j$ for some $i, j \geq 0$ before behaving like Q. Again, this relation can be extended to strings as above.

- We write $\hat{\sigma}$ to denote the string σ with all occurrences of τ removed.

- $P \overset{\hat{\alpha}}{\Longrightarrow} Q$ means the same as $P \overset{\alpha}{\Longrightarrow} Q$ for all $\alpha \neq \tau$. For $\alpha = \tau$ the behaviour is slightly different: $P \overset{\tau}{\Longrightarrow} Q$ means $P \xrightarrow{\sigma} Q$ where $\sigma = \tau^j$ for $j \geq 1$, while $P \overset{\hat{\tau}}{\Longrightarrow} Q$ denotes $P \xrightarrow{\sigma} Q$ where $\sigma = \tau^j$ for $j \geq 0$, i.e. it is possible that $\sigma = \epsilon$.

In simple terms what these definitions mean is that, for all events in $\mathcal{L}$, the three kinds of arrow behave in exactly the same way; their behaviour differs only for τ actions. For $\sigma \in Act^*$, the arrows describe the *labels* in σ exactly, and their behaviour on the τ actions is as follows: $\xrightarrow{\sigma}$ describes exactly the τ actions in σ, $\overset{\sigma}{\Longrightarrow}$ at least the τ actions in σ, and $\overset{\hat{\sigma}}{\Longrightarrow}$ says nothing about τ actions. The definition of these different types of arrows allows us to define the different bisimulations and equivalences in a similar manner.

Strong Equivalence

Strong equivalence is described in terms of a property of relations over processes, called *strong bisimulation.* The notion of bisimulation was first introduced by Park in [Par81].

Definition 1 (Strong Equivalence) *A relation, $R \subseteq \mathcal{P} \times \mathcal{P}$, is a strong bisimulation if $(P, Q) \in R$ implies, $\forall \alpha \in Act$*

1. *if $\exists P' : P \xrightarrow{\alpha} P'$ then $\exists Q' : Q \xrightarrow{\alpha} Q'$ with $(P', Q') \in R$, and*
2. *if $\exists Q' : Q \xrightarrow{\alpha} Q'$ then $\exists P' : P \xrightarrow{\alpha} P'$ and $(P', Q') \in R$.*

Two processes P and Q are strongly equivalent, written $P \sim Q$, if there exists a strong bisimulation R such that $(P, Q) \in R$. The relation $\sim$ is defined to be the largest strong bisimulation, i.e. the union of all strong bisimulations.

A certain amount of confusion arises from the common use of *strong bisimulation* to describe the equivalence as well as the relational property. We will try to avoid such confusion.

There are a number of alternative characterisations of strong equivalence, corresponding to the alternative characterisations of observation equivalence given below, with the exception of HML. The details for strong bisimulation equivalence are not given here as they are similar to those given for observation equivalence.

Observation Equivalence

As with strong equivalence, observation equivalence is also defined in terms of a property of relations over processes. This property is called *weak bisimulation*, also referred to as *bisimulation.* This is similar to strong bisimulation, as the name suggests, only differing in its treatment of the unobservable τ action.

Definition 2 (Observation Equivalence/Weak Bisimulation Equivalence) *A relation, $R \subseteq \mathcal{P} \times \mathcal{P}$, is a (weak) bisimulation if $(P, Q) \in R$ implies, $\forall \alpha \in Act$*

1. *if $\exists P' : P \xrightarrow{\alpha} P'$ then $\exists Q' : Q \overset{\hat{\alpha}}{\Longrightarrow} Q'$ with $(P', Q') \in R$, and*

2. *if $\exists Q' : Q \xrightarrow{\alpha} Q'$ then $\exists P' : P \overset{\hat{\alpha}}{\Longrightarrow} P'$ with $(P', Q') \in R$.*

Two processes, P and Q, are observation equivalent, written $P \approx Q$, if $(P, Q) \in R$ for some (weak) bisimulation R. The equivalence, $\approx$, is the union of all weak bisimulations, and hence the largest weak bisimulation.

The essence of this equivalence is that τ actions may be ignored in determining the equivalence of two processes; only visible actions are taken into consideration.

Again, confusion arises because of the identification of the name of the property with the name of the equivalence. In this case the problem is more severe than with strong bisimulation and strong equivalence as the two names resemble each other less, and often give the impression that there are two equivalences, one called weak bisimulation, and the other called observation equivalence.

There are a number of algebraic laws associated with observation equivalence. For finite agents (i.e. terminating processes) these laws form a complete axiomatisation [HM85]; this result was extended to finite state agents in [Mil89a]. Note that these axiomatisations are not finite; given the operators of CCS, a complete, finite axiomatisation of observation equivalence is not possible [Mol90]. Observation equivalence is undecidable in general.

Alternative Characterisations of Observation Equivalence

There are a number of alternative methods of defining bisimulation. We give them here as they may be useful in developing proof techniques later.

Refining the Universal Relation The above definition of observation equivalence starts from the empty relation and adds pairs of processes to the relation to give a bisimulation. The following definition works the other way, i.e. starting with the universal relation at the leaves of the labelled transition system, and gradually refining the relation as we head towards the root.

This method of defining observation equivalence is the original observation equivalence, and was given in [Mil80], before the bisimulation method was developed.

Definition 3 (Observation Equivalence) *The new relation is defined in terms of a series of relations, with the first in the series being the universal relation, each of the others being defined in terms of the previous relation.*

$P \approx_0 Q$ *holds for all* P *and* Q;

$P \approx_{k+1} Q$ *iff.* $\forall \sigma \in \mathcal{L}^*$

(i) *if* $\exists P' : P \overset{\alpha}{\Longrightarrow} P'$ *then* $\exists Q'$: $Q \overset{\alpha}{\Longrightarrow} Q'$ *and* $P' \approx_k Q'$;

(ii) *if* $\exists Q' : Q \overset{\alpha}{\Longrightarrow} Q'$ *then* $\exists P'$: $P \overset{\alpha}{\Longrightarrow} P'$ *and* $P' \approx_k Q'$;

$P \approx Q$ *iff* $\forall k \geq 0.\ P \approx_k Q$

i.e. $\approx = \bigcap_k \approx_k$.

In fact, the relation obtained here is only identical to that obtained by definition 2 on the domain of *image-finite* processes. A process is image-finite if for each α the set $\{p' \mid p \xrightarrow{\alpha} p'\}$ is finite. This means that the practice of using the terms weak bisimulation equivalence and observation equivalence interchangeably is in fact incorrect.

Hennessy-Milner Logic A third characterisation of observation equivalence can be obtained by using Hennessy-Milner logic (HML), first introduced in [HM85]. This logic is presented fully in chapter 11, so we will not give the details here. The power of the logic comes from the ability to nest the modal operators to arbitrary levels. There is a direct relation between the depth of this nesting and the stratification of the relations in the previous characterisation; the proof of this can be found in [Mil89b]. A direct result of this relationship is the property that HML also precisely characterises observation equivalence, so that if two processes are not equivalent with respect to observation equivalence then an HML formula can be found which distinguishes them.

In [Mil89b] a satisfaction relation between processes and HML is defined which uses the transition relation to define what it means for a formula to be true of a process. This provides us with a mechanism for expressing properties of programs (such as deadlock) in logic and proving the CCS specification of that program satisfies those properties. Again, this is studied in more depth in chapter 11.

Using Tests The last method of characterising observation equivalence to be considered is as a *Testing equivalence.* Since testing equivalence will not be introduced until later in this section, we give only a very brief outline of the way in which this equivalence may be altered to give observation equivalence.

As mentioned in the informal introduction, observational equivalence is based on a notion of testing; however, in general, these tests rely on some very strong assumptions about the control we have over the environment. In the usual definition of *Testing equivalence*, test are constructed in a similar way to processes, and do not have the discriminatory power of observation equivalence. In [Abr87] the extra power required is gained by extending the operator set used in constructing tests to include the existential quantifiers, $\forall$ and $\exists$. These correspond to the informal notion that in observation equivalence we can control and examine the environment and that we can perform tests in all possible configurations of the environment (implying that there are only finitely many configurations). This extension to testing equivalence gives observation equivalence.

We move on now to consider observation *congruence.*

Observation Congruence

Unfortunately, observation equivalence is not preserved by summation in CCS. For example, while it is true that $\tau.b \approx b$ holds (since the τ action is unobservable), it is not true that $a + \tau.b \approx a + b$ holds because, on the left hand side, we can find ourselves in a position in which it is no longer possible to perform a (and only b is possible), i.e. the τ branch has been taken. This is due to the pre-emptive power of the τ action, which essentially converts a deterministic choice into a nondeterministic one.

This is a major failing in the equivalence. Imagine a system in which you wished to replace some subsystem by a new, perhaps more efficient subsystem, without changing the functionality of the system as a whole. Even if the old and the new subsystems were shown to be observation equivalent, we could not guarantee that the system with the new part would behave in the same way as the old system, as shown in the small example above. What is needed is a congruence relation which guarantees that two systems which are identified behave in the same way in all contexts. Observation congruence is defined to be the largest congruence relation contained in observation equivalence.

Definition 4 (Observation Congruence) *P and Q are observation congruent, $P = Q$, if $\forall \alpha \in Act$*

1. *if $\exists P' : P \xrightarrow{\alpha} P'$ then $\exists Q' : Q \overset{\alpha}{\Longrightarrow} Q'$ with $P' \approx Q'$, and*

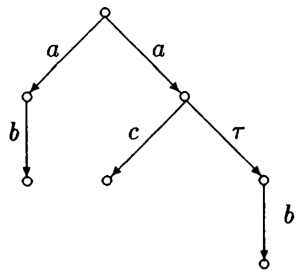
2. *if $\exists Q' : Q \xrightarrow{\alpha} Q'$ then $\exists P' : P \overset{\alpha}{\Longrightarrow} P'$ and $P' \approx Q'$.*

Note that the only difference between this definition and the definition of observation equivalence is that $\overset{\alpha}{\Longrightarrow}$ appears instead of $\overset{\hat{\alpha}}{\Longrightarrow}$ for the first transition from the root. This means that rather than throwing away all information about τ actions when comparing systems, some τ actions, those which occur at the root, are retained. In particular, if Q starts with a τ action, then that action must be matched by *one* or more τ actions in P. Observation equivalence, on the other hand, allows that such an action is matched by *zero* or more τ actions.

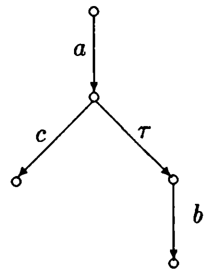
To return to the small example from the beginning of this section, we show that the problem no longer arises because although $\tau.b \approx b$, it is not true that $\tau.b = b$. The congruence does not hold because the first clause of the definition is not true, i.e. the left hand side can perform a τ action to become b, but the only way the right hand side can match this is by performing no action at all, and this is not allowed by observation congruence.

A complete axiomatisation for observation congruence over processes without recursion is given in [HM85].

Two problems arise when using observation equivalence or congruence in proofs. On one hand, it makes distinctions which are not truly observable: as mentioned above, to describe observation equivalence in terms of tests requires that we make tests very strong. This makes observation equivalence too strong for many applications. On the other hand, observation equivalence does not truly preserve the branching structure of a process, i.e. although it does preserve some information about the branch points of processes, it does not preserve the branching potential of *all* states. This can be more easily seen in an example:



$a.b + a.(c + \tau.b)$



$a.(c + \tau.b)$

In the process on the left, the decision to do b might have been made when the a is performed, i.e. the left branch is taken. In the right hand process, after a, we can still do b or c. These processes are equivalent under observation equivalence. Below, we consider equivalences which solve these problems by preserving different amounts of information about the branching structure of the process.

Branching Bisimulation Equivalence

In order to supply a branching equivalence which incorporates the notion of the unobservable action without making unnecessary identifications, ***branching bisimulation*** was developed [vG90]. Although originally defined in the setting of ACP, the definition can be easily translated into CCS (and LOTOS). Essentially the definition is the same as observation equivalence, differing in that, as well as comparing the states at the start and finish of τ sequences, it also compares states along τ sequences.

Definition 5 (Branching Bisimulation Equivalence) *A symmetric relation, $R \subseteq \mathcal{P} \times \mathcal{P}$, is a branching bisimulation if $(P, Q) \in R$ implies, $\forall \alpha \in Act$*

1. *if $\exists P' : P \xrightarrow{\alpha} P'$ then either $\alpha = \tau$ and $(P', Q) \in R$, or*
   $\exists$ a path : $Q \Longrightarrow Q_1 \xrightarrow{\alpha} Q_2 \Longrightarrow Q'$ with $(P, Q_1) \in R, (P', Q_2) \in R, (P', Q') \in R$, and
2. *if $\exists Q' : Q \xrightarrow{\alpha} Q'$ then either $\alpha = \tau$ and $(P, Q') \in R$, or*
   $\exists$ a path : $P \Longrightarrow P_1 \xrightarrow{\alpha} P_2 \Longrightarrow P'$ with $(Q, P_1) \in R, (Q', P_2) \in R, (Q', P') \in R$.

Two processes, P and Q, are branching bisimulation equivalent, written $P \leftrightarrow_b Q$, if $(P, Q) \in R$ for some branching bisimulation R.

Branching bisimulation equivalence has some pleasing properties, including that it is a congruence relation, that it is decidable for a certain class of processes [Hüt91] (see section 4.2.2 for more information) that it has a complete axiomatisation, and a complete term rewriting system corresponding to those axioms [AB90].

Testing Equivalence

As mentioned in section 3.4.2, the strong and weak bisimulation equivalences are generally inadequate in real world applications because they rely on fairly strong assumptions about the degree of control over the environment and tests on processes (e.g. the ability to make multiple copies of the environment and the process under test is assumed). Testing equivalence [DH84, Hen88] is based on the notion of *experiments* on processes. Two processes are testing equivalent if they pass the same tests. To define this equivalence we need *a set of observers*, *a way of observing*, and *a criteria for judging the results of the observations.*

Given a set of states, **States**, a *computation* can be defined as a non-empty (possibly infinite) sequence of states. **Comp** denotes the set of computations and is ranged over by c.

Let $\mathcal{O}$, $\mathcal{P}$ (ranged over by o, p respectively) be a set of **observers** and a set of **processes**. For every o and p there is a non-empty set of computations $\mathbf{Comp}(o, p)$, which denotes the effect of the observer o performing tests on the process p. The outcome of a test is then $c \in \mathbf{Comp}(o, p)$. We define a subset of **States** to be the **Success** states, denoted by $\{\top\}$. Unsuccessful states are denoted by $\{\bot\}$. A computation is successful if it contains a successful state, unsuccessful otherwise. We denote the result set of a test by $\mathcal{R}(o, p)$. The tests may be repeated a number of times, which may yield the additional result, $\{\top, \ \bot\}$, i.e. sometimes the process will pass the test, sometimes it will fail.

Testing equivalence is defined in terms of three relations which reflect different views of how to order these results as domains.

1. *The Hoare Domain*, see figure 3.3, reflects the view that the possibility of failure is not a disaster, and therefore equates $\{\top\}$ with $\{\top, \ \bot\}$, i.e. some experiments may fail, but at least one experiment was successful. Given a process p, p **may satisfy** o if $\top \in \mathcal{R}(o, p)$.

   The relation on processes derived from this domain is written $\sqsubseteq_{may}$, where $p \sqsubseteq_{may} q$ if $\forall o \in \mathcal{O}$, p **may satisfy** o implies q **may satisfy** o.

2. *The Smyth Domain*, see figure 3.3, is the opposite of the Hoare domain, in that the possibility of failure is viewed as catastrophic, and $\{\top, \ \bot\}$ is equated with $\{\bot\}$. Given a process p, p **must satisfy** o if $\{\top\} = \mathcal{R}(o, p)$.

$$\begin{tikzcd}[column sep=small] \{\top\} = \{\top, \bot\} \arrow[d, dash] \\ \{\bot\} \end{tikzcd}$$

The Hoare Domain

$$\begin{tikzcd}[column sep=small] \{\top\} \arrow[d, dash] \\ \{\bot\} = \{\top, \bot\} \end{tikzcd}$$

The Smyth Domain

$$\begin{tikzcd}[column sep=small] \{\top\} \arrow[d, dash] \\ \{\top, \bot\} \arrow[d, dash] \\ \{\bot\} \end{tikzcd}$$

The Egli-Milner Domain

Figure 3.3: Testing Domains

The relation on processes derived from this domain is written $\sqsubseteq_{must}$, where $p \sqsubseteq_{must} q$ if $\forall o \in \mathcal{O}$, p **must satisfy** o implies q **must satisfy** o.

3. *The Egli-Milner Domain*, see figure 3.3, reflects a more balanced view of the possibility of failure, equating it with neither $\{\top\}$ nor $\{\bot\}$.

   The relation on processes obtained from this domain is written $\sqsubseteq_{test}$, and $p \sqsubseteq_{test} q$ iff $p \sqsubseteq_{may} q$ and $p \sqsubseteq_{must} q$.

Each of these relations is a preorder, i.e. a reflexive and transitive relation.

Definition 6 (Testing Equivalence) *Testing equivalence is defined in a natural way by the following:*

p and q are testing equivalent if $p \sqsubseteq_{test} q$ and $q \sqsubseteq_{test} p$.

Informally, p and q are testing equivalent if there are no tests which one passes but the other does not. As with strong and observational equivalence a complete axiomatisation exists for testing equivalence over finite processes [DH84].

The power of testing equivalence varies, depending on three factors: the power of the observers, the criteria for determining success or failure of a test, and the method of tabulating the results of a test. Varying these factors gives a range of different testing equivalences. The observers most commonly used are constructed in the same way, and using the same operators, as processes, with the addition of an action ω which denotes success. A test is successful if the ω action is observed. The resulting equivalence is the one most commonly used, and is the equivalence we mean when we refer to "testing equivalence".

Tests can be made stronger, as mentioned previously, by allowing existential quantifiers, as in [Abr87], as part of the language. This alters the way in which the information from the tests is collated. Another possible extension is to change the treatment of divergence; specifically, if a process enables the observer to perform an ω action but subsequently diverges, should a success or failure be observed?

Characterisations of Testing Equivalence

We now consider other means of characterising testing equivalence. The full definitions may be found in [De 87].

Labelled Transition Systems The previous discussion of testing equivalence was in a general setting. To be applicable for CCS (or LOTOS), the notions of observers and successful states must be defined in terms of labelled transition systems. An observer (or experimenter) is just a process with the additional action ω, denoting success. A state is a $\langle$process, observer$\rangle$ pair. The state is successful if the observer can perform an ω action. Given these definitions, computations are sequences of states, and the preorders **may** and **must** are defined in terms of success states within computations.

Processes are sets of LTS's as defined in section 3.4.3.

Observers (or *experimenters*) will be **Processes** with the additional action ω, which reports success.

States will be pairs $\langle p, e\rangle$ where p is a state of a process and e is a state of an experimenter. A successful state is one whose right component can perform an ω action.

Computations Given two transition systems, T and E, with initial states t and e, a computation from $\langle t, e\rangle$ is a finite or infinite sequence of pairs of states $\langle t_n, e_n\rangle$ where:

1. $\langle t_0, e_0\rangle$ is $\langle t, e\rangle$.
2. (a) $\langle t_n, e_n\rangle \xrightarrow{\tau} \langle t_{n+1}, e_{n+1}\rangle$ if $t_n \xrightarrow{\tau} t_{n+1}$ and $e_n = e_{n+1}$, or $e_n \xrightarrow{\tau} e_{n+1}$ and $t_n = t_{n+1}$,

   (b) $\langle t_n, e_n\rangle \xrightarrow{a} \langle t_{n+1}, e_{n+1}\rangle$ if $t_n \xrightarrow{a} t_{n+1}$ and $e_n \xrightarrow{\bar{a}} e_{n+1}$
3. if the sequence is finite with $\langle t_k, e_k\rangle$ as final element then no more transitions in *Act* are possible from $\langle t_k, e_k\rangle$.

The relations **may satisfy** and **must satisfy** can now be redefined in terms of the above definitions.

1. T **may satisfy** E if $\exists \sigma \in Act^*.\langle t_0, e_0\rangle \xrightarrow{\sigma} \langle t_n, e_n\rangle$ and $\exists e_{n+1}.e_n \xrightarrow{\omega} e_{n+1}$.

2. T **must satisfy** E if for every computation $\langle t_0, e_0\rangle \xrightarrow{\alpha_1} \langle t_1, e_1\rangle \xrightarrow{\alpha_2} \langle t_2, e_2\rangle \longrightarrow \ldots$ there exists $n \geq 0$ such that $e_n \xrightarrow{\omega} e_{n+1}$.

The preorders and the equivalence are defined as before, but using the above definitions of **may satisfy** and **must satisfy**.

Action Sequences The problem with the above characterisations of testing is that, although they are very easy to understand, they are not so useful in proofs of equivalence. Proving non-equivalence is straightforward: only one observer which differentiates the processes must be found, whereas a rigorous analysis of all possible tests is necessary to prove equivalence.

An alternative method of characterising this equivalence uses sequences of actions to define the three orderings. This is the method used in the LOTOS standard to define the testing preorders. As the definitions are given in section 3.5.3 we do not repeat them here.

Failures In [De 87] the important result that testing equivalence has the same discriminatory power as the equivalence induced by failures semantics in CSP is given. This means that testing equivalence can also be defined in terms of traces and failures. The definition turns out to be just the contrapositive of the definition in terms of action sequences given in section 3.5.3, i.e. rather than concentrating on the actions a process *can* perform we concentrate on the actions it *cannot* perform.

3.4.4 Proof Techniques for CCS

The kinds of proofs which can be constructed for CCS processes vary depending on the particular semantics/equivalence relation adopted.

- The operational semantics may be used to "simulate" a behaviour. This allows us to trace through the execution of a process, highlighting situations in which deadlock may occur and so on.
- Two terms to be proved equivalent may be manipulated by applying the algebraic laws associated with the particular equivalence/congruence/preorder used. This is always sound, and in some cases, e.g. for branching bisimulation, may also be complete.
- The alternative methods of characterising the equivalences can be used instead of referring to the labelled transition system or the axioms. For example, using strong equivalence or observation equivalence we may proceed by trying to find a (strong or weak) bisimulation R which includes the pair of terms to be proved equivalent. For observation equivalence we may also try to prove that two processes are not equivalent by finding an HML formula which holds for one but not the other. To prove that two terms are not equivalent under testing equivalence a test must be found which distinguishes them. Alternatively, a case analysis over tests must be carried out to show that there is no test which distinguishes the terms.
- A combination of the above methods can be used as appropriate for the given problem, e.g. first use the algebraic laws to simplify the processes, making the other proof techniques

simpler to apply.

Another important technique used in proofs in CCS is *unique fixed point induction*. This allows us deduce that two recursively defined processes are equivalent if they satisfy the same general equation. This is due to a result about the uniqueness of solutions to equations. Again this definition is taken from [Mil89b].

Definition 7 (Unique Solution of Equations) *Consider solutions of the expression* $\tilde{X} = \tilde{E}$*, where* $\tilde{X}$ *and* $\tilde{E}$ *denote vectors of variables and process expressions respectively.*

Let the expressions $\tilde{E}$ *be guarded and sequential expressions, i.e. all expressions have visible initial events and use only the operators* . *and* $+$*, with free variables in* $\tilde{X}$*. Let* $\tilde{P} = \tilde{E}\{\tilde{P}/\tilde{X}\}$ *and* $\tilde{Q} = \tilde{E}\{\tilde{Q}/\tilde{X}\}$*. We may deduce* $\tilde{P} = \tilde{Q}$*.*

In the above, $=$ stands for observation congruence, but similar results have been obtained for other equivalences/other formalisms.

Having introduced CCS and CSP, we can now consider the formal description technique LOTOS.

3.5 LOTOS

LOTOS [ISO88] is based on the concept of specifying a system in terms of observable behaviour, i.e. events, and was designed by ISO (International Standards Organisation) with the specification of communications protocols in mind. This has had a great bearing on the design decisions taken when developing the language; in particular, LOTOS is very expressive, with a large operator set, including mechanisms for structuring large specifications.

As in CCS and CSP, LOTOS has no explicit representation of time; however, constraints may be placed on the *order* of events. The communication and change of information within a system is expressed by the structure of those events.

LOTOS consists of two parts: the process algebra Basic LOTOS and the abstract data type specification language ACT ONE. In this section, and for the following six chapters, only the process part of the language, Basic LOTOS, is considered; the data type part is discussed in chapter 10. The complete syntax and semantics of full LOTOS is given in appendix B.

3.5.1 Operators of Basic LOTOS

The operators of Basic LOTOS are given in figure 3.4, in which a is an event, P, Q and ex are processes, G is a set of gate names, and S is a relabelling function.

Some LOTOS operators have exactly the same behaviour as their CCS counterparts:

description	notation
internal action	**i**
inaction (deadlock)	**stop**
successful termination	**exit**
action prefixing	$a;\ P$
choice	$P \,[]\, Q$
parallelism (general)	$P \mid[G]\mid Q$
parallelism (interleaving)	$P \mid\mid\mid Q$
parallelism (full synchronisation)	$P \mid\mid Q$
enable	$P \gg Q$
disable	$P [> Q$
hide	**hide** G **in** P
renaming	$P[S]$
recursion	**proc** ex := a; ex **endproc**

Figure 3.4: Operators of Basic LOTOS

- τ becomes **i**.
- 0 becomes **stop**.
- $a.P$ becomes $a; P$.
- $P + Q$ becomes $P \,[]\, Q$.

The remaining operators are either ones which owe more to CSP than CCS, or ones specially introduced to make the specification of communications protocols easier.

- In addition to the CCS unsuccessful termination, i.e. deadlock (**stop**), LOTOS also has successful termination, denoted **exit**. A new event, δ, denoting successful termination is added, along with the transition rule **exit** $\xrightarrow{\delta}$ **stop**.
- In CSP the special communication events are called channels; in LOTOS they are called *gates*.
- The parallelism operator is altered to allow explicit specification of the set of gate names, G, on which the processes must synchronise (this follows the model of CSP parallelism).
- LOTOS has broadcast communication.
- Interleaving is parallelism with an empty set of gate names, i.e. no synchronisation. Full synchronisation is parallelism with the set of gate names equal to the language of the processes being combined, i.e. everything synchronises.
- Enable, denoted by $\gg$, means the sequential composition of processes[4].

[4]In the concurrency literature *enable* has a different meaning. If an action is enabled, then it is ready to be performed. This difference is irritating, but enable is the name used in the LOTOS standard for this operator.

- Disable, denoted [> , allows Q to interrupt the execution of P and take control.
- LOTOS **hide** is like CSP hide, rather than CCS restriction, i.e. hidden events are turned into occurrences of the internal action, and may proceed without constraint.
- The post-fix renaming operator applies a relabelling function to the gate names. The only constraint on the behaviour of the function is that it must map the internal action to itself.

An important area of LOTOS-related research, which is not a feature of CCS or CSP research, is the investigation of the ability to write specifications in different styles. Some of the most common styles are described below.

3.5.2 LOTOS Specification Styles

In [VSvSB91] two basic characterisations of the descriptive style of a specification are given: extensional, an abstract style which describes *what* the system does, and intensional, a more concrete style which describes *how* the system operates, typically also giving internal structuring of the system. For Basic LOTOS, two more styles within each of these classes can be identified. For extensional descriptions these are *monolithic* and *constraint-oriented*, while for intensional specification styles we have *state-oriented* and *resource-oriented.* In the monolithic style, observable interactions are presented as a collection of alternative sequences in branching time. A characteristic of this style is the absence of parallelism, only choice and ordering are used. In constraint-oriented specification, different aspects of the system are separated and described as individual processes. These processes are recombined using parallelism to give the whole specification. This style of specification is popular because in general it is easier to understand the behaviour of small components than large ones. However, this can lead to a false understanding of the system, as the interactions of the components can be very complex and easily misunderstood. The user must rely on tools to check that the combined behaviour is as intended. Examples of this style may be found in [Naj87, Tur92], and also here, in section 7.4.3.

In the state-oriented specification style, interactions manipulate a global state variable. No structuring other than choice is used which is similar to the monolithic style. Finally, we have the resource-oriented specification style, which is similar to the constraint oriented style, except that here each component can be identified with an underlying implementation feature rather than with an abstract feature of the system. Tools exist which can transform a specification written in one style into another style [vE89, LITE].

The constraint-oriented and resource-oriented styles also satisfy more general specification style concerns such as orthogonality (functional independence of parts), generality (parametric specifications), and open-endedness (flexible, easily extended specifications). This makes them more suitable for early, abstract specifications, while the other styles are nearer to the level of the implementation of the system, particularly the state oriented style.

In most specifications, a mixture of styles will be used, starting with constraint-oriented, which is more easily understood by a user, and moving to state-oriented or resource-oriented, which may be closer to implementation. An interesting question, from our point of view, is how the verification process might be affected by the style in which a process is written. Obviously, since there are transformations from one style to another, the specification style should not alter the validity of a property; however, it may be *easier* in a particular style to show that property holds of the specification.

3.5.3 Semantics of Basic LOTOS

Basic LOTOS is based on the same model as CCS: the semantics of processes are given by labelled transition systems. The inference rules defining the semantics of LOTOS, taken from [ISO88], are given in appendix B. Everything that was said about equivalences in CCS can also be applied to Basic LOTOS, and we do not repeat the definitions here. Some laws, also taken from [ISO88], for weak bisimulation congruence and equivalence, testing equivalence, testing congruence, **red** and **cred** are given in appendix B. In the LOTOS laws the operator $\mathcal{L}$ is often used. This operator takes a process and returns the *language* of that process, i.e. the set of events in which it may participate.

In the rest of the thesis the following notations are used. We have tried to maintain compatibility with both the LOTOS standard and the definitions and notation used in the previous section.

- The domain of gates is denoted by $\mathcal{G}$ (this is similar to the domain of labels in CCS, but there is no notion of a co-label in LOTOS as communication is multi-way). The new termination action is treated in the same way as the other gate labels, $\delta \in \mathcal{G}$, but, as in CCS, internal actions are treated differently, $\mathbf{i} \notin \mathcal{G}$. We use $g, g_1, \ldots$ to range over $\mathcal{G}$ and $t, t_1, \ldots$ to range over $\mathcal{G}^*$. Subsets of $\mathcal{G}$ may be denoted by A, G or L (depending on the context).

- The domain of actions is denoted Act as before. $Act = \mathcal{G} \cup \{\mathbf{i}\}$. We use $\alpha, \beta, \ldots$ to range over Act.

- The definitions of $\longrightarrow$ and $\Longrightarrow$ are as before, modulo renaming of variables and substitution of $\mathbf{i}$ for τ.

- We denote LOTOS weak bisimulation congruence by $\equiv_{wbc}$ and weak bisimulation equivalence by $\approx_{wbe}$. The notation for the other relations is defined in a similar manner.

In addition to defining weak bisimulation equivalence and congruence for LOTOS, the LOTOS standard also defines the preorder **red** and its congruent counterpart **cred**. These are essentially the *testing* preorders of section 3.4.3, as described in [DH84, De 87, Hen88], although the LOTOS convention is to write the arguments in the opposite order, i.e. *impl* **cred** *spec*, rather than *spec* $\sqsubseteq_{\approx}$ *impl.*

We give the LOTOS standard's presentation of testing in terms of action sequences below. To define the **red** and **cred** relations some more auxiliary definitions are required. Remember the slightly different naming conventions of variables in LOTOS as opposed to those used in CCS. For example, traces, denoted by t, may also contain δ.

- P **after** $t = \{P' \mid P \overset{t}{\Longrightarrow} P'\}$.
- Let $L \subseteq \mathcal{G}$, then

  $$(P \textbf{ must } L) \textit{ iff. } P \overset{\epsilon}{\Longrightarrow} P' \textit{ implies } \exists\, P''.\ P' \overset{l}{\Longrightarrow} P'' \textit{ for some } l \in L$$

  P **must** L is equivalent to saying that if P performs any visible actions then it must perform an action from L, assuming $L \neq \emptyset$. The expression $P \overset{\epsilon}{\Longrightarrow} P'$ guarantees freedom from divergence, i.e. any sequence of internal actions is finite. There are two special cases of note: **stop must** L and P **must** $\emptyset$, both of which give false, regardless of L and P.

  Note that this relation really has more of the flavour of **may** than **must**, since it expresses the *possibility* that an action may be performed rather than the *certainty* that that action will be performed. The **must** flavour comes from the following definition extending the relation to *sets* of states.

- The **must** predicate can be defined over sets of states in the obvious way. Let $\mathcal{Q} \subseteq \mathcal{P}$, then $\mathcal{Q}$ **must** L *iff.* $\forall\, Q \in \mathcal{Q}.\ Q$ **must** L.

  An interesting special case here is $\emptyset$ **must** L which is always true, even when $L = \emptyset$. In the definition of **red** below this case occurs when the trace t is not a valid trace of the process, therefore P **after** t is empty.

- Let $C[\,]$ be a LOTOS context, then $C[P]$ denotes the process P in the context $C[\,]$.

The **red** and **cred** relations may now be defined.

Definition 8 (The red relation) P **red** Q *iff.*
$\forall t \in \mathcal{G}^* \ \forall L \subseteq \mathcal{G}.\ (Q \textbf{ after } t) \textbf{ must } L \Rightarrow (P \textbf{ after } t) \textbf{ must } L$

Definition 9 (The cred relation) *For LOTOS behaviour expressions P and Q,*
P **cred** Q *iff. for all LOTOS contexts* $C[\,]$, $C[P]$ **red** $C[Q]$.

Both of these refinement relations are preorders (i.e. reflexive and transitive relations). Divergence is ignored, unless it comes before a successful state.

An alternative definition of **red** is given by [Bri88a] and [Lan92].

Definition 10 (The red relation 2)
P **red** Q *if*

$\forall t \in \mathcal{G}^*, \forall L \subseteq \mathcal{G}$

if $\exists P'.\ P \overset{t}{\Longrightarrow} P' \ \wedge\ \forall a \in L\ \neg(\exists P''.\ P' \overset{a}{\Longrightarrow} P'')$ *then*

$\exists Q'.\ Q \overset{t}{\Longrightarrow} Q' \ \wedge\ \forall a \in A.\ \neg(\exists Q''.\ Q' \overset{a}{\Longrightarrow} Q'')$

Informally, this definition describes **red** in terms of events which it *cannot* perform, i.e. refusals, rather than in terms of events which it *may* perform. These two forms of definition are in fact equivalent, since this is just the contrapositive of the previous definition of **red**. This also underlines that tests and failures are just different ways of expressing the same equivalence relation.

Similar/related preorders to **red** and **cred** have been defined elsewhere in the literature. For example, **red**, **ext** and **imp** in [BSS87] and **red** and **conf** in [Bri88a]. Note that the same name need not denote the same relation; however, the differences are minor.

The definition of **red** in [BSS87] is like definition 10 above except that the trace set of the implementation must be contained within the trace set of the specification. The **ext** preorder, on the other hand, insists that the trace set of the specification process be contained within that of the implementation process. Moreover, although the rest of the definition is the same as the alternative **red** given above, the traces are taken from the language of the specification, rather than from Act^*. This difference makes sense as the intention of [BSS87] is to develop a method of deriving tests from the specification. Using tests from Act amounts to testing the system for *robustness*, i.e. correctness in the presence of incorrect/unexpected inputs, whereas restricting the tests to traces of the specification means that only defined inputs are tests; all others are ignored. The equivalences induced by **red** and **ext** are the same. The third preorder of [BSS87], **imp**, is defined by **imp** = **red ext**, i.e. a composition of the two previous preorders. It behaves like the **red** of definition 10 except that it insists that the specification and the implementation have the same traces. These three relations evolve into **red** and **conf** in [Bri88a], where **red** is as in definition 10 and **conf** is similar to **ext** except that no constraints are placed on the trace sets of the processes.

The definitions of [BSS87] were early attempts to express the notion of implementation which were later refined in [Bri88a], therefore we choose to use the later definitions in the following. We do not use **conf** as it is not a preorder (it is not transitive) and therefore does not yield an equivalence, unlike **red**. Since most of our work will be based on equivalence relations **red** is therefore a more desirable relation to use.

Complete axiomatisations for LOTOS equivalences and preorders do not exist, as far as the author is aware, or rather, they have not been documented. Certainly finite Basic LOTOS can be transformed into finite CCS, see [BIN92], and complete axiomatisations exist for finite CCS, and therefore for finite Basic LOTOS.

3.5.4 Proof Techniques for LOTOS

The proof techniques for LOTOS are borrowed directly from the proof techniques for CCS. With a little manipulation, CSP-like proofs can also be obtained, this is described in [GL91] and [GLO91]. A problem of the approach is that the fixpoint induction principle cannot be applied to LOTOS with failures semantics as the solutions to equations are not unique; this is because the ordering over sets of failures is non-monotonic with respect to the corresponding processes in the presence of **i**.

In addition to the usual CSP and CCS proof techniques, *property testing* is a commonly used proof technique for LOTOS. The technique consists of describing the property in the style of a test process and executing that test in parallel with the system to be tested. The test process includes a special action which indicates that the test has been passed and the two processes synchronise on all other events. If the system can perform the same actions as the test process then the test is passed; if not, the test fails. This technique is based on testing equivalence, but typically only one test (one property) is considered, rather than all tests. This proof technique used in [Tho93], which is also described in chapter 9.

The power of the tests can be increased by extending the language of tests. One possible extension is to add an action which can detect deadlock; this form of testing is described in [Lan90].

We have now presented the three process algebras CSP, CCS and LOTOS. Before moving on to consider verification of properties of LOTOS specifications in more detail, we summarise the differences and similarities of the three formalisms. We also compare the relative strengths of the different equivalence and congruence relations defined for all three process algebras.

3.6 Comparison of Process Algebras

3.6.1 Comparing the Formalisms as Specification Languages

The process algebras CSP, CCS and LOTOS are all very similar in their basic concepts. Each formalism is described in terms of a set of operators over the same basic domain: processes and actions. They differ in their philosophy and area of application however. When CCS was defined the intention was to have a minimal set of operators which would allow the semantics of the language to be more easily explored. Although good for small examples, CCS is more difficult to use in larger scale examples because it lacks the operators necessary to structure a larger specification. However, some of these operators can be constructed from the basic operators if necessary. On the other hand, both CSP and LOTOS were designed to be used as specification languages. LOTOS in particular was designed for large communications systems, and therefore has lots of operators which make it easy to build a large specification out of smaller parts.

Having a large operator set can cause problems, simply because the proliferation of operators may cause confusion, thus making the specification process more difficult in some circumstances. Also, as mentioned in the introduction, the more expressive a specification language is, the more complicated verification becomes (either through the language being based on a more complex mathematical model, or simply through the confusion of having lots of operators).

The three formalisms, although based on a common semantic model (CSP can also be defined in terms of labelled transition systems), have slight differences in the way in which that model is interpreted. For example, communication in CSP and LOTOS is based on multi-way synchronisation,whereas CCS restricts communication to two parties. The differences in the approach to communication are also reflected in the approach to hiding and restriction. While CSP and LOTOS turn hidden actions into internal actions (which may then proceed instantaneously), CCS uses restriction to prevent a process from communicating with its environment, usually forcing it to communicate with some other process.

Another difference is the use of distinguished actions. CSP has one distinguished action, $\surd$, which signifies successful termination. Although the hiding operator in CSP, as in LOTOS, produces internal, invisible, actions, CSP has no special notation for this. CSP invisible actions really are invisible! CCS, on the other hand, relies quite heavily on the internal action for the result of a communication, and for modelling nondeterminism in the system (whereas CSP has a special operator for nondeterministic choice). LOTOS, as might be expected, reflects both formalisms in that it has a distinguished internal action; however, its treatment of **i** is a combination of the CCS use of τ, and the CSP creation of internal events by hiding.

With regards to syntax and semantics of operators, it can be seen that CSP and LOTOS are quite similar and useful as practical specification languages, while CCS is more suited to small theoretical investigations. When considering proof systems, LOTOS is aligned with CCS rather than CSP. The operators of all three formalisms are associated with a set of algebraic laws which allow transformation of process expressions. In addition to this, CSP also has an associated abstract specification language, which can be used to express the requirements of a system, and a satisfaction relation, which tells us when a CSP process meets those requirements. CCS is also closely related with a more abstract specification language, the logic HML, which can be used in a similar way. LOTOS however, has no such language. What LOTOS does have, which the other two lack, is extensive tool support; this is a result of being designed for use in industry rather than academia. Of course there are tools associated with CSP and CCS, but there are not so many, and they are mostly proof tools, whereas LOTOS tools offer support in various other forms of analysis. Details of tool support for CCS and LOTOS are given in appendix A. Where CSP and CCS gain is that there is an extensive associated literature, while LOTOS, being relatively new, has only a small literature.

Finally, we may also consider *implementations* of processes: CSP is closely related to the language occam™, CCS is incorporated in the language LCS [BS94], and tools exist which translate LOTOS into C [LITE].

Further comparison of CCS and CSP may be found in [vG86]; the evaluation of LOTOS with respect to other concurrency formalisms, including CSP and CCS, may be found in [Fid93].

Although proof systems are mentioned above, they are not discussed in detail. All methods of proof, i.e. traces, failures, tests and bisimulations, are relatively easy to understand, and to a great extent preference for one or the other depends on the subjective choice of the specifier. However, the behavioural equivalences associated with each proof technique can be compared more objectively.

3.6.2 Comparing the Different Equivalences

Several different equivalences have been defined in sections 3.3.2, 3.4.3 and 3.5.3. In [vG90] a general framework, based on transition systems, is presented which allows these equivalences to be compared in terms of distinguishing power. The use of a common model also enables us to see that an equivalence defined for use with CSP may just as easily applied to LOTOS or CCS, or vice versa.

A similar study of some of the commonly used equivalences is detailed in [De 87], but without the cohesive underlying framework. However, this work does consider equivalences which involve the τ action, while the study of [vG90] considers only *finitely branching*[5], *concrete, sequential processes.* This means that the processes under consideration can have no internal actions, they must be *compliant* (they may not block actions — only the environment may block actions), and actions may not occur simultaneously. In other words: no infinite summation (choice), no silent actions, no nondeterminism (although nondeterminism occurs when a choice is given which has the same first action in each side) and no true parallelism are permitted.

Note: we do not consider the **red** and **cred** preorders in this framework since it does not make sense to compare preorder relations with equivalence and congruence relations.

[5]Finitely branching processes may be recursive, but at each stage, only finitely many choices are allowed.

Below we give the informal description of the comparative framework of [vG90], the ordering of the relations presented here in that framework, and some examples illustrating the differences between those relations.

Observing Processes

The semantics of a process is determined by its observable behaviour. An observer may observe all actions of a process, and may terminate the observation at any time. An observation is a sequence of actions over A, in other words, a *trace*. Three parameters of observable behaviour are used. The first concerns time:

1. **Linear Time** The observer may observe the process many times and under all circumstances, giving a set of traces.
2. **Branching Time** At any time in the execution the observer may split the process into several copies of itself, each taking a different execution path. This results in a tree of observations.

The second parameter is concerned with the sort of events which may be observed:

1. **Only Actions**
2. **Also Idleness** Traces are defined over $A \cup \{\delta\}$, where δ denotes idleness, i.e. periods during which the process performs no actions.
3. **Menus** At each idle period, a menu of possible actions is recorded.

Finally, the effect of the environment on the process is considered.

1. **Static Environment** The environment will allow any action to proceed.
2. **Intermediate Environment** The environment may block actions. Once blocked, an action cannot be re-enabled.
3. **Dynamic Environment** The environment is free to block and unblock actions at any time in the execution of the process.

This results in a total of 18 semantics or equivalence relations, but many of these coincide in the particular range of processes chosen. We also limit this discussion to those relations presented in this chapter, with the inclusion of *complete trace equivalence* which is just the language equivalence of finite state automata.

The semantics can be ordered by the partial order "makes at least as many identifications as", written $\mathcal{S} \preceq \mathcal{T}$, if $\mathcal{S}$ makes as many (possibly more) identifications as $\mathcal{T}$. This means that

$\mathcal{S}$ is a coarser equivalence than $\mathcal{T}$, i.e. let iden($\mathcal{S}$) be the set of identifications made by $\mathcal{S}$, then iden($\mathcal{S}$) $\supseteq$ iden($\mathcal{T}$). See figure 3.5 for a diagram showing the positions of the seven defined semantics in relation to each other. In this diagram $\mathcal{S} \leftarrow \mathcal{T}$ represents $\mathcal{S} \preceq \mathcal{T}$. The diagram is part of a complete lattice on this ordering over all variants of observing processes, called the linear time — branching time spectrum. See [vG90, De 87] for details of the rest of this lattice, and also for proofs that the relations are indeed ordered as shown in figure 3.5.

$$\begin{tikzcd}
\textit{strong bisimulation equivalence} \arrow[d] \\
\textit{branching bisimulation equivalence} \arrow[d] \\
\textit{observation congruence} \arrow[d] \\
\textit{observation equivalence} \arrow[d] \\
\textit{failures congruence} = \textit{testing congruence} \arrow[d] \\
\textit{completed trace equivalence} \arrow[d] \\
\textit{trace equivalence}
\end{tikzcd}$$

Figure 3.5: The Distinguishing Power of Common Equivalences

Deterministic Processes

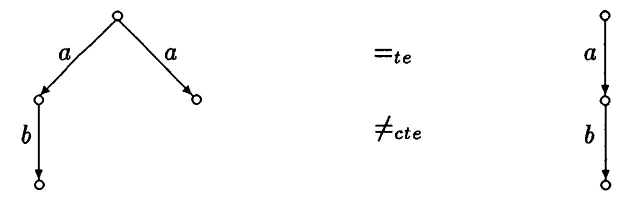
An interesting result discussed in [vG90] is that if the processes are deterministic, the whole structure of figure 3.5 collapses, i.e. all equivalences coincide.

Definition 11 *A process is deterministic if* $P \xrightarrow{\sigma} Q \;\&\; P \xrightarrow{\sigma} R \Longrightarrow Q = R$.

If the processes are deterministic, the equivalences are also decidable (since strong bisimulation equivalence is decidable for the class of normed BPA processes [GH91, Hüt91]).

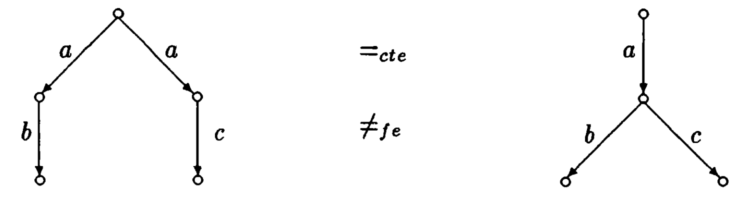
Examples

Below we give some examples which demonstrate the differences between equivalences in the above ordering. The examples are expressed in CCS syntax and are taken from [vG90].



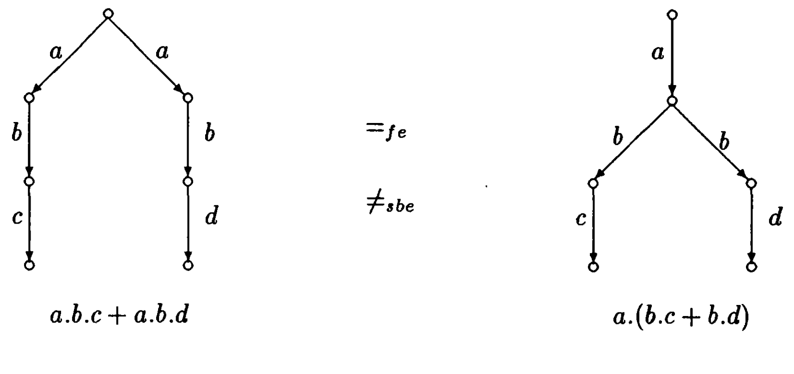
$a.b + a$ $a.b$

Completed trace equivalence is more discriminating than trace equivalence as *traces(left-hand-process)* = *traces(right-hand-process)* = $\{\epsilon, a, ab\}$, but *complete-traces(left-hand-process)* = $\{a, ab\}$, while *complete-traces(right-hand-process)* = $\{ab\}$.



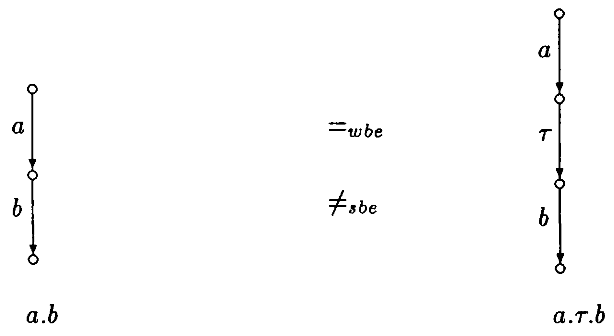
$a.b + a.c$ $a.(b + c)$

Failures equivalence/testing equivalence is more discriminating than complete trace equivalence as *complete-traces(left-hand-process)* = *complete-traces(right-hand-process)*= $\{ab, ac\}$, while $\langle a, \{b\}\rangle \in$ *failures(left-hand-process)* but not *failures(right-hand-process)*.



$a.b.c + a.b.d$ $a.(b.c + b.d)$

Strong bisimulation equivalence is more discriminating than failures equivalence, as in the left hand process we can perform a and then find ourselves unable to perform bc, while the right hand process will always be able to perform bc after a.



Any processes which are equivalent under weak bisimulation congruence by removing an internal action will never be equivalent under strong bisimulation equivalence. Similarly for branching bisimulation equivalence.

For an example of processes which are distinguished under branching bisimulation equivalence, but identified under weak bisimulation congruence, see section 3.4.3.

Discriminatory power is one of the most important ways in which we can compare these relations. In section 4.2.2 we discuss some other possible means of comparison.

3.7 Summary

In this section we have laid the foundations for the discussions of verification to come. The semantics of CSP and CCS have been introduced, and the many interpretations of those semantics have been put forward. Since LOTOS inherits much of its semantics from these two languages, in most cases the same interpretations may also be applied to LOTOS semantics, possibly with some small alterations. In addition, the various proof techniques associated with each formalism have been described. These proof techniques may be useful in giving us different ways of approaching the problem of verifying the correctness of a system. Finally, we have summarised the differences and the similarities of the three formalisms, particularly concentrating on the different semantic equivalences which may be defined for each.

In the next chapter, one particular aspect of verification is singled out for further study, namely the approach of proving two specifications, both described using process algebra, in particular Basic LOTOS, are related in some way, i.e. by a preorder, equivalence or congruence relation.

Chapter 4

Verification Requirements II: Satisfaction

In this chapter we examine in detail the proof method of evaluating the correctness of an implementation with respect to a specification by showing that the implementation *satisfies* the specification, i.e. some relation, an equivalence, congruence or preorder, holds between the two descriptions. We shall call this the *satisfaction* approach to verification. We assume the specification and implementation are given, and are both expressed in Basic LOTOS. The most important question to be answered is "what sort of relation should be used to model satisfaction?".

We begin by considering the types of relation available to us in LOTOS, equivalences, congruences and preorders, and some criteria in deciding which kind of relation to use. We then consider more specifically the various equivalence relations which can be used with LOTOS and put forward some criteria for choosing between them. We also briefly consider the question of using these relations to help *derive* the implementation from the specification.

Having considered the theoretical background of the satisfaction approach to verification, we move on to consider proof techniques and tools currently in use for either CCS or LOTOS which are based on this approach. The reason for considering CCS techniques and tools is the lack of verification tools available for LOTOS; we may be able to adapt these techniques and/or tools for use with LOTOS. From this survey of tools and techniques, we identify the technique of *equational reasoning* as appropriate for our use.

4.1 Proving the Implementation satisfies the Specification

Many case studies in the literature include examples of proving the equivalence of two behaviours, one which represents a "specification" of the system, the other the "implementation"; see below for references to a selection of such case studies. The specification here is an *abstraction* of the implementation. This may mean that it is described in a more abstract, less implementation dependent way; the LOTOS specification styles mentioned in section 3.5.2 illustrate some of the different levels of abstraction possible within one language. Alternatively, abstraction may mean that the specification is *partial*, i.e. it describes only certain aspects of the behaviour of the system, whereas the implementation may capture the full behaviour of the system. In this case an equivalence relation is an inappropriate means of comparing the specification with the implementation; a preorder relation should be used instead. Having said that, most of the examples in the literature use equivalence relations, therefore we will concentrate on the former interpretation of abstraction.

Examples of the satisfaction approach to verification are particularly common in OSI communications examples, due to the structure of the OSI Reference Model [ISO74]. Each layer of the Reference Model is specified in two ways: a description of the service provider in terms of the services available to the layer above, and a description of those services as *protocols*, i.e. interfaces between the user, which is the service provider, and the service providers of the layer below.

This approach to verification is applied by many case studies in the literature. Some use tools to aid in the proof, others carry out the verification by hand (this is only feasible for relatively small examples). Those which use tools to automatically construct an equivalence relation include: train/car level crossing [Bai91], communications protocols [Par88, BA91, CN91], sliding window protocol [MV91b], ISO Reference Model layer [Naj87], communications protocol [Ern91], protocol for overtaking cars [EFJ90], and hand-over procedure in a mobile phone network [FO91].

Other case studies rely on equational reasoning: curious queues [vG90], sliding window protocol [Gro87], and hand-over from one base station to another in a mobile phone network [OP91]. Most of these are done by hand. A particularly rich source of equational reasoning examples is [Bae90] which deals with applications of the process algebra ACP (which is defined only by sets of axioms, making equational reasoning an obvious paradigm to adopt) to specific examples. Also good for many examples of equational reasoning and constructing bisimulations by hand is [Mil89b].

The papers on automatic tools referenced in section 4.4 and in appendix A contain examples of automated proofs; common examples include Milner's scheduler and the alternating bit protocol. Another example is the verification of protocols governing the logging-on interactions between a user and a computer system [Kir93], also described in chapter 7.

Given that we wish to show the implementation satisfies the specification, we have first to decide which sort of relation, i.e. equivalence, congruence, or preorder, is appropriate for a particular example. In particular, if an equivalence relation is used, which of the many relations defined in the last chapter should be used. This decision depends on which parts of the observable behaviour are important and are to be used in evaluating the equivalence. The answers to these questions are considered below.

4.2 What Sort of Relation Should Be Used?

4.2.1 Equivalence, Congruence or Preorder?

For LOTOS, we have a choice of three kinds of relation with which we may compare processes: equivalence, congruence or preorder relations. In most verification proofs, we will want to ensure that the substitutivity property is preserved, i.e. that two equivalent processes will have the same behaviour in all contexts, therefore if a congruence relation is available, we should always choose it in preference to the corresponding equivalence relation. This applies to weak bisimulation congruence and testing congruence; strong equivalence, branching bisimulation equivalence and trace equivalence are all also congruence relations. This criteria also applies to preorder relations.

We still have to decide between using an equivalence/congruence relation and a preorder relation.

If the specification and implementation are developed independently, then choosing between congruence and preorder is largely a matter of trial and error. However, if we assume that the implementation is somehow derived from the specification and that we have information about that derivation process, then we can use that information in our decision. This is not a strong assumption, as in general we expect that the specification will at least be used as a reference when defining the implementation. Note that we do not assume that the implementation is *formally* derived from the specification, as this would make the verification trivial, see section 4.3.

As in section 2.2.2 we assume a sequence of specifications, $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$. This time we also assume we know a little more about the relationship between each specification and the next.

Given two descriptions in the sequence, S_i and S_j, where $i < j$, we view S_i as the *specification* of the system and S_j as the *implementation*. Four interpretations of implementation are identified in [BSS87]; we consider three of these interpretations below. The fourth interpretation of the term implementation is as the final code of the system. Since we stated in chapter 2 that we are not interested in program code, this particular interpretation is ignored here.

We consider three sorts of steps which may be taken from the specification to reach the implementation.

Reduction Resolving choices which were left open in the specification, i.e. removing nondeterminism. This means the implementation may have fewer possible behaviours than the specification, but certainly not behaviours which the specification does not have.

Extension Adding new information about what to do/how to do something (different observable behaviour). Extension supports incremental specification, moving from a partial specification to a total one [IYK90]. This implies that the implementation will have behaviours which the specification does not have.

Refinement Providing more detail about a particular component, or method of structuring, e.g. changing a method (for efficiency for example) (same observable behaviour).

Examining these steps can tell us something about the sort of relation we need to capture the difference between descriptions. For example, if S_{i+1} is obtained from S_i by *refinement*, preserving the observable behaviour, then some sort of equivalence relation is appropriate. On the other hand, if S_{i+1} was obtained by *reduction* then a preorder relation might be more appropriate since the behaviour of S_{i+1} will be contained within the behaviour of S_i. In the same way, *extension* steps also indicate the use of a preorder relation; in this case the behaviour of S_i will be contained within the behaviour of S_{i+1}. The preorder relation models the case where we want the behaviour of one specification to *approximate* the behaviour of the other.

Unfortunately, in most cases, the step the specifier takes between S_i and S_{i+1} will be larger, and probably a combination of many of all three types of design decision, making the choice between equivalence and preorder less clear cut.

Since we are dealing with LOTOS, there is only one preorder relation to consider, i.e. **cred** (the congruent counterpart of **red**). If, however, the interpretation of implementation indicates an equivalence or congruence relation, we still have to choose between several different equivalence and congruence relations commonly used for LOTOS and presented in detail in the last chapter.

Possible criteria for making this choice are detailed next.

4.2.2 Choosing between Different Equivalence/Congruence Relations

The most commonly used equivalence for LOTOS is weak bisimulation congruence (observation congruence), which is unfortunate since it seems to be too strong for most applications. (It is well known that the differences between distinct processes under observation equivalence are not truly observable [Abr87].) Also used are testing equivalence and, less commonly, failures and trace equivalence [GL91, GLO91]. However, any other equivalence which can be defined over labelled transition systems can also be used for LOTOS, e.g. branching bisimulation equivalence.

Given that such a number of equivalences exist, how can a user wishing to compare two specifications choose which relation is the most appropriate for a particular example? Of course the satisfaction relation must capture the property we wish to express, but it can be difficult to determine exactly what that is. A number of possible criteria for making this choice are given below. Our intention here is *not* to thoroughly explore the question of choosing an equivalence relation; we merely suggest some possible criteria, illustrating them via some selected examples. The exploration of the full implications of these criteria is considered as further work.

Strength In the portion of the linear time — branching time spectrum presented in figure 3.5, section 3.6.2, the relations are compared in terms of the number of identifications they make. We can use the relative strength of relations in terms of identifications to make our choice. For example, a strong equivalence will probably hold if a very short step has been taken between specification and implementation. On the other hand, a weaker relation is more likely to hold than a strong relation if the implementation is very far removed from the specification.

At the very least, when considering proving satisfaction, the user should start with a strong equivalence relation, moving to a weaker relation if the proof fails, if that relation still captures the property to be proved.

Properties Preserved As mentioned in section 3.4.3, logic may be used to describe properties of the system. Some equivalences respect certain properties while others do not. For example, trace equivalence does not preserve deadlock properties, and is therefore unsuitable for verification purposes (since we normally do not want to equate a process which deadlocks with one which does not). Similar relationships may exist between other equivalences and properties. The relationships between the equivalences induced by various logics and the standard process algebra equivalences is investigated in [BR83].

Congruence This was discussed above. Normally we expect that a congruence relation is more desirable in a verification setting than an equivalence relation.

Refinement of Actions Normally actions are assumed to be atomic, but a useful development procedure is to use an action to model a more complex process, substituting that process for the action at a later stage in the development of the specification. Unfortunately, equivalence may not be preserved by this procedure. An equivalence which does allow refinement of actions is branching bisimulation equivalence [vG90].

Axiomatisations In order to carry out axiomatic proofs, it is helpful if the axiomatisation is sound and complete, and, for automation purposes, finite. Given the usual operators of process algebras, it is shown in [Mol90] that no such axiomatisation can exist; the expansion theorem required to express interleaving semantics is actually an axiom schema which expands to give an infinite axiomatisation. A finite axiomatisation can only be obtained by altering the operator set. For example, the addition of the left merge operator to the language can give a finite axiomatisation. Alternatively, we may restrict ourselves to a *subset* of the language. For example, in [BIN92] a complete axiomatisation for finite Basic LOTOS has been derived by translating Basic LOTOS into CCS (and a complete axiomatisation exists for observation equivalence over finite state agents [Mil89a]).

Rewrite Rules Some axiomatisations can be turned into a finite confluent and terminating sets of rewrite rules, while others cannot. Experiments have been carried out in [DIN89] to investigate this for various equivalences, yielding the following results. Those equivalences which do have a corresponding complete rewrite rule set include branching bisimulation equivalence, also discussed in [AB90], and trace equivalence; while those that do not admit a finite confluent and terminating rule set include observation equivalence and testing congruence. A complete rule set may be derived for these latter equivalences in one of two ways: removal of "troublesome" operators, e.g. observational congruence without the parallel operator yields a complete set of rewrite rules, [DIN89], or by application of special term rewriting techniques which allow infinite rule sets to be generalised, giving finite rule sets. This is also discussed in section 6.4.3.

Alternative Characterisations For hand proofs in particular, some methods of proof may be more appealing than others. For example, for weak bisimulation congruence, a proof may be completed by using the axioms, or by exhibiting a bisimulation, or by using logical properties. As another example, while proving testing equivalence does not hold can be relatively easily done by demonstrating that one test exists which differentiates the two processes, it is harder to be sure that all tests have been considered in a proof that the equivalence does hold. In this case we might resort to the alternative characterisation of failure trees (as testing equivalence and failures equivalence are identical). Unlike the other criteria, which are quantifiable in some way, this criterion depends mainly on the subjective opinion of the specifier.

Decidability Given two arbitrary processes, is it always possible to compute whether or not they are equivalent? Obviously if the underlying equivalence is undecidable, then we cannot fully automate the proof. Taking the classification of process algebra equivalences as put forward by [vG90] (the linear time – branching time spectrum), it has been shown in [GH91, Hüt91] that of these, only strong bisimulation equivalence and branching bisimulation equivalence are decidable in general. All other equivalences in this spectrum are undecidable. (This result was shown for a special class of processes, equivalent to the context free languages, called normed BPA (Basic Process Algebra).)

Restrictions may be imposed on the language which ensure decidability, e.g. if the processes are deterministic then completed trace equivalence is decidable. Since all equivalences are the same over deterministic processes, [vG90], all equivalences are decidable over deterministic processes.

Another alternative is that we can relinquish fully automated methods of proof, moving to partial automation (where some measure of user guidance will be required).

Computation How *feasible* is it to compute the equivalence? Although most of the above equivalences are undecidable in general, algorithms have been developed which allow automation of proofs of equivalence for a subset of processes [KS90]. We might consider the efficiency of these algorithms with respect to each other.

Until these criteria are fully investigated we must rely on the method of trial and error in selecting a relation.

4.3 Refinement and Transformation

Above we have considered the situation in which we are given two descriptions and are required to prove them related; however, we may also briefly consider the question of refining or transforming a specification to give an implementation, preserving some relation in the refinement. Again the choice of relation is important, since some relations will preserve properties which others do not.

Obviously, if equivalence relations and preorders can be used to determine whether or not one specification satisfies another, the axioms of an equivalence (or a preorder) can be used to derive S_{i+1} from S_i. It is also possible that the correctness of any transformations which are developed can be expressed in terms of the equivalences above. For example, as part of the LOTOSPHERE project a catalogue of correctness preserving transformations [Bol92] was developed for LOTOS. Several of these transformations preserve weak bisimulation congruence. One of the transformations is discussed in section 10.3.1; we note that special variants of the usual bisimulation equivalences were developed to express correctness of this transformation.

Another example of specification transformation in given in section 10.3.3, in which weak bisimulation congruence is preserved in transforming an abstract data type specification into a process algebra specification. We may also consider transformations within process algebra, e.g. from one specification style to another.

Related to the above question is how putting a specification in a different context may change the behaviour of the system as a whole. For example, if the specification replaces another congruent specification, then there is no change to the observable behaviour. On the other hand, the constraint oriented style of specification is based on the ability to alter a specification by composing it in parallel with a new subsystem. This composition preserves the safety properties of the specification but not the liveness properties [Bri89].

The whole question of refinement and transformation is really a side issue of the main consideration of proving an implementation satisfies a specification, as we assume that the implementation is given, rather than being derived by us from the specification.

We now return to the main question, and consider proof techniques for proving the equivalence/ordering of two process algebra specification, and tools which implement those techniques.

4.4 Proof Techniques and Proof Tools

An important factor in the acceptability of a specification formalism is the quality and type of methods and automated tools with which it is associated. The previous sections have dealt with the theoretical foundation of the verification method of proving two specifications are related. We now consider automated techniques applying this method, and mention specific tools which can perform the comparison automatically (or semi-automatically). The features of these tools are discussed in more detail in appendix A.

There are several approaches to automating the proof of implementation satisfies specification, and many different tools. Here we consider proof tools in general because of the lack of specific LOTOS tools for this approach to verification. Below, these are grouped into those which use *semantic reasoning*, i.e. the approach relies on a deeper understanding of the semantics of the formalism, particularly the details of the equivalence relation, or *syntactic reasoning*, i.e. the approach relies on being able to manipulate the process expressions, without the machine having any knowledge of the underlying semantics. We might also call this the formal approach.

We consider several approaches in detail.

4.4.1 Semantic Reasoning

Partition Algorithms To construct an equivalence, the processes are converted into finite state automata and partition algorithms, such as those given in [PT87, KS83] are used to construct a bisimulation. Equivalences such as testing equivalence can be alternatively expressed in terms of bisimulations by varying the information on the nodes of the graph, see e.g. [CH90]. Systems using this approach to equivalence checking include the Concurrency Workbench [CPS89] and AUTO [MV89], also used as part of the LITE toolkit [LITE].

These systems are limited by the fact that the partition algorithms can only be applied to finite state graphs, therefore not all processes which are equivalent can be proved equivalent using these tools. We note however, that this must be true of any method for equivalence checking, as most of the relations we deal with are undecidable. Constraints on the syntactic structure of a process which ensure finiteness have been developed in [BS87, MV91a]. The partition method also suffers from the state explosion problem, i.e. if the size of the graph gets too big, which happens easily when parallel statements include several components, the algorithm may become too slow to make its use practical.

A further drawback of the original partition method is that it can only give a yes/no answer, i.e. if two processes are not equivalent it cannot supply any reason as to why this is the case (such reasons can be useful in gaining understanding of the specification). One way of expressing the difference between two processes is by finding a modal logic formula which one may satisfy but the other cannot; such a formula is guaranteed to exist [HM85]. [Cle91] refines the algorithm of [KS90], allowing the construction of a distinguishing formula in the case of inequivalence. This applies only to the algorithm for observation equivalence; similar refinements for other equivalences may be discovered in the future.

Backtracking Method This method, described in [Lar86] and automated in TAV [GLZ89], is different from the partition algorithm above in that it tries to find a minimal bisimulation rather than a maximal one. The method involves backtracking and is slower than the partition algorithm. Its main advantage is that reasons for inequivalence are generated automatically (but this can now be accomplished by partition methods also).

Milne's Method This method, described in [MM92] also relies on the representation of the processes by finite state machines, but instead of partitioning to find an equivalence, the equivalence is demonstrated more directly, by composing the processes in parallel and comparing the resultant transition system with that of one of the original processes using tree equivalence. This allows the construction of a distinguishing trace in the case of inequivalence.

This method computes strong bisimulation for deterministic finite state machines (although note that the spectrum of equivalences collapses when nondeterminism is eliminated, so this is the same as trace equivalence) and testing equivalence for nondeterministic finite state machines (by transforming the graph into a deterministic Acceptance tree).

This method is not reliable as the equivalence computed is not exactly the same as testing equivalence.

In summary, of the semantic reasoning approaches to equivalence checking, the most promising seems to be the graph partition method of proof, since it subsumes the backtracking method, and since Milne's method is not sound. However, the graph partition technique has some features which we find undesirable, such as the need for a special representation of processes, and the lack of intuition into the system being investigated given by this approach, due to the lack of meaningful information supplied during the proof.

We now consider approaches to equivalence checking which fall under the syntactic reasoning approach.

4.4.2 Syntactic Reasoning

Symbolic Manipulation Using the technique of *equational reasoning*, the axioms of the equivalence under consideration can be used to manipulate process expressions. Informally, the goal is to prove equivalence by transforming one expression until it has the same syntactic form as the other. The advantages of this approach are that the state explosion problem is avoided, since one equation may encompass several states, but also we can use lazy expansion, expanding only one part of an expression at a time as we choose, rather than indiscriminately expanding everything. Other advantages are that even infinite labelled transition systems are easily dealt with, and that the intermediate stages in the proof are easily understood, which is not true of intermediate partitions. The main disadvantage of this approach is that the proof procedure may not be fully automatic; some user intervention may be required to determine which axiom should be applied next and the direction in which it should be applied. This problem is partially overcome by term rewriting systems, a method of automating equational reasoning, in which the axioms are turned into rewrite rules by orienting them, i.e. giving them a direction. Even with this improvement, full automation is only possible if the rule set forms a decision procedure for the relation modelled by the axioms. The rule set only forms a decision procedure under certain conditions (of course, the decidability of the original relation is important!). Tools which are based on term rewriting systems are therefore usually proof *assistants*, rather than automatic theorem provers.

In this category, there are tools which are general equational reasoning tools which have had special rule sets or tactics built into them to deal with a particular formalism and equivalence, e.g. [CN92] which is based on the LP theorem prover. In addition, [Lin92] is a process algebra specific rewrite tool, i.e. it has been tailored to enable easy manipulation of process algebras, but does not adopt a particular process algebra.

Other systems may start with the rewriting paradigm and build a tool which implements just the equivalences of one particular formalism, e.g. [IN90, DIN89] for CCS. This particular tool also performs proofs in LOTOS by using additional rules to translate finite Basic LOTOS into CCS [DIN91].

Logical Systems The semantics of a process algebra may be expressed in terms of logical formulae, allowing equivalence between processes to be reduced to a logical statement, and therefore allowing the use of a general logic based theorem prover to perform equivalence checking. Examples of such systems include: [Fle87], which uses the Boyer-Moore theorem prover, [Boo89] which uses LCF, and [CR90, CIN91, Nes92] which are based on HOL.

Note that [Nes92] does not fit comfortably into our categories as the work is based on formalising the semantics of CCS in HOL. Nevertheless, this is certainly a formal approach, rather than a graph based one. Once the transition relations are described, and the equivalence relations, proof tactics and operators of the language are defined on top of them, the user has no need to refer to the transitions again.

Verification via Decomposition There has been particular interest in compositional methods, both of specification and verification, particularly since this should make analysis of large systems easier. Compositional methods of verification include partial methods [LT91] and modal specifications [Lar90a]. Both of these methods rely on somehow modifying the transition system (by adding unspecified events to give a partial specification or by adding new sorts of transition relation). Modal specifications use two transition relations, *must* and *may*, and give a rich theory of refinement. Note that if only *must* transitions are used (which is likely in later refinements) then the system is equivalent to a labelled transition system with the usual sort of transitions.

Alternatively, the method of [GM92] allows the truth of equivalence of two specifications to be deduced from the truth of the equivalence of corresponding parts of the specifications using a special operator to project the parts for verification out of the main process. The original transition system remains unchanged. This method is more a symbolic preprocessing, allowing equivalence to be calculated using the partition algorithms above more efficiently. Only a limited number of systems, those in which communication between the components is minimal, are suitable for this transformation.

To summarise the syntactic approach to reasoning about the equivalence of processes, equational reasoning and logically based systems seem the most promising techniques; verification by decomposition is only applicable in a few cases. We now consider the tools and techniques with specific reference to LOTOS.

4.4.3 LOTOS Considerations

Amongst the tools mentioned above, only two are LOTOS specific, [LITE] and [DIN91].

Graph partition methods, as used in [LITE], have the advantage of speed, but as a special representation is used, the proof steps, assuming we can look at intermediate steps in the proof process in the first place, are not informative. In particular, if a proof fails, we may not gain any information as to *why* it failed.

On the other hand, syntactic reasoning, as used in [DIN91] avoids the need for a special representation for processes, and the proof steps, being applications of the axioms of the equivalence, are simple and easy to follow. The normal forms produced in a proof of inequivalence may help identify the reason for the inequivalence as the normal forms are generally more compact and clearer to understand than the original process terms. The main drawback of this approach is the lack of automation and high reliance on the skills of the user, who must frequently guide the proof. This may also be seen as a benefit, as such close interaction may lend additional understanding of the system under examination. We do not adopt the tool of [DIN91] as it deals only with finite Basic LOTOS, and it is our intention to perform proofs over as much of the language as possible. This proof system is claimed to be modular and easily extensible; this is only true if the developer is familiar with Prolog programming. We intend to develop a system in which the user only needs to know about the laws of the LOTOS relation being used, even in the case that new laws or even relations have to be added.

4.5 Summary

In this chapter we have described an approach to verification which allows us to prove that the implementation of a system satisfies the specification of that system, where we assume that both descriptions are written using Basic LOTOS. What it means for one description to satisfy another can be conveniently modelled by one of the LOTOS equivalence, congruence or preorder relations. Since many different relations can be used, the choice of the most appropriate relation for a particular problem is an important decision in this approach to verification. We have presented a number of criteria which may help in this choice.

The satisfaction method of verification may be automated in a variety of ways; we surveyed current techniques and tools in order to identify a suitable tool for our work. Equational reasoning as a proof technique provides a simple proof system for the user to understand, and may also lend extra understanding of the system under investigation, by examination of intermediate stages in the proof. However, the only equational reasoning based tool for LOTOS is applicable to only a subset of Basic LOTOS, and may prove difficult to extend (the user must have knowledge of Prolog as well as of LOTOS).

In view of the deficiencies of this system we decided to develop our own proof system for proving equivalence or ordering of two Basic LOTOS specifications, using equational reasoning as the underlying proof paradigm.

Chapters 6 and 8 detail our attempts to develop an equational reasoning tool to perform proofs of equivalence over LOTOS processes. Equational reasoning is implemented by term rewriting; we base our approach on existing term rewriting tools in order to avoid unnecessary implementation work. To prepare for the description of the development of our tool, we first introduce the basic concepts of equational reasoning and term rewriting which will be used in the following chapters.

Chapter 5

Equational Reasoning, Term Rewriting and LOTOS

In the previous chapter an approach to verification of LOTOS specifications was discussed which requires a proof of the equivalence of two specifications (or that one specification is related via a preorder to the other). We considered various implementation techniques, eventually identifying equational reasoning implemented by term rewriting as our favoured method of automation. The terms "equational reasoning" and "term rewriting" have only been described in a very informal way up till now; it is the purpose of this chapter to give formal definitions of these terms. In addition, we will describe how we plan to use term rewriting in the proofs of equivalence of LOTOS specifications and any special term rewriting techniques that may be required for these proofs.

5.1 Introduction

Equational reasoning provides a means of reasoning about equational specifications, i.e. specifications that are given by a signature (a set of function names and arities) and a set of equations specifying the properties of the functions of the signature. LOTOS processes and the laws relating to the LOTOS equivalences can be viewed as such a specification.

A LOTOS expression is built from variable names, and the function symbols of LOTOS. An expression of the language is also called a *term*. In equational reasoning two expressions (terms) are proved equivalent by applying axioms to one or both until they are syntactically identical. In a typical proof the same axiom may be used several times, applied in either direction, sometimes making the expression smaller, sometimes bigger. Of course, we cannot always transform one term into the other; in this case either the terms are not equivalent, or our axioms or laws do not fully describe the equivalence relation of the mathematical model, i.e. they are not *complete* with respect to the model. In the latter case we may have to define a new axiom, derive a new law, or use a different proof technique altogether to complete the proof of equivalence.

A problem which arises when automating equational reasoning is that a great deal of skill is required when deciding which axiom to apply (for several may be applicable), and in which direction that axiom should be applied (left to right, or right to left). In order to combat this problem, instead of using axioms, we use *rules* which are obtained by orienting the axioms, i.e. by giving the axioms a fixed direction of application, so we decide once only the direction of application for each rule. The rules obtained in this way may then only be applied in that direction. A system of such rules is called a *term rewriting system.* Assuming that the underlying equivalence is decidable, a term rewriting system may give us a decision procedure for the equality of terms. This is not always the case: even if the equivalence is decidable, we also require that the term rewriting system satisfies certain properties.

In the following sections we give some standard definitions of basic term rewriting concepts which will be required for chapters 6 and 8. Further introductory material to term rewriting theory in general may be found in [HO82] and discussion of specific topics may be found in a special issue of the Computer Journal [Com91] devoted to term rewriting.

5.2 Term Rewriting Systems

We begin by giving a formal definition of terms and signatures. A signature is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a set of sorts and $\mathcal{F}$ is a set of functions. Each function consists of a name and arity describing the sorts of the function arguments and result. For example $f : s_1, \ldots, s_n \to t$, where f is the function name, $s_1, \ldots, s_n, t \in \mathcal{S}$ and $n \geq 0$, denotes the function f which takes arguments from each of the s_i and returns a result in t. If $n = 0$ then the function takes no arguments and is known as a constant.

Given such a signature, a term, w, is then constructed from the functions of that signature and variables over the sorts, $w = f(u_1, \ldots, u_n)$ where $f \in \mathcal{F}$ and $u_1, \ldots, u_n$ are also terms (of the correct sort with respect to the arity of f). Subterms of w are w itself and all the subterms of the u_i, $1 < i \leq n$.

To explain the use of term rewriting systems we use a small, familiar, example of a rule set describing addition in the integers. The rule set is defined as follows:

$$x + 0 \to x \qquad \text{R1}$$
$$0 + x \to x \qquad \text{R2}$$
$$-x + x \to 0 \qquad \text{R3}$$
$$(x + y) + z \to x + (y + z) \qquad \text{R4}$$

where x, y, z denote variables.

These rewrite rules are obtained from the corresponding axioms of the system by exchanging the $=$ symbol for a $\rightarrow$ symbol, and by giving the rule a direction of application. Notice that all of the rules in the system given above, with the exception of the associative rule, have smaller terms on the right hand side of the $\rightarrow$ than on the left hand side. The associative rule is ordered so that the brackets group to the right.

Although the direction can be given manually by specifying for each rule whether it should be applied left to right, or right to left, normally an *ordering* is used to describe the direction of the rules in a consistent manner. In general we use standard pre-defined orderings such as Knuth-Bendix Ordering (KBO) and Recursive Path Ordering (RPO), or variants of these. We do not give details of these orderings here; the reader need only know of their existence. Further information on orderings may be found in [Der82]. The ordering provides a method of evaluating the complexity of a term, and usually the rules are oriented so that the left hand side of the rule is more complex than the right hand side.

Assuming we have constructed a set of rewrite rules as above, how are they used? The basic step is a *reduction* or *rewrite*. Given a rule, $l \longrightarrow r$, and a term, $t = f(u_1, \ldots, u_n)$, $n \geq 0$, we compare l and t. We may apply a substitution σ to l which exchanges the variables in l for other terms and/or variable names so that $\sigma l = w$, where w is a subterm of t. This process is known as *matching*. The rewriting continues by replacing w in t by σr.

Iteration of this process is called *reducing* or *rewriting*. We write $t_1 \longrightarrow^* t_n$ for the sequence $t_1 \longrightarrow \ldots \longrightarrow t_n$, where $n \geq 0$. *Termination* means that there are no infinite reductions, i.e. rewrite sequences of the form $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$. If the ordering used to orient the rewrite rules is *well-founded* then termination of the reduction process is guaranteed, i.e. well-foundedness guarantees there are no infinite decreasing sequences of terms in the ordering, so eventually we must reach a term to which no rule applies.

The process of reducing a term until no more rules can be applied is called *normalisation*, and the final term obtained is called the *normal form*. Usually the rule set contains more than one rule, so the rewrite step compares the left hand side of each of the rules to t in turn, looking for a match. Also, more than one rewrite rule may be applicable to a given term, therefore rewriting is nondeterministic.

For example, given the rule set above, consider the term $a + 0$. Using the rule R1 and substituting a for x, we can rewrite $a + 0$ to get a, i.e. $a + 0 \longrightarrow a$. Since none of the rules has a left hand side which matches a, this is a normal form.

A *rewrite proof* consists of rewriting two terms until their normal forms are obtained, i.e. no more rules can be applied. If the two terms obtained at the end of the process are syntactically equivalent, then the original terms are equivalent in the semantics. This process is illustrated in figure 5.1. The terms s and t are equivalent since they both rewrite to the normal form u. If we assume that the rule set forms a decision procedure for the equivalence, then the normal forms are unique, therefore if the normal forms resulting from the rewriting process are different the original terms are not equivalent in the model.

$$\begin{tikzcd}
s \arrow[dr] & & & & & & & & t \arrow[dl] \\
& s' \arrow[dr] & & & & & & t' \arrow[dl] & \\
& & s'' \arrow[dr] & & & & t'' \arrow[dl] & & \\
& & & \cdots \arrow[dr] & & \cdots \arrow[dl] & & & \\
& & & & u & & & &
\end{tikzcd}$$

Figure 5.1: A Successful Rewrite Proof

Unfortunately, the procedure is not always as easy and straightforward as described in figure 5.1. Even assuming the original equivalence is decidable, a set of rewrite rules derived by orienting the equations of the equivalence only yields a decision procedure for the equivalence if the rule set satisfies certain properties. If the rule set does not satisfy these properties, then the normal forms are not unique, and we cannot deduce from the failure of a rewrite proof that the original equation does not hold, i.e. we have a semi-decision procedure. The reason for this loss of power is that when using the equations we can apply them in either direction, whereas we restrict rules so that they may be applied in one direction only. In forcing the equations to be unidirectional rewrite rules we have lost the power of the original equational system.

In the next section, we discuss this problem, and a partial solution, further.

5.3 Knuth-Bendix Completion

If a rewrite rule set derived from a set of equations is to give a decision procedure for the equivalence relation expressed by those equations, it must satisfy two properties: the rule set must be *confluent* and *terminating* (termination was defined above, confluence is defined below). If these properties hold we say that the rule set is *complete* or *canonical.* These properties ensure that every term has a *unique* normal form and equations which can be proved to hold in the equational system can also be proved to hold by the rewrite rules. Conversely, equations which cannot be proved to hold by the rewrite rules do not hold in the equational system.

A term rewriting system is *confluent* if, and only if, $\forall t_1, t_2, t.\, t \rightarrow^* t_1 \ \wedge\ t \rightarrow^* t_2 \ \Rightarrow\ \exists t'.\, t_1 \rightarrow^* t' \ \wedge\ t_2 \rightarrow^* t'$. This is also known as the diamond lemma for obvious reasons:

$$\begin{tikzcd}
& t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\
& t' &
\end{tikzcd}$$

Figure 5.2: Confluence

Confluence means that if there are two ways to rewrite a term, it doesn't matter which one is chosen as eventually both will rewrite to the same term. *Local confluence* is like confluence except that we replace $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$ in the above by $t \rightarrow t_1$ and $t \rightarrow t_2$, i.e. t_1 and t_2 can be obtained by exactly one reduction from t.

The properties of confluence and termination are undecidable in themselves. However, Newman's theorem [New42] shows that a terminating term rewriting system is confluent if, and only if, it is locally confluent.

A procedure exists, due to Knuth and Bendix [KB70], which checks a rule set for local confluence, and if the set is not locally confluent, adds rules to try to make it locally confluent. Used in conjunction with a termination ordering, the procedure may produce a *complete* rule set, hence the procedure is known as completion, or Knuth-Bendix Completion.

The completion procedure is based on the examination of *critical pairs*. A critical pair is a pair of terms generated by applying two different rewrite rules (or two different applications of the same rewrite rule) to a term (the *critical expression*). For example, assume a rewrite system which allows the term t to be reduced as in figure 5.3.

$$\begin{tikzcd}
& t \arrow[dl] \arrow[dr] & \\
t_1 & & t_2
\end{tikzcd}$$

Figure 5.3: A Critical Pair

The critical pair is (t_1, t_2). Assuming t_1 and t_2 cannot be further reduced, the rule set is not confluent. To correct this we must add a new rule to the rule set, $t_1 \longrightarrow t_2$. The main theorem of [KB70] says that a terminating term rewriting system is locally confluent if, and only if, every critical pair is trivial. An example of a trivial critical pair is (t, t), i.e. the terms are the same, or have the same normal form.

Critical pairs are generated from rewrite rules automatically by *superposition*. This means *unifying* subterms. Unification is the process of matching where substitutions may be applied to both terms, i.e. we look for σ_1 and σ_2 such that $\sigma_1 l_1 = \sigma_2 l_2$. Rules may also be superposed onto themselves.

Knuth-Bendix completion works by generating all critical pairs and adding non-trivial pairs as rules until no more non-trivial critical pairs are generated. In this way, if Knuth-Bendix completion is successful, a non-confluent system may be turned into a confluent one. Of course, the completion procedure is not always successful.

Three things can happen as a result of running the Knuth-Bendix completion procedure:

1. The algorithm may halt, leaving a confluent and terminating set of rewrite rules, giving a decision procedure for the original equivalence.

2. The algorithm may halt because a critical pair has been generated that it cannot orient.

3. The algorithm may diverge, i.e. superposition of the rules generates an infinite number of new non-trivial critical pairs and therefore new rules.

The second case *may* be solved by adopting a different termination ordering. There are several methods of attempting to solve the third case, i.e. divergence of Knuth-Bendix completion; this topic is discussed further in section 6.4.3.

Even if we have a set of rules which is not complete, we can still use this set of rules for rewrite proofs, but we will not be able to prove that an equation does *not* hold. In order to prove an inequation, we must use some other proof technique. Note that we can use rewriting to produce normal forms of the terms of the inequation, possibly simplifying the application of the other proof technique, i.e. rewriting with a non-complete set of rules is *sound*.

A further introduction to the ideas of Knuth-Bendix completion may be found in [Dic91].

5.4 Extensions to Term Rewriting

There are three extensions to the basic paradigm of term rewriting which are of interest to us: *order-sorted* rewriting, rewriting modulo a set of equations, and *conditional rewriting.*

Typically, term rewriting systems have one sort, the universal sort, and all terms belong to that sort. In order-sorted rewriting we can define subsorts, giving a sort structure. Using this structure we may declare function arities more accurately, which may help clarify the specification. Additionally, rewriting may be restricted to allow terms to be rewritten only to the same sort or to a subsort.

The next extension is concerned with specific equations; those, such as the commutativity axiom, which cannot be oriented in the conventional rewriting framework. To get around this problem rewriting modulo a set of equations was introduced. The special equations are held separately from the rest of the rewrite rules. Since the special equations most often express associativity and commutativity this is also known as *a-c* rewriting.

Finally, we may wish to define rules which only hold in certain situations, i.e. they have side conditions. Again, a special modification of term rewriting allows us to deal with conditional rules.

5.5 Application to LOTOS

As the LOTOS equivalences are given by a set of laws, equational reasoning is an obvious choice of proof technique. Although most of the LOTOS equivalences are not decidable, as discussed in section 4.2.2, and therefore no corresponding complete rule set can exist, we may be able to find a complete rule set for a subset of the language. Finding this rule set will require the application of the Knuth-Bendix completion procedure. Since some LOTOS operators are associative and/or commutative, we should try to use a term rewriting system which implements rewriting modulo a-c equations. The RRL (Rewrite Rule Laboratory) term rewriting system [KZ87] has these features. RRL also has a limited form of order-sorted rewriting in that operators may be given sorts, and subsorts may be defined, but variables have the universal sort. Order-sorted rewriting is required by some methods of solving divergence in Knuth-Bendix completion. RRL also has conditional rewriting. Other rewriting tools are available, see [HKK91] for a survey; however, none have all the features we require.

5.5.1 Using Term Rewriting Techniques in Other Ways

An interesting application of term rewriting to concurrency protocols described as finite state machines (fsms) appears in [RS91] where the properties of a fsm are shown to have a direct correspondence with the properties of the term rewriting system implementing that fsm.

First the system to be investigated is described as a set of rules giving the transition relation for the fsm. We can then check that rule set for properties such as completeness, confluence and termination using a standard term rewriting tool and use this information to deduce the corresponding properties of the fsm. For example, if the rule set is complete, then the protocol described by the fsm is complete, i.e. all possible input scenarios are described, and it makes progress. If the rule set is terminating then the protocol is bounded and so on. The Knuth-Bendix completion procedure can be used to check that the rule set has the desired properties, and may also be used to supply the extra rules required by a deficient rule set to ensure that the rule set has these properties. This work was carried out using the RRL system [KZ87].

Also, specific properties such as "input x leads to state s, or to output y" may be expressed as equations (over states of the finite state machine) and shown to be consistent with the original rule set, i.e. they are valid with respect to the original equational theory. The equation may contain variables, in which case either it holds for all possible instantiations of the variables, or RRL can be used to supply the particular values for which the equation holds.

This method has limitations in that Knuth-Bendix completion is not guaranteed to terminate, and the method of description restricts the applicability to finite state machines only, therefore we would have to translate finite LOTOS expressions into a finite state machine to use this technique. Furthermore, the finite state machine gives a fairly low-level description of a system, and we would prefer to express our properties/equations etc. to RRL using normal LOTOS syntax. It is also difficult to see how higher-level language constructs such as parallelism could be accommodated in this framework. These limitations deter us from adopting this approach to verification of concurrent systems.

5.5.2 Soundness of the Laws of [ISO88]

One of the common applications of term rewriting is to prove the consistency of a specification: for boolean terms, if a rule set is not consistent the completion procedure will (eventually) generate the rule *true* $\longrightarrow$ *false*. During our attempts to find a canonical rule set corresponding to the laws of weak bisimulation congruence for LOTOS, which are described in the next chapter, we also managed to find an inconsistency in the laws of [ISO88], i.e. the laws are not sound. Fortunately, this inconsistency can be corrected. In this section we discuss how completion uncovers this inconsistency and our correction of the laws.

Before we began our experiment we were aware of a possible inconsistency in the LOTOS laws for weak bisimulation congruence as given in [ISO88]. Knowing of this problem helped greatly; as we shall see, inconsistencies in term rewriting systems containing sorts other than booleans are less obvious than the *true* $\longrightarrow$ *false* example above.

The inconsistency can be detected as follows. We give the following rules, derived from d4, c3b and d2a of the LOTOS standard,

$$\begin{array}{rcl} B \gg \mathbf{stop} & \longrightarrow & B \mid\mid\mid \mathbf{stop} \\ \mathbf{exit} \mid\mid\mid \mathbf{stop} & \longrightarrow & \mathbf{stop} \\ \mathbf{exit} \gg B & \longrightarrow & \mathbf{i}; B \end{array}$$

to RRL and invoke the completion procedure. The following rule is derived:

$$\mathbf{i}; \mathbf{stop} \longrightarrow \mathbf{stop}$$

It is not immediately obvious that this rule is not valid; however, an example will demonstrate that although valid under weak bisimulation equivalence, this rule is not valid under weak bisimulation congruence as the two processes do not have the same behaviour in all contexts. It is necessary only to consider the behaviour of the processes when part of a choice expression, i.e. we must show $B \, [] \, \textbf{stop} \not\equiv_{wbe} B \, [] \, \textbf{i}; \textbf{stop}$, for some B. Consider the corresponding equation with $B = \textbf{i}; \textbf{exit}$, i.e.

$$(\textbf{i}; \textbf{exit}) \; [] \; \textbf{stop} \; \equiv_{wbe} \; (\textbf{i}; \textbf{exit}) \; [] \; (\textbf{i}; \textbf{stop})$$

By the reduction derived above, $\textbf{i}; \textbf{stop} \longrightarrow \textbf{stop}$, this equation can be shown to hold in RRL; however, by considering the behaviour of each side of the equation, it is clearly false. While the left hand side will always terminate successfully, the right hand side may either terminate successfully or deadlock, since $[]$ is forced to be nondeterministic in the presence of **i** actions.

The law in the standard will be sound if a side condition is added:

$$B \gg \textbf{stop} \equiv_{wbc} B \, ||| \, \textbf{stop} \qquad \text{where } B \neq \textbf{exit}$$

We use this form of the law in the remainder of this thesis.

5.6 Summary

This chapter introduced the basic concepts of equational reasoning and term rewriting. We also discussed how these proof techniques will be used for LOTOS proofs, how they contributed to finding an unsound law in the standard [ISO88] and described a previous use of term rewriting techniques for verification of concurrent systems [RS91].

The next section describes how we used a particular rewrite tool to develop a complete set of rules for use with LOTOS, and how these rules can be used in rewrite proofs.

Chapter 6

Using RRL to Implement LOTOS Weak Bisimulation Congruence Laws

6.1 Introduction

This chapter describes the use of Knuth-Bendix completion to try to find a *canonical* rule set corresponding to the laws of LOTOS weak bisimulation congruence. We take the laws of weak bisimulation congruence as given in the LOTOS standard [ISO88] as our starting point. Since weak bisimulation congruence is undecidable, there will be no complete rule set corresponding to the equivalence of the model; however, we are able to find a rule set corresponding to a subset of the laws of weak bisimulation congruence which is complete. The term rewriting system used for this work is RRL (Rewrite Rule Laboratory) [KZ87]. The use of the rule set is illustrated by some small, simple, rewriting proofs. From these examples, it is clear that the rule set suffers certain deficiencies; we attempt to correct these by adding new rules but in doing so lose completeness. Finally, we discuss the problem of using a rule set which is not complete in rewrite proofs, and also consider some laws we would like to add to the rule set, but which cause the completion procedure to diverge.

6.2 Implementing LOTOS Laws as Rules in RRL

Our first experiment with the LOTOS equivalences is to attempt to obtain a complete set of rules for weak bisimulation congruence, starting with the laws as given in the LOTOS standard [ISO88].

In the following sections we use two numbering schemes for the rules. In most cases, we refer to a rule by the number of the corresponding law in the standard, this will be of the form αn, where α is an alphabetic character from $\{$a ... m$\}$ and n is a number, e.g. d1. Sometimes the code will have an additional letter, for subdivisions of a law, e.g. d2a. We also use the number assigned to the rule by RRL for rules generated by the completion procedure which do not appear in [ISO88].

This work is also reported in [Kir91] and [KN91]. Here we give a slightly modified version of those results, allowing a more coherent presentation.

6.2.1 Basic Rule Set for Weak Bisimulation Congruence

The first decision to be taken is to choose which laws from the LOTOS standard are to be adopted as rules. In RRL, all data structures to be used have to be defined by the user; the only built-in data type is *Boolean*. In this first experiment, no auxiliary data types such as lists or sets are used, therefore operators relying on these types, such as **hide** and relabelling, are largely ignored; the only exceptions to this are equations requiring base cases of the types, i.e. the empty list or set, or which do not require investigation of the values, i.e. the value of the list or set is not important to the equation. We also do not yet deal with recursive processes: recursion cannot be easily expressed in the rewriting framework, and this problem is discussed in more detail in chapter 8.

The set of rewrite rules is formed by adding rules derived from the laws in the LOTOS standard one at a time, discarding laws that "misbehaved", i.e. cause divergence of the completion procedure. We also have other factors, such as operator precedence to consider, see below. Other methods, resulting in different rule sets, could be used; for example we could include only laws relating to a particular set of operators. Since our goal is completion we choose to add just those rules which do not interfere with achieving this goal. Although this results in a rather haphazard rule set it is at least a *complete* rule set. The "bad" rules, those which cause divergence, are discussed further in section 6.4.3.

The laws eventually chosen and run through the completion algorithm are as follows, using the numbering scheme of the LOTOS standard (also used in appendix B): b1, b3, b4, c3a (particular case where **exit** takes no parameter), c3b, d1, d2a, d3, e2, e3, e4, e5, k1, k2, m1, m2 and m3. We refer to this rule set as *core*. The rules corresponding to these laws are given in figure 6.1, with the exception of law b1. The RRL system keeps a special set of operators which are declared to be commutative and commutative-associative and therefore has no need of the actual laws expressing commutativity or associativity. Note that associativity on its own cannot be expressed using the special declaration.

We have edited the input files to give a more familiar, LOTOS, look in the presentation. The actual input files may be found in appendix C, together with details of the relationship between the two representations (LOTOS and RRL).

[b3]	P [] stop	$\longrightarrow$	P
[b4]	P [] P	$\longrightarrow$	P
[c3a]	exit |[*A*]| exit	$\longrightarrow$	exit
[c3b]	exit ||| stop	$\longrightarrow$	stop
[d1]	stop >> P	$\longrightarrow$	stop
[d2a]	exit >> P	$\longrightarrow$	i; P
[d3]	P >> (Q >> R)	$\longrightarrow$	(P >> Q) >> R
[e2]	P [> stop	$\longrightarrow$	P
[e4]	stop [> P	$\longrightarrow$	P
[e3]	(P [> Q) [] Q	$\longrightarrow$	P [> Q
[e5]	exit [> P	$\longrightarrow$	exit [] P
[k1]	stop[S]	$\longrightarrow$	stop
[k2]	exit[S]	$\longrightarrow$	exit
[m1]	u; i; P	$\longrightarrow$	u; P
[m2]	P [] (i; P)	$\longrightarrow$	i; P
[m3]	(u; (P [] (i; Q))) [] (u; Q)	$\longrightarrow$	u; (P [] (i; Q))

Figure 6.1: *core* set of rules

The following information is also given to RRL: function arities as given in appendix C, declaration of [] as commutative, >> associates left to right, allowing orientation of rule d3, and function precedences: >> > ; and [> > []. These precedences ensure that rule d2a and e5 respectively are oriented left to right. We note here that it is possible to use different precedences, and that other precedences give different results, for example, an infinite sequence of rules is produced if all precedences are reversed and >> associates right to left. The precedences above are chosen because these particular directions make sense for rules d2a and e5 if we assume that we are trying to push occurrences of the choice and sequencing operators out, and therefore occurrences of operators such as disable in. We should eventually be able to remove the higher level operators. The precedences chosen also give the smallest canonical rule set.

6.2.2 Result of the Completion Procedure

The completion procedure, when given the above information, terminates, producing the following extra rules:

[17]	Q [] (Q [] exit)	$\longrightarrow$	Q [] exit	[e5,e3]
[18]	(i; Q) >> R	$\longrightarrow$	i; (Q >> R)	[d2a,d3]
[19]	u; Q [] u; (P [> (i; Q))	$\longrightarrow$	u; (P [> (i; Q))	[e3,m3]

The numbers on the right indicate the rules from *core* which generate the critical pair giving the new rule.

We use the name *newcore* to denote the set of rules comprising the *core* set plus the rules generated by the completion algorithm. The rule set *newcore* is complete, i.e. confluent and terminating.

The next section illustrates the use of the rule set *newcore* in rewrite proofs.

6.3 Equational Proofs — The Buffer Example

Consider the example given in figure 6.2. The aim of this example is to prove that the behaviour of a two-way buffer is the same as the behaviour of two one-way buffers in parallel. Unfortunately the example is a little forced as recursion is not part of our restricted LOTOS; this means that the buffers in figure 6.2 only handle one or two data elements, depending on which buffer is being considered, and then stop.

```
    in1; (   in2; (   out1; out2; exit
                   [] out2; out1; exit)
          [] out1; in2; out2; exit)         ≡wbc     (in1; out1; exit)
 [] in2; (   in1; (   out1; out2; exit              |||(in2; out2; exit)
                   [] out2; out1; exit)
          [] out2; in1; out1; exit)
```

Figure 6.2: Buffer example

The right hand process in this example is far easier to read than the one on the left. This demonstrates that simply defined processes in parallel are frequently easier to understand than large process written without parallelism (a principle of the constraint-oriented style of specification).

The *newcore* rules generated above by Knuth-Bendix completion are not enough to prove the buffer example; laws relating some of the higher level constructs (such as parallelism) to the more basic constructs are also required. Such laws are known as the ***expansion laws*** and describe how any LOTOS process can be rewritten to a form using just ; and []. The *particular* expansion law required here is law nl of the LOTOS standard, also given in figure 6.3.

$$
\begin{aligned}
B \,|[A]|\, C \equiv_{wbc}\; & [] \{b_i;\ (B_i \,|[A]|\, C) \mid \text{name}\ (b_i \notin A, i \in I\} \\
& [] \; [] \{c_j;\ (B \,|[A]|\, C_j) \mid \text{name}\ (c_j \notin A, j \in J\} \\
& [] \; [] \{a;\ (B_i \,|[A]|\, C_j) \mid a = b_i = c_j,\ \text{name}\ (a) \in A, i \in I, j \in J\}
\end{aligned}
$$

where $B = \{b_i;B_i \mid i \in I\}$ and $C = \{c_j;C_j \mid j \in J\}$

Figure 6.3: The Expansion Law for Parallelism, law (nl) of [ISO88]

This law cannot be entered straight into RRL because of its complexity, i.e. use of choice over an indexed set of processes, so some simple instantiations of this law are generated by hand and added to the set *newcore* in RRL. These new rules, call them *little-exp*, are oriented left to right. The rule set formed by *newcore* and *little-exp* is no longer canonical; the completion procedure diverges.

```
a1; P1 ||| a2; P2    ==       a1; (P1 ||| a2; P2)
                           [] a2; (a1; P1 ||| P2)
```

```
a; P ||| exit        ==   a; (P ||| exit)
exit ||| a; P        ==   a; (exit ||| P)
```

Although no longer complete, the rule set can be used to prove the equation in figure 6.2 holds.

6.4 Adding Other Rules

In the Buffer example, extra rules had to be added to turn an expression involving parallelism into one involving only sequencing and choice in order to complete the proof of equivalence. In that particular case only three rules had to be added. Looking ahead, since the rules in *little-exp* do not express the full generality of the expansion law, other (larger) examples will certainly require more such rules. If using the rules *newcore* plus *little-exp* as a proof system, adding rules on an ad-hoc basis is undesirable, therefore what is required is a set of rules which will be generally applicable, i.e. able to translate any expression involving parallelism into its sequential equivalent, rather then an ever-increasing number of specific instantiations of the expansion law. Section 6.4.1 below details our development of a set of rules which implement the full expansion law for parallelism and hide and discusses the problems created by this rule set.

The expansion laws are not the only LOTOS weak bisimulation congruence laws omitted from *newcore*. As mentioned in section 6.2.1, we also rejected laws which caused divergence of the Knuth-Bendix completion procedure. Although such divergence generates an infinite set of rules, in some cases it may be possible to *generalise* this set, i.e. if a pattern may be detected in the form of the rules, it may be possible to find a (finite) re-expression of those rules.

Section 6.4.3 details some of the rules which, together with *core*, cause divergence of the completion procedure. We also present portions of the corresponding infinite rule sets generated and our attempts to find patterns in those rule sets. Only in one case does this allow us to add rules eliminating the cause of divergence; however, the other examples have been used by our colleagues on the SERC project "Verification Techniques for LOTOS" in order to illustrate the application of special term rewriting techniques which attempt to generalise infinite sequences of rewrite rules [TW93, Wat92]. Finally we discuss the possibility of adding rules which describe the behaviour of the relabelling operator more fully than those in *core*.

6.4.1 Developing The Expansion Rules

The Expansion Law for Parallelism

In this section we give an overview of the definition of rules expressing the generality of the expansion laws, in particular the expansion law for parallelism. The full RRL input files for these rules may be found in appendix C.

First consider the general form of this expansion law as given in figure 6.3. Two sorts of events are considered: events which may proceed independently, and events which must synchronise. In the expansion law the events which may proceed independently are dealt with by the first and second branches of the law; these are all initial events of B which are not in A, and all initial events of C which are not in A, respectively. The events which must synchronise are dealt with by the third branch of the law; these are events which are initial events of both B and C and are also in A.

The division of the law by type of events is also reflected in the RRL implementation; the function *expanda* deals with events which may occur independently, and the function *expandb* deals with events which must synchronise. These functions are declared as follows:

expanda : procset, procset, gatelist → procset
expandb : procset, procset, gatelist → procset

where *procset* is a set of processes.

The rules defining these functions follow the style of functional programming, i.e. each rule is defined by pattern matching over its arguments. To cut down the number of patterns which must be considered we assume that, as in figure 6.3, the process arguments B and C are formed by a generalised choice, denoted *gch*, over a set of processes and moreover, that the only operators used are **stop**, **exit**, ; and event constants. Since all LOTOS processes can be expressed in this way this does not restrict the scope of the rules.

The general rule for the expansion law is

par (pp(gch(x),gch(y)), v) == gch(expanda(x,y,v) ++ expanda(y,x,v) ++ expandb(x,y,v))

where ++ is similar to the set union operator. The three branches in the right hand side of the rule correspond to the three branches of the expansion law as given in figure 6.3. In addition to this rule, we must also define rules which correspond to special patterns in the input; namely B and/or C equal to **stop**, **exit**, or simple processes $b; B$ and $c; C$. These are similar to the laws in *little-exp*.

The individual *expand* functions operate as follows:

expanda This function takes both process arguments, B and C, and also the list A. The result of the function is a set of processes whose initial elements may proceed independently.

The function examines each component of the first set of processes recursively and, depending on whether or not the initial action may proceed independently, either adds a new component to the result set or proceeds to the next component. For each process examined there are two possible outcomes; either the process is **stop**, **exit** or $b_i; B_i$, where $b_i \in A$, in which case no action may proceed independently so no process is added to the result set, or the process is $b_i; B_i$,where $b_i \notin A$, and the process $b_i; (B_i | [A] | C)$ is added to the result set.

This function implements the first and second branches of the law in figure 6.3 by switching the first and second arguments accordingly.

expandb This function again takes both process arguments and the list A. The result of applying *expandb* is a set of processes which proceed by synchronising with each other.

The task is to compare pairs of processes from B and C to determine whether any initial actions from these processes may synchronise. *expandb* strips one process at a time from B and gives it to the auxiliary function *expandc* together with the whole set of processes C and the gate list A. *expandc* compares the process with each component of the process set C in turn, producing the set of processes which may proceed by synchronisation. *expandb* combines the results from repeated applications of *expandc*.

expandc is declared

expandc : process, procset, gatelist $\rightarrow$ *procset*

For each iteration, assuming we have the first argument $b_i; B_i$, and $c_j; C_j$ from the set, *expandc* compares b_i and c_j. If they are the same and also in A then the process $b_i; (B_i \,|\,[A]\,|\, C_j)$ is added to the result set, otherwise nothing is added and we go on to the next member of the set. If either process is **stop** then again nothing is added and we go on. Finally, if both processes are **exit** with the same exit list then **exit** is added to the result set of processes.

The rules introduced to "implement" the expansion law are essentially a functional program and therefore are deterministic; at any point, only one rule is applicable. The correctness of these functions is not formally proven, since initially we plan to test the correctness of the rule set in use, i.e. validation to increase our confidence in the rules, with formal verification of their correctness coming later. As shall be seen in the next chapter, deficiencies in RRL lead us to adopt a different tool for our proofs. This new tool has the expansion law built in therefore our *expand* rule set becomes redundant, and a formal correctness proof becomes a rather pointless exercise.

To implement these functions we also have to introduce rules for the standard functions on sets and lists (as generalised choice requires sets, and the parallel operator is expressed using a list of gates; lists are also required for the **hide** expansion law). We do not present the functions or details of the rules which implement them here; see appendix C for the full rule set and some explanation of their operation.

The expansion law for hide is also added. Apart from requiring the definition of lists, the implementation of this law is unremarkable; it is not as complex as the law for parallelism, and requires no auxiliary operators or functions.

We note that the rule sets now obtained, *expand* and *hide* respectively, are not complete; this is a result of the constructive way in which the functions are defined. This means that it is always possible to make more superpositions, and hence more rules.

An example of the use of the rule sets formed by *newcore*, *expand* and *hide* in proofs by rewriting may be found in the Login case study detailed in chapter 7.

6.4.2 Strategy in Applying the Expansion Rules

When using RRL with the rules *newcore*, *expand* and *hide* for proofs, we discover that the rules are best split into two groups: one with *expand* and one with *newcore* plus *hide*. It is also important to use these two sets in the correct way, otherwise the normalisation process diverges: this is a result of not having a complete set of rules. We rely on the system applying all the *newcore* and *hide* rules before trying to apply the *expand* rules, and vice versa: all the expansion rules must be applied before trying to apply the *newcore* plus *hide* rules. This is because, although the expansion laws will eventually produce the correct expansion of a given term, there are a number of intermediate stages in the expansion which do not correspond to well-formed LOTOS processes. Ultimately, the need to have more control over the application of rules becomes more important as we move further away from a complete rule set. RRL cannot provide the facilities to do this.

6.4.3 Laws causing Infinite Sequences of Rewrite Rules

Other laws we might wish to have in our rule set for weak bisimulation congruence include: f8 (**hide** distributes over $\gg$), b2 (associativity of $[]$) and e1 (associativity of $[>$). These laws cannot be included because they cause divergence of the completion procedure, i.e. they generate an infinite sequence of rewrite rules.

Some special term rewriting techniques [TJ89, Lan89, TW93] exist which allow (some cases of) infinite rewrite rule sets to be *generalised* by a finite rule set. These techniques require either an extension of the underlying term rewriting theory or the addition of extra operators or sort structure to the definition of the rule set. In all techniques, the first step of the technique is to identify a pattern in the infinite sequence of rules. From this pattern it may be possible to express a general rule, or set of rules, which subsumes the infinite sequence, i.e. to make a generalisation. Of course, the trick is to ensure that this generalisation is *exact*, i.e. it only produces rules which belong to the infinite sequence, and does not introduce new rules. An exact generalisation may not exist.

Below, we consider the infinite rule sets given by the completion procedure on *core* plus each of the rules mentioned above in turn. In each case we give some representatives of the infinite rule set and try to detect a pattern in these rules. In most cases we give fairly informal over-generalisations, i.e. the rules fit the pattern, but the pattern also generates other rules which are not in the infinite sequence. Fuller details of the rule sets and of our attempts to generalise them may be found in [KN90]. These investigations were made in the hope that a simple solution to the problem of divergence could be found, as is the case for the first example below. In the other (unsuccessful) cases, there appears to be no simple solution, and we do not pursue the problem here. These infinite rule sets provide example material for the special techniques for solving (some cases of) divergence of Knuth-Bendix completion developed by our colleagues on the SERC project "Verification Techniques for LOTOS", see [TW93, Wat92].

Distributivity of hide over $\gg$

This law, f8, is just a representative of a family of similar laws involving **hide** or **relabel** and the other operators.

```
hide v in (P1 >> P2) == (hide v in P1) >> (hide v in P2)
```

This law is added to the set *core*, given left to right orientation by declaring the precedence **hide** $>$ >>. The infinite sequence generated by Knuth-Bendix completion includes the following rules, where the first 16 are just the rules of *core*:

```
[f8]  hide A in (P >> Q) ⟶ (hide A in P) >> (hide A in Q)
[19]  (hide A in stop) >> (hide A in Q) ⟶ hide A in stop          [d1, f8]
[20]  (hide A in exit) >> ((hide A in exit) >> hide A in P)
      ⟶ (hide A in exit) >> (hide A in P)                         [m1, 18]
[24]  (hide A1 in (hide A in stop)) >> (hide A1 in (hide A in Q))
      ⟶ hide A1 in (hide A in stop)                               [19, f8]
[35]  (hide A in stop) >> ((hide A in Q) >> R)
      ⟶ (hide A in stop) >> R                                     [19, d3]
  ⋮
```

It can be seen that some terms in these rules take the form P_n, where

$$\begin{aligned} P_n &= \textbf{hide } A_n \textbf{ in } P_{n-1} \\ P_0 &= \textbf{hide } A \textbf{ in stop} \end{aligned}$$

or Q_n, defined similarly to P_n, but with **exit** instead of **stop** in Q_0. We also note that these terms occur on the left hand side of a $\gg$ operator. Intuitively, it seems that hiding gate names in **stop** or **exit** should have no effect, but there are no rules in [ISO88] which allow, for example, P_n to be rewritten to **stop**. What we do have is the following law:

$$\textbf{hide } A \textbf{ in } P = P \qquad \textit{where } A \cap \mathcal{L}(P) = \emptyset$$

This law, f4, which allows applications of **hide** to be removed if the actions to be hidden are not in the process P anyway, was passed over originally because it requires conditional rules. Using the facts $\mathcal{L}(\textbf{stop}) = \emptyset$ and $\mathcal{L}(\textbf{exit}) = \emptyset$, we generate two instantiations of this law by hand to give the rules:

```
[f4a] hide A in stop ⟶ stop
[f4b] hide A in exit ⟶ exit
```

In addition, we know from rules d1 and d2a that occurrences of **stop** or **exit** on the left hand side of ≫ can be eliminated.

Given rules f8, f4a and f4b the completion procedure terminates producing a canonical set of rules comprising the rules originally generated to give *newcore*, and one more rule:

```
hide A in i; P ⟶ i; hide A in P                                 [d2a, f8]
```

This is in fact one of the laws of weak bisimulation congruence in the standard, number (f5a). We note that this rule set is not canonical if rule f8 is oriented right to left, i.e. if >> > **hide**.

This example shows that some infinite sequences may be easily eliminated.

Associativity for []

In *core*, [] is declared to be a commutative operator. If [] is declared to be associative as well as commutative the completion procedure diverges. Below is an initial portion of the infinite sequence generated when completion is attempted of *core* with [] associative and commutative. The first 16 rules are the same as those in figure 6.1.

```
[18]  R [] (P [> (R1 [] R)) ⟶ (P [> (R1 [] R))                  [e3, b4]
[19]  R [] i; (R1 [] R) ⟶ i; (R1 [] R)                          [m2, b4]
[20]  R [] (P [> (P1 [> R)) ⟶ (P [> (P1 [> R))                  [e3, e3]
[21]  R [] (P [> (R1 [] (P2 [> R))) ⟶ P [> (R1 [] (P2 [> R))    [e3, e3]
[22]  R [] (i; (R1 [] i; R)) ⟶ i; (R1 [] i; R)                  [m2, m2]
[23]  R [] (i; (P [> R)) ⟶ (i; (P [> R))                        [m2, e3]
[24]  R [] (P [> i; R) ⟶ (P [> i; R)                            [m2, e3]
  ⋮
```

The rules in the infinite sequence generated by the addition of associativity of [] follow the basic pattern:

```
R [] P ⟶ P                                                      (*)
```

Of course, a rule like (∗) is undesirable as it makes choice totally unfair (it will always choose the second argument). Also it is not an exact generalisation (i.e. not every instantiation is a rule) and so adding it would imply unsound equivalences.

There seems to be a more complex relationship between R and P which deserves further investigation. For example, R is a subterm of P, and P is built from applications of [>, i, and []. Such a study lies outside the scope of this thesis.

In [Wat92] a method of generalising infinite sequences of rewrite rules by recurrence terms is investigated. Recurrence terms extend the underlying term rewriting theory by allowing terms to be constructed using operators not in the signature; meta-operators in some sense. The basic extension allows a term schema to be defined, thus one recurrence term represents a sequence of the usual kind of term. The rule set generated above is used as an example to illustrate the application of the method, and an exact generalisation is produced. Since the solution relies on an extended form of term rewriting we cannot use it here (because RRL does not support that extension). The previous infinite sequence of rules is also used as an example in [Wat92], but we were able to find a different solution which does not require any special term rewriting techniques.

Associativity for [>

When rule e1 (associativity for [>) is added to the core set, with the operator [> declared to associate from right to left, the completion procedure diverges. The first 16 rules are as in figure 6.1; examples of the remainder are as follows.

```
[e1]  P [> (Q [> R) ⟶ (P [> Q) [> R
[18]  Q [] (Q [] exit) ⟶ (Q [] exit)                            [e5, e3]
[19]  (Q [] exit) [> R ⟶ exit [] (Q [> R))                      [e5, e1]
[20]  P [> (R [] exit) ⟶ (P [> exit) [> R)                      [e5, e1]
[21]  (Q [> R) [] ((P [> Q) [> R)) ⟶ (P [> Q) [> R              [e1, e3]
[22]  (P [> exit) [> exit ⟶ P [> exit                           [b4, 20]
[23] (P [> exit) [> (i; exit) ⟶ P [> i; exit                    [m2, 20]
[24]  ((P [> exit) [> P1) [> exit ⟶ (P [> P1) [> exit           [e3, 20]
[25]  (R [] exit) [] ((P [> exit) [> R) ⟶ (P [> exit) [> R      [20, e3]
[26]  exit [] (R [] exit) ⟶ R [] exit                           [b4, 19]
[27]  exit [] (i; exit [> R) ⟶ i; exit [> R                     [m2, 19]
[28]  exit [] ((P [> exit) [> R) ⟶ (P [> exit) [> R             [e3, 19]
  ⋮
```

This time generalisation is more difficult as there seem to be several infinite sequences of rules, each with a different pattern. Since no simple pattern is obvious, we try altering the composition of the rule set slightly by omitting law e5, which can be seen to be contributing to the critical pairs above, from *core*. Completion then gives the following rule set:

```
[b3]  P [] stop ⟶ P
[b4]  P [] P ⟶ P
[c3a] exit |[v]| exit ⟶ exit
[c3b] exit |[]| stop ⟶ stop
[d1]  stop >> P ⟶ stop
[d2a] exit >> P ⟶ i; P
[d3]  P >> (Q >> R) ⟶ (P >> Q) >> R
[e2]  P [> stop ⟶ P
[e4]  stop [> P ⟶ P
[e3]  Q [] (P [> Q) ⟶ P [> Q
[k1]  stop[v] ⟶ stop
[k2]  exit[v] ⟶ exit
[m1]  u; i; P ⟶ u; P
[m2]  P [] i; P ⟶ i; P
```

```
[m3]  u; Q [] u; (P [] i; Q) ⟶ u; (P [] i; Q)
[e1]  P [> (Q [> R) ⟶ (P [> Q) [> R
[17]  (Q [> R) [] ((P [> Q) [> R) ⟶ (P [> Q) [> R                    [e1, e3]
[18]  (i; Q) >> R ⟶ i; (Q >> R)                                     [d2a, d3]
[19]  P >> (i; R) ⟶ (P >> exit) >> R                                [d2a, d3]
[20]  (P >> stop) >> R ⟶ P >> stop                                  [d1, d3]
[21]  ((P >> exit) >> exit) >> P1 ⟶ (P >> exit) >> P1               [m1, 19]
[22]  ((Q [> Q1) [> R) [] (((P [> Q) [> Q1) [> R)
      ⟶ (((P [> Q) [> Q1) [> R)                                     [e1, 17]
[23]  (u; Q) [] u; (P [> (i; Q)) ⟶ u; (P [> (i; Q))                 [e3, m3]
[24]  (((Q [> Q1) [> Q2) [> R) [] ((((P [> Q) [> Q1) [> Q2) [> R)
      ⟶ ((((P [> Q) [> Q1) [> Q2) [> R)                             [e1, 22]
[25]  ((((Q [> Q1) [> Q2) [> Q3) [> R)
      [] (((((P [> Q) [> Q1) [> Q2) [> Q3) [> R)
      ⟶ (((((P [> Q) [> Q1) [> Q2) [> Q3) [> R)                     [e1, 24]
[26]  (((((Q [> Q1) [> Q2) [> Q3) [> Q4) [> R)
      [] ((((((P [> Q) [> Q1) [> Q2) [> Q3) [> Q4) [> R)
      ⟶ ((((((P [> Q) [> Q1) [> Q2) [> Q3) [> Q4) [> R)             [e1, 25]
  ⋮
```

This time the infinite sequence appears to have only one pattern.

arg [] (P [> *arg*) ⟶ P [> *arg*
ignoring brackets in (P [> arg) *since* [> *is associative*

where the form of *arg* can be described as a grammar.

arg	::=	$subterm_1$ [> R
$subterm_i$	::=	Q | $subterm_{i+1}$ [> Q

As before, the discovery of an exact generalisation of this rule set lies outside the scope of this work. However, this rule set has also been used as an example in [Wat92], where an exact generalisation is found.

Finally, we consider adding rules which express more of the functionality of the LOTOS relabelling operator (the *core* set has only two rules: one for **stop** and one for **exit**).

6.4.4 More Rules for the LOTOS Relabelling Operator

Since the addition of the expansion laws requires the addition of rules for some of the usual operations over lists and sets, it is possible to add more detailed versions of the LOTOS operators which were initially ignored, or only given a few base case rules because of their use of data types. The operator **hide** has its expansion law as mentioned above in section 6.4.1; what about the relabelling operator?

Although we can model the relabelling function as a list of pairs, some of the laws in the standard for relabelling have rather involved side conditions, e.g. the relabelling must be the identity function, or the relabelling must be injective. The identity condition can be easily checked by adding an auxiliary function which first collapses the list, i.e. replaces pairs of the form $[a/b, b/c]$ by the pair $[a/c]$. All that is then required is to check no pairs of the form $[a/b]$ appear, where a is different from b.

It is also possible to check the injectivity condition, but to do so would require so much extra work that it seems unlikely to be worth the effort. For example, we would require an auxiliary function which checks that the relabelling does not map two different gate names onto the same result. At this point we note that in LOTOS we have both gate parameters to processes and the relabelling operator, therefore a specifier will typically use gate parameters and ignore relabelling. In that case, our effort would be better spent in finding a way to implement gate parameters in the rewriting framework, than in adding complex rules for the relabelling operator.

The problem of the complicated side conditions of the laws of the relabelling operator is also discussed in chapter 8.

6.5 Summary

In this chapter we described the process of attempting to find a canonical rule set corresponding to the laws of weak bisimulation congruence for LOTOS as given in the LOTOS standard [ISO88]. We were able to find such a rule set for a portion of the LOTOS laws and used this rule set in performing simple rewrite proofs. It was clear from these examples that other, more powerful, rules were also required, i.e. the expansion laws. Although rules could be expressed implementing the expansion laws, these rules cause divergence of the completion procedure, as do a number of other desirable rules, such as associativity of [] . This means that the final rule set, which will be used in the next chapter, is not complete, and therefore some care has to be exercised in applying the rules. We have a semi-decision procedure for weak bisimulation congruence of LOTOS processes, i.e. normal forms are not unique. This means that if two terms can be shown to be congruent by our rules, then they are congruent in the LOTOS semantics, but if two terms cannot be shown to be congruent by our rules, then they may or may not be congruent in the semantics. No special techniques to cope with non-confluent rule sets are adopted other than resorting to a hand proof if two terms cannot be shown congruent by RRL.

Chapter 7

Using Term Rewriting for LOTOS: Login Case Study

7.1 Introduction

One of the main aims of this work is to investigate the verification requirements of LOTOS, i.e. to find out what sort of properties we *want* to verify, what sort of properties we *can* verify, and *how* the proofs may be carried out. In order to gain a better understanding of the problems of verification we undertake the study of the verification of the small communications problem presented in this chapter.

The verification requirement of the example studied is to show that some sort of relationship holds between the given specification and implementation, i.e. the implementation ***satisfies*** the specification. The problem is compounded as the implementation is not derived in any way from the specification. In the course of the verification we explore various ways of expressing the property to be proved and consider several approaches to the proof. We also try to automate the proofs required (by using the rules developed in the previous chapter). The aims of this chapter are:

- to investigate the verification requirements of LOTOS via an example,
- that the example chosen should display
  - some measure of realism, but also
  - simplicity (to allow easier exploration of the verification requirement),
- to obtain a successful proof that the verification requirement is met, and
- to demonstrate the viability of the rules developed in the last chapter as a proof technique.

The example is presented in section 7.2: an informal overview of the whole system is given, followed by formal and informal descriptions of the specification and implementation of the system. The formal descriptions are given in Basic LOTOS [ISO88]. Section 7.3 is concerned with a preliminary discussion of the interpretation of the verification requirement, and possible approaches to proving that it is satisfied. The formal details of the verification requirement are given in section 7.3.2. The process of automating the proofs is as was described in chapter 6.

Section 7.4 describes how we initially fail to meet the verification requirement. In fact, we can show that the implementation does not satisfy the specification. Close examination of the proofs that the relation does *not* hold results in a deeper understanding of the requirement and the development of a different approach to the proof. The new approach hinges on adding some extra information in a modular way to the specification; this is achieved by adopting the constraint oriented style of specification [VSvSB91] and allows the proof to be successfully completed. The new approach, the resulting specification, the statement of the verification requirement and its proof are presented in section 7.4.3.

We recognise that the example as it stands is simple, so possible extensions to the case study are discussed in section 7.5. In section 7.6 we review our experience with LOTOS and RRL, making suggestions for improvements. Finally, we give our conclusions and ideas for further work arising from this study.

The version of the case study presented here is essentially the same as the one presented at TAPSOFT'93 [Kir93], with some elaboration of various points drawn from the technical report [Kir92].

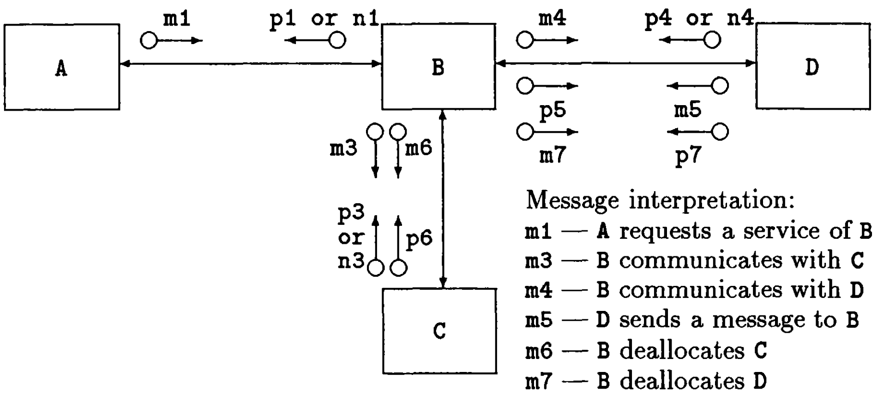
7.2 The Example

7.2.1 Informal Overview of the System

The example is an abstraction of a real communications problem involving four communicating processes at OSI Session level. It was first investigated as a case study for the "Verification Techniques for LOTOS" SERC project [VTL93].

There are four communicating entities: `A`, `B`, `C` and `D`, shown in figure 7.1. In the diagram, a box represents an entity, and a ○—► represents a message (sent in the direction of the arrow). Each message is labelled by `mx`, where `x` is a number in $\{1, 3, 4, 5, 6, 7\}$. Informal interpretations of the `mx` are given in figure 7.1. Messages of the form `px` or `nx` are positive and negative acknowledgements, respectively, to the corresponding `mx` messages.[1]

[1]Note that some messages only require a positive acknowledgment, while others require both positive and negative acknowledgments (see figure 7.1) — this is to do with the nature of the messages which they acknowledge, e.g. it does not make any sense to allow `C` to respond in a negative way to the message `m6` "deallocate".



Message interpretation:
m1 — A requests a service of B
m3 — B communicates with C
m4 — B communicates with D
m5 — D sends a message to B
m6 — B deallocates C
m7 — B deallocates D

Figure 7.1: The Processes and their messages

A requests a service from B; in order to provide that service, B must communicate with C and D. B has an internal timer which "times out" if D does not reply to its communication within a previously set time limit. B must send deallocation messages to C and D when they are no longer required.

The original example [Dic90] was supplied by Jeremy Dick, who worked for RACAL at the time. For reasons of security, we were given only the abstract description of the system as above; no indication of the real content or meaning of the messages was given. To help illuminate the system, we invented a possible interpretation of our own. This provides some intuition as to what happens in the system, although it is not an exact match. We view the system as follows: A is a user wishing to log-on to a system with a username and a password. C takes a username and checks that it is valid. D takes a valid username, acknowledges receipt of the name, and then returns the corresponding password. B co-ordinates these activities to ascertain if A is a valid user and has supplied the correct password. Since we use Basic LOTOS to model the example, the password and username are not in the formal description of the system.

The two descriptions of the system supplied to us are given below: firstly, a group of protocols which make up the *specification*, and secondly a group of processes which make up the *implementation*. Note that inconsistencies may be found between the way the specification describes something and the way the implementation describes the same thing. This is because the implementation was not derived from the specification and one of the problems considered here is that of trying to reconcile any differences between the two.

Below, the informal introductions to the specification and implementation are followed by their formal descriptions, given in Basic LOTOS. Note that in the remainder of this chapter, the term *processes* will be used to refer to the implementation part of the example.

In these descriptions the simplifying assumptions are made that the carrier is faithful and that no messages or acknowledgements are lost or corrupted.

7.2.2 Protocols

Communication in the system is governed by protocols P1, P2 and P3. Each protocol describes the interface between just two of the processes in the system, e.g. P1 describes the interface between A and B, completely ignoring C, D and their associated actions.

P1: A sends m1 to B, which must be acknowledged by p1 or n1.

P2: B sends m3 to C which must be acknowledged by p3 or n3. Following p3, B may or may not send m6 to C which must be acknowledged by p6.

P3: B sends m4 to D which must be acknowledged by p4 or n4. After p4, D may or may not send m5 to B. m5 must be acknowledged by p5. Also after p4, B may or may not send m7 to D. m7 is acknowledged by p7. If m7 is received it is no longer possible to send m5.

The LOTOS description of these protocols is as follows:

```
process P1 := m1; (n1; exit [] p1; exit) endproc
process P2 := m3; (n3; exit [] p3; (exit [] m6; p6; exit)) endproc
process P3 := exit [] m4; (   n4; exit
                           [] p4; (   exit [] m7; p7; exit
                                   [] m5; p5; (exit [] m7; p7; exit))) endproc
```

Note that the alphabets of the LOTOS processes P1, P2 and P3 are disjoint, i.e. the protocols are independent of each other.

In a real system the protocols, and also the processes, would probably be described recursively, i.e. cycling over the same behaviour forever. This is ignored at the moment, the simpler finite case being dealt with first. The problems of dealing with recursive processes in the rewriting paradigm are discussed in the next chapter. The initial **exit** branch of P3 is a result of the way the proof is carried out; the process names are given here as a convenience, but in the RRL system, the conjecture is entered using the full process expressions, therefore the **exit** branch models the case in which the P3 protocol is not activated.

7.2.3 Processes

The implementation of the system is achieved by four interacting processes.

A: A sends m1 to B. After this message B sends either p1 or n1 to A, indicating success or failure of the transaction respectively.

C: C receives m3 from B to which it replies either p3 or n3. If p3 is sent then C expects an m6 deallocation message, to which it replies p6.

D: D receives m4 from B, to which it replies p4, and the transaction continues, or n4, and the transaction terminates. After p4, D sends m5 to B, expecting p5 in response, then deallocation by m7, to which D replies p7. The transaction may be terminated if D receives m7 before it sends m5, i.e. the timer has expired causing B to terminate the transaction.

B: In a successful execution B receives m1 from A, allocates C with m3 p3 and D with m4 p4, then sets a timer as D must send m5 within some time limit. When m5 arrives the timer is cancelled and B replies with p5. C and D are deallocated by m6 p6 and m7 p7 respectively. Finally B signals the success of the transaction by sending p1 to A.

This sequence of actions may fail in a number of ways: either C or D could refuse to participate by returning negative acknowledgments (n3 or n4), or D might not send m5 within the time period, in which case the timer "times out". In these cases B replies n1 to A. Deallocation of C and D occurs if and only if they originally agreed to participate in the transaction, i.e. if p3 and p4, respectively, were sent.

The LOTOS descriptions of the processes are as follows:

```
process A := m1; (n1; exit [] p1; exit) endproc
process C := m3; (n3; exit [] p3; m6; p6; exit) endproc
process D := exit [] m4; (   n4; exit
                          [] p4; (   m5; p5; m7; p7; exit
                                  [] m7; p7; exit)) endproc

process B :=
  m1; m3; (   n3; n1; exit
           [] p3; m4; (   n4; m6; p6; n1; exit
                       [] p4; set; (   timeout; m6; p6; m7; p7; n1; exit
                                    [] m5; tcancel; p5; m6; p6; m7; p7; p1; exit)))
endproc
```

Note the differences and similarities between the descriptions of the protocols and the processes. For example, the description of A is identical to the description of P1, whereas the description of C differs slightly from P2 because in C deallocation is compulsory, whereas it is optional in P2. As with the protocols, the processes A, C and D are independent of each other; however, here we also have the process B which interacts with all other processes.

Now we have the formal descriptions of the specification and the implementation, we try to verify that the implementation is correct with respect to the specification.

7.3 Verification of the Example

7.3.1 Informal Discussion

The statement to be verified can be expressed as: does the implementation (the processes A, B, C and D) satisfy the specification (the protocols P1, P2 and P3)? The terms used here are deliberately vague, allowing exploration of different possible interpretations, discussed informally here and more formally in section 7.3.2. Three terms have yet to be defined: "specification", "implementation" and "satisfies".

In chapter 4 we assumed that the interpretation of the specification and the implementation was straightforward; however, consideration of this example shows us that this is not the case. For example, the protocols form the specification, but how they should be combined, or indeed *if* they should be combined, is not mentioned. The same is true of the processes and the implementation.

Suppose the protocols are to be combined to form the specification and the processes combined to form the implementation. The statement then becomes:

$$(\texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D}) \text{ satisfies } (\texttt{P1} \mid \texttt{P2} \mid \texttt{P3}) \tag{7.1}$$

where the "|" operator denotes "combined with". Note that each instance of "|" may be replaced by a slightly different operator when the statement is made concrete, i.e. the combinator used in `A | B` may be different from that used in `C | D`, or `P1 | P2`. This is formalised in section 7.3.2.

An alternative approach to expressing the verification requirement exploits the modular way in which the system has been defined: each facet of the interaction can be examined separately.

$$(\texttt{A} \mid \texttt{B}) \text{ satisfies } \texttt{P1} \tag{7.2}$$
$$(\texttt{C} \mid \texttt{B}) \text{ satisfies } \texttt{P2} \tag{7.3}$$
$$(\texttt{D} \mid \texttt{B}) \text{ satisfies } \texttt{P3} \tag{7.4}$$

As they stand, these equations are not quite correct since the language of the left-hand expression may not be the same as that of the right-hand expression, e.g. `A | B` will use events not mentioned in `P1`. Either these events will have to be hidden, or the interpretation of "satisfies" must take account of the extra events.

Since equations (7.2), (7.3) and (7.4) each yield a boolean, the results can be combined using a boolean operator. The correctness of the system as a whole is expressed by ((7.2) $\wedge$ (7.3) $\wedge$ (7.4)). We choose $\wedge$ since we want all facets of the interaction to be satisfied, but we must also be sure that satisfying all equations separately is the same as satisfying the system as a whole. In this case, since `P1`, `P2` and `P3` are all concerned with distinct facets of the communication of the system, it seems likely that the verification can safely be split into parts. Note that this really depends on choosing the right methods of splitting up the system, hiding unimportant events, making individual proofs, and recombining the results.

7.3.2 Formalising the Verification Requirement

We now give the formal interpretation of "|", the hiding of events, and "satisfies" in Basic LOTOS.

The general parallelism operator of LOTOS is used to combine both processes and protocols. Variations of the events in the synchronisation list give subtly different combinations of the components of the system as required above.

If we choose the modular approach to the verification requirement proof we must also formalise the means by which actions can be ignored in each of the equations (7.2), (7.3) and (7.4). The **hide** operator is used to restrict the processes to protocol events only.

There are many different possible interpretations in LOTOS for the "satisfies" relation. The various equivalence relations and preorders which can be used with LOTOS were discussed in detail in chapters 3 and 4. Here we re-examine these relations with specific reference to the current problem.

Given the use of the **hide** operator which converts hidden events into the internal event, an equivalence which ignores these internal events is required, therefore strong bisimulation cannot be used. At the other extreme we have trace equivalence which is too weak for verification purposes as deadlock properties are not preserved, leaving us with the weak bisimulation and testing relations. The system under examination will probably have to interact with other systems, so it is important that it behaves in the same way in all contexts. This leads us to reject weak bisimulation equivalence and testing equivalence, in favour of their congruent counterparts. We may also consider using the testing preorder **cred**. Any of these relations might be taken as our interpretation of "satisfies" in the verification requirement.

In the following section, we initially use the strongest relation available to us, weak bisimulation congruence, in place of "satisfies", progressing to weaker relations as necessary. If we start with the strongest relation and prove the equation holds for that relation, then all other (weaker) relations follow.

The next section contains details of the conjectures we attempted to prove hold, and some discussion of why many of those conjectures do not hold.

7.4 Verification Proofs

Two possible approaches to proving that the implementation of the system satisfies its specification have been presented above. One involves splitting the proof into three parts corresponding to the three protocols in the specification, while the other deals with the system as a whole. These two approaches are explored below.

7.4.1 Splitting the Conjecture into Three Parts

Since each protocol describes the interface between just two of the processes, the idea of proving each interface is correct and deducing from that the correctness of the whole system is very appealing. The equations to be proved in this section are all of the form:

$$\begin{aligned}
&\texttt{P1} \quad \text{satisfies} \quad \textbf{hide}\ [\texttt{m3,p3,n3,m4,p4,n4,m5,p5,m6,p6,m7,p7}]\ \textbf{in} \\
&\qquad\qquad (\texttt{A}\ |[\texttt{m1,p1,n1}]|\ \texttt{B}) && (7.5)\\
&\texttt{P2} \quad \text{satisfies} \quad \textbf{hide}\ [\texttt{m1,p1,n1,m4,p4,n4,m5,p5,m7,p7}]\ \textbf{in} \\
&\qquad\qquad (\texttt{C}\ |[\texttt{m3,p3,n3,m6,p6}]|\ \texttt{B}) && (7.6)\\
&\texttt{P3} \quad \text{satisfies} \quad \textbf{hide}\ [\texttt{m1,p1,n1,m3,p3,n3,m6,p6}]\ \textbf{in} \\
&\qquad\qquad (\texttt{D}\ |[\texttt{m4,p4,n4,m5,p5,m7,p7}]|\ \texttt{B}) && (7.7)
\end{aligned}$$

where `P1, P2, P3` and `A, B, C, D` are as defined in section 7.2.

The verification requirement is obtained by substituting different relations for "satisfies" in the above equations. Correctness of the system as a whole is proven when all three equations can be shown to hold for a particular relation. Note that the only parts of the conjectures which change from one proof to another is the relation substituted for "satisfies", and the orientation in the case of the **cred** refinement relation.

In the following proofs, we use *procAB, procCB* and *procDB* to denote the right hand sides of equations (7.5), (7.6) and (7.7) respectively. The rule sets *newcore*, *expand* and *hide* are used in RRL to attempt to show the conjectures hold for weak bisimulation congruence. We have no rule set for testing congruence or the **cred** relation, therefore these proofs are carried out entirely by hand.

Unfortunately, this approach turned out to be unsuccessful. Although some conjectures about the relationship between the specification and the implementation can be shown to hold, the results are not strong enough to satisfy the correctness requirement. Hand proofs of the negation of the conjectures which could not be shown to hold are given below. Examination of these conjectures and their proofs may help to illuminate the reasons for the failure of this approach overall.

Weak Bisimulation Congruence Consider:

$$\begin{aligned}
\texttt{P1} &\equiv_{wbc} \mathit{procAB} && (7.8)\\
\texttt{P2} &\equiv_{wbc} \mathit{procCB} && (7.9)\\
\texttt{P3} &\equiv_{wbc} \mathit{procDB} && (7.10)
\end{aligned}$$

These equations, (7.8), (7.9) and (7.10), cannot be proved to hold using the rule sets in RRL; however, we can use the reduced forms of the terms, which are normal forms only with respect to our rules, not necessarily unique normal forms, to help us in hand proofs of the negation of these conjectures. Remember that as our rule sets are not complete we cannot use RRL to prove inequalities. Also note that RRL cannot identify the cause of failure of proofs, it merely returns the normal forms of both sides of the conjecture.

By hand, we *can* prove the following:

$$\neg\,(\texttt{P1} \quad \equiv_{wbc} \quad procAB) \tag{7.11}$$
$$\neg\,(\texttt{P2} \quad \equiv_{wbc} \quad procCB) \tag{7.12}$$
$$\neg\,(\texttt{P3} \quad \equiv_{wbc} \quad procDB) \tag{7.13}$$

Proof. The key to the proofs of conjectures (7.11), (7.12) and (7.13) lies in the difference between weak bisimulation congruence and weak bisimulation equivalence. Informally, the difference between weak bisimulation equivalence and weak bisimulation congruence lies in the way in which initial internal actions are treated. Let P denote the left hand side of an equation and Q the right hand side, as in definitions 2 and 4 of chapter 3. In both the equivalence and the congruence, every action performed by P must be matched by an action performed by Q, where extra internal actions may be added in the matching process. Similarly, every action performed by Q must be matched by an action performed by P. However, for weak bisimulation congruence, an initial internal action must be matched by *one* or more internal actions, while weak bisimulation equivalence allows an initial internal action to be matched by *zero* or more internal actions.

The actions which can be performed by the various components of the system can be more easily seen by examining normal forms with respect to $\equiv_{wbc}$ produced by our rewrite rules on the terms in equations (7.11), (7.12) and (7.13). These are given in figure 7.2. Note that the protocols have not been reduced; they were already in normal form with respect to our rule sets. The normal forms in figure 7.2 will be referred to in the proofs in the remainder of this section.

Informally stated, the proof proceeds as follows for each inequation:

Equation (7.11) Clause (1) of definition (4) holds; *procAB'* can match every action `P1'` can perform. However, clause (2) fails because (*procAB'* **after** `m1`) can perform an **i** action to get into a situation in which it can perform the action `n1` and no other, whereas (`P1'` **after** `m1`) can only match an **i** action by doing nothing, which leaves it in a state in which either `n1` or `p1` are possible. Obviously a state in which two distinct actions are possible cannot be equivalent to one in which only one action is possible.

Equation (7.12) Again clause (1) is satisfied, but clause (2) fails as an initial internal action must be matched by one or more internal actions. `P2'` cannot match the initial internal action performed by *procCB'* except by zero actions, and therefore the expressions are not weak bisimulation congruent.

Equation (7.13) Exactly the same as for equation (7.12).

This ends the proof that the inequations (7.11), (7.12) and (7.13) hold. ∎

`P1` **normal form (`P1`$'$) is**
```
        m1; ((n1; exit) [] (p1; exit))
```

`P2` **normal form (`P2`$'$) is**
```
        m3; ((n3; exit) [] (p3; (exit [] (m6; p6; exit))))
```

`P3` **normal form (`P3`$'$) is**
```
          exit
        [] m4; (   n4; exit
                [] p4; (   exit
                        [] m7; p7; exit
                        [] m5; p5; (exit [] m7; p7; exit)))
```

procAB **normal form (*procAB*$'$) is**
```
        m1; ((i; n1; exit) [] (i; p1; exit))
```

procCB **normal form (*procCB*$'$) is**
```
        i; m3; (n3; exit [] p3; m6; p6; exit)
```

procDB **normal form (*procDB*$'$) is**
```
        i; (   i; exit
           [] i; m4; (   n4; exit
                     [] p4; (   i; m7; p7; exit
                             [] m5; p5; m7; p7; exit)))
```

Figure 7.2: Normal Forms of the Processes and the Protocols with respect to the Weak Bisimulation Congruence rule sets *expand* and *newcore* plus *hide*.

Since we failed to show the conjectures of the verification requirement hold for weak bisimulation congruence, showing instead that the verification requirement is not met for the specification and implementation with respect to weak bisimulation congruence, we move to a weaker relation, testing congruence.

Testing Relations Taking equations (7.5), (7.6) and (7.7) as above, we substitute the **cred** relation for "satisfies" and try to show the new equations hold left-to-right and right-to-left (giving testing congruence), i.e.

$$\texttt{P1}\ \ \mathbf{cred}\ \ procAB \tag{7.14}$$
$$\texttt{P2}\ \ \mathbf{cred}\ \ procCB \tag{7.15}$$
$$\texttt{P3}\ \ \mathbf{cred}\ \ procDB \tag{7.16}$$

and vice versa.

Equations (7.14) and (7.15), in the left to right direction, can be shown to hold by applying the **cred** laws of the LOTOS standard, which means that the protocols are a deterministic reduction of the processes, i.e. the processes may have some nondeterminism not present in the protocols.

However, we normally expect the implementation of a system to be *less* nondeterministic than the specification. Neither of these equations can be shown to hold by application of the **cred** laws in the right-to-left direction; equation (7.16) cannot be shown to hold in either direction.

Proofs of the corresponding inequations are slightly more tricky than the proofs of the weak bisimulation congruence inequations. We begin by showing that the equations (7.14), (7.15) and (7.16) do not hold in the right-to-left direction for the **cred** relation, i.e.

$$\neg\,(procAB \quad \textbf{cred} \quad \texttt{P1}) \qquad (7.17)$$
$$\neg\,(procCB \quad \textbf{cred} \quad \texttt{P2}) \qquad (7.18)$$
$$\neg\,(procDB \quad \textbf{cred} \quad \texttt{P3}) \qquad (7.19)$$

Proof. In definition 9 of chapter 3, the **cred** relation was characterised by the tests a process must pass. To prove that two LOTOS processes are not related by **cred**, a test must be found which one must pass but the other need not. In particular, because the orientation of the expression is important, a test must be found which the right hand side must pass, but which the left hand side may fail. For example, in equation (7.17) a test must be found which `P1` must pass, but which *procAB* may fail. This is sufficient to prove that the **cred** relation does not hold in the given direction. If no such test can be found, there may be a context which differentiates the terms, since **cred** is a congruence relation.

In the following proofs we refer back to the normal forms given for weak bisimulation congruence in figure 7.2 and use these in the proofs. This procedure is sound.

Theorem 1 *To make use of the normal forms we need to show, given* $p \equiv_{wbc} p'$ *and* $q \equiv_{wbc} q'$,

$$\neg\,(p' \ \textbf{cred} \ \ q') \Rightarrow \ \neg\,(p \ \textbf{cred} \ \ q)$$

where p' *and* q' *denote the normal forms with respect to our rule sets relating to weak bisimulation congruence of* p *and* q *respectively.*

Proof. The method of proof is by contradiction. The following statements hold: $p \equiv_{wbc} p'$ and $q \equiv_{wbc} q'$, and $\neg\,(p'$ **cred** $q')$, which we will demonstrate for each case in the remainder of this section. Now suppose p **cred** q. Since $(B_1 \equiv_{wbc} B_2) \Rightarrow (B_1$ **cred** $B_2)$, from [ISO88], we may deduce p' **cred** p and q **cred** q', and hence p' **cred** q' by transitivity of **cred**, thus contradicting our initial statements. ∎

Although there are **cred** laws which can reduce some of the normal forms given further, they are not used here as they apply only to the process normal forms and reducing terms on the left hand side of the **cred** relation when we are trying to show that it doesn't hold could lead to false conclusions (this problem is discussed in more detail in section 8.4).

Again informally stated, the proofs that equations (7.17), (7.18) and (7.19) hold are as follows:

Equation (7.17) (P1′ **after** m1) can always pass the test n1, but (*procAB′* **after** m1) will sometimes fail that test (because the internal actions can proceed silently and put us into a state in which only p1 can go ahead).

Equation (7.18) There is no test which can differentiate between P2′ and *procCB′*; however, they can be differentiated by the context P [] m8; **exit**, where m8 is an arbitrarily chosen action and P is a place holder for the system under test. In this context, ((P2′ [] m8; **exit**) **after** ϵ) can always pass the test m8, whereas ((*procCB′* [] m8; **exit**) **after** ϵ) could fail this test. This is because the internal action can move *procCB′* [] m8; **exit** into a state where only m3 is possible.

Equation (7.19) To prove this equation we must find a test which P3′ must satisfy, but which cannot be passed by *procDB′*. Given the test m5, (P3′ **after** m4 p4) must always pass this test, but (*procDB′* **after** m4 p4) won't necessarily pass this test (as the internal action may be taken, putting *procDB′* into a state in which it can only perform m7).

This concludes the proofs that equations (7.17), (7.18) and (7.19) hold. ∎

The final inequation to be proved is to show that (7.16) does not hold, i.e.

$$\neg\,(\texttt{P3}\quad \textbf{cred}\quad procDB) \tag{7.20}$$

Proof. The proof proceeds by demonstrating that there is a test which *procDB* must satisfy but which may not be satisfied by P3. Consider the state (*procDB′* **after** m4 p4 m5 p5 δ), call it DB'. Although DB' **must** m7, (P3′ **after** m4 p4 m5 p5 δ) may sometimes fail the test m7, therefore P3′ is not a refinement of *procDB′*.

This concludes the proof that equation (7.20) holds. ∎

Equations (7.5), (7.6) and (7.7) do hold for trace equivalence, but as we said earlier, this relation is really too weak to be useful in verification.

At this point it appears that trying to prove the verification requirement is satisfied is hopeless. However, we strongly believe that the processes are a valid implementation of the system. Since we tried weakening our interpretation of satisfies with no success we must conclude that it is the approach to the proof which is incorrect. The strategy of splitting the conjecture into three parts does not work, or rather, proofs of some parts of the conjecture can be completed, but these are not sufficient to satisfy the verification requirement. By examining the normal forms given in figure 7.2, it seems that the hiding of events causes the failure of the proofs by spotlighting apparently nondeterministic choices in the process normal forms. These choices are not really nondeterministic; they are determined by factors in the other processes. For example, *procAB′*

makes a nondeterministic choice between replying `p1` and replying `n1`. However, we know that this choice really depends on the receipt of `m5` (which is hidden). This problem affects proofs using weak bisimulation congruence or testing congruence. We observe that we are not the only ones to encounter this problem; the same phenomenon also causes problems for other authors, e.g. [Bai91, BA91].

We now go on to try the other approach to the proof, where the system is considered as a whole, thus avoiding the use of **hide**.

7.4.2 Proving the System as a Whole

No relationships between the processes all combined and the protocols all combined can be demonstrated because, although the processes can be combined using parallelism, there is no meaningful way in which to combine the protocols.

Two operators are possible candidates for combining the protocols: sequential composition of process expressions (the "enable" operator, $\gg$) and interleaving (general parallelism synchronising on no events, since the protocols have no events in common). Using sequential composition to combine the protocols we obtain the following expression:

```
P1 >> P2 >> P3
```

This is an unsuccessful way of combining the protocols because, for example, the events of `P1` do not all precede the events of `P2`. Interleaving gives:

```
P1 ||| P2 ||| P3
```

which is similarly unsuccessful as the protocols contain no information about the relative ordering of events in different protocols. Interleaving results in a process expression which has a large number of traces which make no sense given our informal understanding of the system. For example, one trace which results from the above expression is ⟨`p4 p3 m4 m3 m1`⟩; however, we know from the description in section 7.2 that `m1` should be the first event in the interaction, and that events occur in numerical order, i.e. `m3` comes before `m4`, and that messages occur before their respective acknowledgements.

Given this way of expressing the protocols we can only prove *protocols* **cred** *processes*, i.e. the protocols implement the processes, which is not a true reflection of the verification requirement. The reason we cannot show *processes* **cred** *protocols* is that the protocols also specify lots of other behaviours which the processes do not. Really our problem is that the specification is too weak; there are some details which have been omitted.

The missing information, which is implicit in the implementation, includes details of a timer, deallocation and what constitutes success or failure of the transaction. In the specification there is no information about any of these things. Our solution is to add the information in the form of *constraints*, giving a successful approach to the problem.

7.4.3 Adding Constraints to the Example

In the constraint oriented style of specification, described in [VSvSB91] and also earlier in section 3.5.2 of this thesis, different aspects of the behaviour of the system are described by separate processes, the full system description being given by the parallel composition of these subparts. The effect is similar to using conjunction in a logical specification; each part must be satisfied for the whole to be satisfied. This style of specification is only possible because of the multi-way synchronisation of the LOTOS parallel operators.

Using this specification style, we define more LOTOS processes which express aspects of the specification not included in the protocols. These include a timer in B to determine how long it should wait for D to send the m5 message, compulsory deallocation of C and D, ordering of events as mentioned in the informal overview of the system, and conditions dictating success or failure of the transaction as a whole. The following constraints are added to the specification:

```
Timer Constraints
process timer     := exit [] set; ( tcancel; exit [] timeout; exit) endproc
process timer_on  := exit [] p4; set; exit endproc
process timer_off := exit [] set; (   m5; tcancel; p5; m7; exit
                                   [] timeout; m7; exit) endproc

Deallocation Constraints
process dealloc_C := p3; m6; p6; exit [] n3; exit endproc
process dealloc_D := exit [] m4; (p4; m7; p7; exit [] n4; exit) endproc

Success and Failure
process system    :=    m5; p1; exit
                     [] n3; n1; exit
                     [] n4; n1; exit
                     [] timeout; n1; exit endproc

Ordering Constraints
process order13  := m1; m3; (   n3; n1; exit
                             [] p3; (n1; exit [] p1; exit)) endproc

process order34  := m3; (   n3; n1; exit
                         [] p3; m4; (   n4; n1; exit
                                     [] p4; (n1; exit [] p1; exit))) endproc

process order457 :=    n3; n1; exit
                    [] m4; (   n4; n1; exit
                            [] p4; (   m5; p5; m7; p7; p1; exit
                                    [] timeout; m7; p7; n1; exit)) endproc

process order56  :=    n3; n1; exit
                    [] n4; m6; p6; n1; exit
                    [] timeout; m6; p6; n1; exit
                    [] m5; p5; m6; p6; p1; exit endproc
```

```
process order67  :=     n3; n1; exit
                      [] p3; (   n4; m6; p6; n1; exit
                                 [] p4; m6; p6; m7; p7; (   n1; exit
                                                           [] p1; exit)) endproc
```

As with the descriptions of the protocols and the processes, some **exit** branches are introduced to express the notion that a constraint may not be activated.

Given these constraints, the correctness of the system may now be expressed by the following equation:

```
        (((P1 |[p1, n1]| system) |[m1, p1, n1, n3]| order13)
            |[p1, n1, m3, p3, n3, n4, m5, timeout]|
        ((((P2 |[p3, n3, m6, p6]| dealloc_C)
            |[m3, p3, n3, m6, p6]| (order34 |[p3, n3, p4, n4]| order67))
            |[p1, n1, n3, m4, p4, n4, m7, p7]|
        (((P3 |[m4, p4, n4, m7, p7]| dealloc_D)
            |[m4, p4, n4, m5, p5, m7, p7]| order457)
            |[p4, m5, p5, m7, timeout]|
        ((timer |[set]| timer_on) |[set, timeout, tcancel]| timer_off)))
            |[m5, p5, m6, p6, timeout]| order56))
```

$\equiv_{wbc}$

```
        (((A |[m1, p1, n1]| B) |[m3, p3, n3, m6, p6]| C)
            |[m4, p4, n4, m5, p5, m7, p7]| D)
```

Note that, although the order in which the process expressions are combined does not affect the meaning of the process expression as long as the synchronisation lists are adjusted accordingly; we find that in practice it is helpful to add the processes which restrict behaviour before adding the ones which add behaviour. In the equation above, this means adding as many constraints as possible to each protocol before combining it with the other protocols. The reason that this is necessary is that, in performing the reductions, our system of rule sets can only deal with one parallel statement at a time, which means that the proof has to be built up gradually from small units. This is a feature of the way the rules to expand parallel expressions have been implemented and was mentioned in section 6.4.1. Adding as much information as early as possible helps to cut down the size of the intermediate stages in the proof.

The above equation can be proved to hold by the rule sets. This is an adequate proof of correctness since it means not only that the processes have the same observable behaviour as the protocols, but also that they behave in the same way in all contexts. The proof requires only the *expand* rules. As the specification and implementation contain no internal actions we may also deduce that the above equation holds for strong equivalence as well as for weak bisimulation congruence. This is to be expected as the final process expressions are deterministic (and therefore all equivalences are the same).

Although we have achieved our aim of proving the verification requirement is satisfied, it is at some cost; we had to alter the specification and the new one is much more complex.

7.5 Extensions to the Example

The example as considered so far is very simple; there are a number of ways in which it can be made more complex.

- A useful extension would be to add an "abort" message, call it `m2`. `A` can abort the service at any time by sending `m2` to `B`, which should clean up by deallocating any resources held and then replying to `A` with p2.

  In LOTOS it would be simple to add `m2` p2 as an abort sequence using the operator [> , which allows one process to take control from another. However, in this example the system is more complicated, requiring varying sequences of actions between `m2` and p2, depending on the events which occurred before `m2`. The original solution could not be easily extended to include this new behaviour. This could indicate a fault in the solution to the original problem: perhaps it is not modular enough, or it could be that there is no simple, elegant way to extend the solution. Certainly it is true to say that some of the constraints used in the final description and proof of the example are not perhaps the most obvious descriptions. In particular, some constraints contain too much information, in that they are not as modular as their names suggest. This information was necessary to be able to combine the LOTOS processes in a meaningful way. We do not claim that our solution is in any way optimal; further investigation may reveal better solutions which can be easily extended when the specification of the problem is altered. Another possibility is that it is the form of this particular modification which is causing the problem, see section 7.6.1 for further discussion.

- Data types could be added to the messages, e.g. the login name and password of the informal interpretation of the example.

- The most obvious extension would be to introduce recursion.

The first two extensions are not considered further here; the case study with recursion is considered in the next chapter.

7.6 Review of the Tools Used

Although some degree of success is achieved in the case study, there are also many problems, not all of which arise from the example itself; some are due to either LOTOS or RRL.

7.6.1 Improvements to LOTOS

LOTOS is not always suitable to describe the example. A major problem is revealed when attempting to extend the original problem to include the abort message. The [> operator is unsuitable for this purpose because it does not allow the abort sequence to be dependent on the state of execution before the abort message. One way round this is to write each abort possibility into the LOTOS processes as choice branches, which makes the specification rather cumbersome. What is required is an operator which allows the abort sequence to be flexible, perhaps allowing parameters to be passed from the interrupted to the interrupting process (there is an extension of this sort for *sequential* composition of processes in full LOTOS).

Another feature which would be useful is an operator to "wrap-up" several actions and make them behave as a single action, i.e. like a critical section in a mutual exclusion problem. For example, we want to be able to combine two process expressions using interleaving, but to have a section in one of the expressions which, once it has begun, has to finish without interleaving with the other process until after the last action is completed. This could be achieved by using a mutual exclusion algorithm, but a language construct to do this would be more convenient. This problem is also identified in [Got87].

7.6.2 Improvements to RRL

The following is a list of features which we would find useful in carrying out our proofs.

- The ability to specify which rule to apply next, particularly important when the rule set is not complete.
- A list of all the rules used in a reduction. RRL does not supply this information when a reduction is carried out.
- The ability to split rules into groups *within* the system and to specify which groups of rules can be used in a reduction. (We achieve this effect by running simultaneous copies of RRL, each with a different rule set. This requires the user to cut and paste equations from one system to another, which provides the opportunity to introduce errors.)
- The ability to reference the last term reduced and to use that reference in the next reduction. As our rule sets are not complete, the order in which rules are applied can be very important. For the case study proofs this means that subparts of the proof have to be fully expanded before they can be added to the rest of the expression. Again we use cut and paste to perform this function, increasing the possibility of error.
- The ability to save rule sets, rather than having to regenerate them every time.
- Although RRL has fairly sophisticated functionality, the user interface is poor, being only a list of commands followed by a prompt. A graphical interface would make the system more appealing to use.

Perhaps these features will be available in later versions of the software, but we solve some of the problems by adopting a different tool. This is described in the next chapter.

7.7 Summary and Discussion

After much experimentation, we have successfully shown that the verification requirement of a small communications protocol are indeed satisfied. It must be noted that the given specification was not sufficient for our purposes and had to have more information added to enable the proofs to be carried out. The new information was added in a modular way however, and the text of the original specification was unaltered, although it must be admitted that the size and complexity of the specification was greatly increased. Possible extensions to the problem, including the introduction of recursion to the LOTOS processes, were provided in section 7.5, but not explored. It is hoped that these can also be made in a modular way.

In some ways, the initial failure to meet the verification requirements was perhaps more fruitful than the final proof, because we were able to identify problems in the verification process which need to be further researched. For example, the effect of **hide** on our proofs, introducing nondeterminism and thus causing failure, and the difficulty of choosing which of the many equivalence relations of the process algebra literature to use.

Another way to look at this problem is that perhaps we chose the wrong approach to the verification. Essentially the protocols constitute a partial specification of the system, which is why we had to hide events when splitting the conjecture into three parts. However, using **hide** and equivalence relations we could not prove that the implementation satisfied the specification. We also tried using the preorder **cred** which captures some aspects of the notion of partial specification, but not the ones pertinent to this example because we were unable to show the implementation satisfied the specification for **cred** either. The failure of **cred** was due to the fact that **cred** only captures the notion of reduction with respect to nondeterministic choice, whereas we require the reduction of deterministic choice, e.g. we need something which says $a; A \; sat \; a; A [] \, b; B$ where $a \neq b$.

A third possible solution would be to express the protocols, the specification, in terms of logic and to show the processes provide a model in which that logical formula holds. This possibility is explored further in chapter 11.

The main result of our work on this case study is the demonstration that verification, even of such a small and simple system, is a difficult process, one which is full of opportunities to take the wrong decision and thereby to fail to prove the correctness of the system under investigation. This is true even though we restrict our attention to a particular formulation of verification, namely equivalence between two processes. In this study we only arrived at a successful conjecture and proof because we persevered, having a strong belief that the verification requirement could be formulated in this way. In more complex examples it would perhaps be less easy to hold such a belief and this prompts several worrying thoughts: how long must we persevere to gain an acceptable formulation of the verification requirements and proofs that they are satisfied, how do we measure acceptability, and how do we know when to give up. The answers to these questions can only be gained through more experience of the verification process.

Chapter 8

Using PAM to Implement LOTOS Relations

In the previous chapter we presented a case study in verifying that a LOTOS specification was satisfied by its implementation, also given in LOTOS. There we were more concerned with different ways of expressing the verification requirement, so the system was simplified to ease the proof process. Specifically, the processes were not recursive. At the time we pointed out that this was somewhat unrealistic and that a future exercise should be to introduce recursion to the processes and show that the verification requirement was still satisfied. We also encountered various problems with the use of RRL as a proof tool. Although when a complete rule set is used RRL is very useful and can apply all the rules automatically until a normal form is reached, we wish to have more control over the rules applied since our rule sets are not complete. This is something which RRL cannot supply. This aspect, together with the recursion problem, encouraged us to seek a new proof tool more suited to our purpose. The new tool, PAM [Lin92], which is again an equational reasoning tool, is described in this chapter. Having a different tool gives a slightly different approach to the proof process, therefore we re-examine our choice of rewrite rules, or rather, our choice of the underlying laws. The different possibilities for weak bisimulation congruence are explored in section 8.3, with laws for the other equivalences also considered here, but not in as great detail. In section 8.4 we consider the problem of axiomatising the **cred** preorder for input to PAM.

8.1 Proof: Technique and Automation

In this section we consider the problem of extending our rewriting system to *recursive* LOTOS processes; this implies adding recursive rules. Below we discuss several possible approaches to this problem, using the simple buffer, Buffer := in; out; Buffer, as an illustrative example.

The obvious (naive) approach adds recursive process definitions directly as recursive rules, i.e.

$$\text{Buffer} \longrightarrow in;\ out;\ \text{Buffer}$$

This is unsatisfactory because the rule set becomes non-terminating; since this rule can always be applied, the process of rewriting can never end.

To ensure termination, we could instead add the rule

$$in;\ out;\ \text{Buffer} \longrightarrow \text{Buffer}$$

This rule is no good for investigating the behaviour of the buffer as it folds all the observable behaviour away, leaving only the process name itself, making it impossible to say anything about the behaviour of the system.

The next possible approach is to somehow control the use of the first rule in the rewriting by modelling the recursion as *primitive recursion*, rather than allowing unbounded recursion. This can be done by, for example, adding a counter to indicate the number of times the rule may be applied, thus ensuring the rule can only be applied a finite number of times; this approach is used in [CN92]. Although this approach may work sufficiently well for small examples, where the number of times the recursive equation has to be unfolded is easily calculated, it is probably unworkable in practice. However, consideration of this technique gives a clue to the final approach.

The naive approach of adding recursive rules does not work because it is the term rewriting system which decides which rule to apply next, and it only stops when no more rules can be applied, therefore rewriting with recursive rules is non-terminating. By using primitive recursion we try to *control* the unfolding and application of the recursive rules, but the method of control is too restrictive, forcing us to decide in advance how many times the rule can be applied. If we cannot automate the decision of when to unfold the expression, then we must give the user interactive control over the unfolding of the recursive equations. We must therefore abandon traditional rewriting tools (in which virtually no control over the application of rules is given to the user), turning instead to a tool which gives the necessary control, preferably without totally abandoning equational reasoning as a proof technique. PAM (the Process Algebra Manipulator) [Lin92] is such a tool and is described in the next section.

8.2 PAM

PAM [Lin92] is a parameterised rewriting-based proof assistant designed with process algebras in mind. Rather than implementing the operators and equivalences of a particular process algebra, like tools such as the Concurrency Workbench [CPS89], the philosophy behind the development of PAM was to create a general tool which could be used for any process algebra. PAM has some built-in transformation steps, relating to common steps in equational proofs which are not language specific. An example of one of these built-in steps is the *unique fixed-point induction* technique (ufi); this is of particular interest to us as it allows reasoning about the equality of recursively defined processes. Other functions are available which manipulate the presentation of the proof in the proof window (zoom, outline) and which perform routine operations (substitution, definition, folding/unfolding).

To operate, PAM requires the user to supply a definition of the language to be used, Basic LOTOS in our case. Although the description of a number of process algebras, including CCS and CSP, is given in the PAM manual, PAM has not, to our knowledge, been used for LOTOS before. The language definition contains arities and precedences of the operators of the language and axioms which describe the behaviour of those operators. Once set up, PAM can be used to carry out equational reasoning style proofs on terms of the language.

An important feature of PAM is that it has a *graphical interface*. The proof window consists of two parts: a control panel and a proof display. The control panel has lots of buttons; one for each of the built-in functions, and one for each of the user defined axioms. The proof display is essentially the paper on which the conjecture and the subsequent transformations are written. To apply an axiom, the user must select a term in the proof side, and then click on a button in the control side. Since the language definition gives *equations* describing the behaviour of the operators, the direction of application must be specified by the user every time that equation is applied. A switch in the control panel specifies either left-to-right or right-to-left; this direction then applies for all equations and substitutions. The user has complete control over the proof process; indeed, PAM cannot perform proofs automatically, instead the system is an automated pencil and paper; aiding in the book-keeping of proofs.

At the heart of each proof is a conjecture to be proved. This is supplied, together with any process definitions required, in a separate file by the user. To help structure the proofs, they are divided into sections; so if a subterm of the main conjecture has to be reduced it can be done in a new section, the result of that reduction being substituted for the original subterm in the main section.

As well as giving the freedom to use equations individually, the current version of PAM (v1.0) also gives power in the form of equation groupings. A simple language allows tactics to be built which describe common patterns of equation application, thus freeing the user from the tedium of applying each equation individually. Tactics are built by single or multiple applications of equations (the number of applications can be limited). Direction of application can also be specified, effectively specifying a rule rather than an equation. A tactic can then be used to specify a rule set. If the rule set so defined is confluent and terminating, then application of that tactic to a term will produce the (unique) normal form of that term, thus giving us an automatic proof technique. PAM cannot determine whether or not a rule set is complete; this must be done using one of the traditional rewriting tools such as RRL.

Tactics are problem specific, i.e. the tactic descriptions go into the problem definition files rather than the process algebra description file. Useful tactics, developed for use with LOTOS, may be found in the PAM problem definition files of appendix D.

Using tactics, PAM can offer flexibility for the experienced user, and a nice interface to an automatic proof process for the novice, i.e. the user need not make any hard decisions about the order of application of the equations, or the direction of application. This approach can also cut down some of the more laborious proofs by automating portions of them. Another useful feature for the novice is the command "ask". Given a term, "ask" will tell the user which equations are currently applicable.

A full description of the PAM system may be found in [Lin92]. We now present some of the steps in customising PAM for use with LOTOS.

8.2.1 Setting up PAM

Creating a proof in PAM requires two steps: description of the process algebra in the language definition file, and description of the particular conjecture to be proved in the problem definition file. Below we give details from the actual input files for PAM for LOTOS and weak bisimulation congruence to illustrate the method of definition. In these files, comments are prefixed by --.

The first part of the language definition is to declare the types of the language, and the operators and their arities. Subtypes can also be declared. PAM has two built in types: boolean and set. Infix operators are defined by including _ in the operator declaration as a place holder. For any operator, PAM also allows the user to declare priority (a number between 0 and 9999, where a higher priority (higher number) means tighter binding), associativity and commutativity, and the direction of association (left or right).

The LOTOS definition begins with the following declarations:

```
signature
type Gate Action Process
with Gate < Action
operator
    _ [] _      ::  Process Process -> Process 120 AC RIGHT
    _ ._        ::  Action Process -> Process 200 RIGHT
    stop        ::  -> Process
    exit        ::  -> Process
    hide _ on _ ::  Gate set Process -> Process 300
    i           ::  -> Action
    delta       ::  -> Action
```

```
_ |[ _ ]| _ :: Process Gate set Process -> Process 150 AC LEFT
_ ||| _     :: Process Process -> Process 150 AC LEFT
```

The syntax of the operators is close to the usual LOTOS syntax; departures have only been made where clashes with PAM's inbuilt operators occur, e.g. **hide _ in _** becomes **hide _ on _** as PAM uses **in** for set membership, and the action sequencing operator ; becomes **.** (as in CCS).

The next part of the definition file is concerned with the axioms which define the operators declared above. Note that, although for weak bisimulation congruence these are really *laws* with respect to the model of labelled transition systems, to PAM they are *axioms*, as PAM knows nothing about the underlying model. In the remainder of the document, we will refer to "PAM axioms" in an attempt to make the distinction clear.

Below we give an example of some of the PAM axioms for weak bisimulation congruence, to illustrate the style of definition. The makeup of the PAM axioms is more fully discussed in section 8.3 and the full PAM input files may be found in appendix D.

```
axiom
    -- B1 and B2 AC laws redundant; choice declared AC above
    B3    x [] stop = x
    B4    x [] x = x

    H1    hide A on stop = stop
    H2    hide A on (x [] y) = (hide A on x) [] (hide A on y)
    H3A   hide A on a. x = a. (hide A on x) if not(a in A)
    H3B   hide A on a. x = i. (hide A on x) if (a in A)
```

The above example illustrates the usual form for simple axioms, such as the choice axioms, and more complex conditional axioms, such as the **hide** axioms. The code on the left is the name of the axiom; this appears on the button in the command panel, and may also be used in defining tactics, see below.

A special law of process algebras is the expansion law. Since the expansion law really denotes a infinite family of laws, it cannot be expressed in the same way as the laws above, therefore PAM provides a special template. For each language, only the particulars of how synchronisation is achieved need be supplied by the user. For LOTOS we must supply information about the *style* of communication, i.e. broadcast, and information about which actions may synchronise, and which actions may proceed independently. This information is used by the expansion law to determine the possible actions resulting from a parallel expression.

```
expansion law
EXP let x = a1. x1 [] ...  [] an. xn          y = b1. y1 [] ...  [] bm. ym then
    (x |[A]| y) = stop
                  if (sync_move(x,y) eq nil) and (async_move(x,y) eq nil)
    (x |[A]| y) = Sum([],async_move(x,y))
                  if sync_move(x,y) eq nil
    (x |[A]| y) = Sum([],sync_move(x,y))
                  if async_move(x,y) eq nil
    (x |[A]| y) = Sum([],async_move(x,y)) [] Sum([],sync_move(x,y))
                  otherwise

with communication function
broadcast
    sync(a, b) =  a    if (a eq b) and not(a eq i) and ((a in A) or (a eq delta))
    async(a)   =  true if not((a in A) or (a eq delta))
```

The functions `Sum`, `sync_move` and `async_move` are built in to PAM.

Some of the laws of LOTOS require information about the language of a process, denoted $\mathcal{L}(P)$. PAM has this function built in, and calls it `Sort`; the process of calculating the sort is called *sort computation*. Note that this procedure is only decidable when calculating the *syntactic sort*, i.e. PAM does not carry out any analysis of whether or not an action is ever possible in the given environment.

Sort computation may or may not be necessary for a particular example; a flag must be set in the problem definition file to enable it. The function `Sort` is defined by the user by a set of equations, each relating to a particular language construct:

```
sort computation
    Sort(stop) = {}
    Sort(i. P) = Sort(P)
    Sort(a. P) = a union Sort(P)
    Sort(P [] Q) = Sort(P) union Sort(Q)
    Sort(P |[A]| Q) = Sort(P) union Sort(Q)
    Sort(hide A on P) = Sort(P) diff A
```

Note that PAM does not require braces round a in the definition of `Sort(a. P)`.

This completes the description of the form of the language definition file.

Each proof in PAM requires a language definition file, as described above, and also a problem definition file. The problem definition file contains the conjecture to be proved, which may be an equation or inequation, and any auxiliary process definitions required by the conjecture. This file may also contain tactics, see below, and the sort computation flag. Several example of problem definition files may be found in appendix D.

Using the defined axioms, tactics may be defined which help to partially automate the proof process. For example, below we define a tactic which will repeatedly apply the axioms of **hide** until no more are applicable, thus pushing the occurrence of **hide** as far into the process as possible.

```
rule HIDE = *{H1 H2 H3A H3B}
```

Here the tactic name is `HIDE` and the equations it refers to are just the **hide** axioms defined earlier. The braces group the axioms together, indicating that any one of the rules may be applied (alternatively, they could have been separated by semi-colons, indicating that they should be tried one after the other). The asterisk indicates that the process should be repeated until no more rules are applicable (or until the rewrite limit defined in the .pam file, default 20, is reached). We could also have specified the direction of application for any of the equations, e.g. `H1>` for left to right and `H1<` for right to left. More examples of tactics may be found in the problem definition files in appendix D.

PAM also allows macros to be defined which help make the process definition more readable. These are typically used to define sets of events.

Finally, although most of the operators of LOTOS were implemented in PAM easily, the relabelling operator continues to cause problems.

8.2.2 Adding the LOTOS Relabelling Operator

In section 6.4.4 we noted that adding rules for the LOTOS relabelling operator is not trivial, mainly due to the complex side conditions of some of the relabelling laws of the standard, i.e. requiring the relabelling function to be the identity function, or to be injective. In PAM, the side conditions of an axiom may only be simple boolean expressions, therefore we cannot even express the necessary conditions for the relabelling laws in PAM. However, although the difficulty of expressing these laws in RRL led us to reject the relabelling operator, we are reluctant to do so this time.

Since PAM does not allow processes to have parameters, one important use of relabelling in PAM is to model the effect of gate parameters and their instantiation. In order to have this facility we implement a restricted version of relabelling in which only one gate is renamed in one application of the operator, e.g. $P[a/b]$. This form of implementation means that, for individual axioms, the side conditions become trivial. However, several relabellings can be applied to one process, and there is no way to add conditions which deal with more than one application of an operator at a time. This means that it is up to the user to ensure that the function modelled by all the relabelling applications is the identity function, or is injective, depending on the axioms used. PAM cannot enforce these conditions.

The PAM axioms for relabelling are given in appendix D.2, and examples of the use of relabelling to implement gate parameters may be found in the examples in the next chapter; specifically, the readers and writers problem, section 9.3, and the candy machine, section 9.4.

Having presented the form of the language and problem definition files, we spend the rest of this chapter considering the choice of laws/axioms for the language definition file.

8.3 PAM Axioms for LOTOS Equivalence Relations

In moving from RRL to PAM, power has been gained, in that recursive processes may now be used in specifications, and that the user has a lot of control over equation application. Unfortunately, a great deal of convenience has also been lost, in that PAM is not automated, and the user must direct every equation application. The ability to specify tactics compensates a little for this loss by providing partial automation. In addition to these changes, there is one other of significance: whereas RRL allows rules to be applied in one direction only, PAM deals with equations and allows them to be applied in either direction, therefore we are no longer concerned with the completeness of the rules with respect to the laws, except possibly when defining tactics. Instead we are concerned with the completeness of the PAM axioms with respect to the underlying model of equivalence. We aim for a more complete representation of the LOTOS language and the LOTOS relations considered so far, namely: branching bisimulation equivalence, weak bisimulation congruence, testing congruence and trace equivalence. We know that, for the latter three relations, it is not possible to have a complete and finite axiomatisation, as discussed in section 4.2.2. However, we can attempt to find a set of axioms whose form follows some pattern and which are simple to apply; a set which is appealing in some sense. Since the aim for completion in chapter 6 led to the rejection of several rules, and therefore their underlying laws, the decision to forego completion allows us to reconsider our choice of laws. Below we consider the laws of a number of relations, weak bisimulation congruence in particular, since this is the relation most often used in verification proofs.

8.3.1 Laws Given in [BIN92]

In the LOTOS standard it is stated that the laws given for weak bisimulation congruence are not complete (with respect to the model) and that other laws could be used to express the relation. In [BIN92], laws are given which translate *finite* Basic LOTOS into finite CCS (i.e. expressions involving only choice and sequencing). Once in CCS, there exist complete (with respect to to the model, not necessarily confluent and terminating) sets of laws for several of the standard relations. The ones given are: observation congruence (weak bisimulation congruence), branching bisimulation equivalence, testing congruence and trace equivalence. These laws are given in PAM form in appendix D.1. We remark that the laws of branching bisimulation equivalence, when added to the laws of strong congruence (the choice laws), form a confluent and terminating set of rewrite rules; this was proved in [AB90] for axioms expressed using the left merge operator, but we also used RRL to form a complete rule set corresponding to the laws as given in appendix D.1. None of the other equivalences described in appendix D.1 admit a complete rule set; attempting completion with RRL on rule sets corresponding to these laws (using all possible permutations of precedences offered by RRL) results in divergence of the completion procedure.

Note that for implementation in PAM these rules have been augmented by three other laws which are derived from the LOTOS standard to ease the proof process. For example, the rule DELTA which states that **exit** is the same as δ; **stop**. This is added because of the way in which the PAM expansion law works (it does not allow **exit** as an argument to the parallel operator as all processes/process branches must have at least one initial action). The others also state specifically the behaviour of parallelism with **exit** or **stop** as arguments.

The laws of [BIN92] are useful in situations in which a definition of a process must be unfolded, pushing occurrences of the higher level operators, such as **hide**, $\gg$ and [> , further into the expression so that the initial portion of the process uses only [] and sequencing. This sort of rewriting is required, for example, when trying to identify occurrences of particular events, or patterns of behaviour. However, although complete with respect to the model of finite Basic LOTOS, these laws do not yield a confluent and terminating rule set, and therefore must not be used indiscriminately. In particular, since recursive equations can be continuously unfolded, these laws form a nonterminating set of rules. This is not a great problem, as the user is unlikely to keep unfolding a recursive definition. The advantages of these laws, that their form follows some sort of simple pattern and that they are easy to apply, outweigh this possible disadvantage.

8.3.2 Extra Laws Taken from in [ISO88]

A further disadvantage of the laws of [BIN92], which cannot be discounted, is that these laws are geared only towards removing occurrences of the higher level operators to give a more operational definition of the system; they cannot manipulate occurrences of these operators in any other way. For example, given the expression **hide** A **in hide** A **in** P, we would expect to be able to reduce it to **hide** A **in** P, or, if $A \cap \mathcal{L}(P) = \emptyset$, P. The laws of [BIN92] cannot perform these reductions because they are intended to push occurrences of high-level operators through a process expression until an occurrence of **exit** or **stop** is reached, at which point the high-level operator can be removed. The laws of [BIN92] are complete only with respect to *finite* LOTOS, but our proofs typically involve infinite processes. The above strategy fails on infinite processes as **exit** and **stop** are seldom encountered. We found during the course of the experiments detailed in chapter 9 that such reductions involving high-level operators are often required, typically to allow duplicate states in a process unfolding to be recognised as such. These laws usually take the form of those of the LOTOS standard, or slight variants thereof. To this end, we have prepared a supplementary set of PAM axioms which follow the standard and which may be added to the set from [BIN92] as required.

It is important to remember that the PAM axioms/laws given in appendix D are only guidelines. For any particular proof, it is possible that another rule, derived from the inference rules defining

LOTOS, may be required. One of the benefits of using PAM is that several different language definition files may be loaded at the same time, so it would be possible to maintain files in which different equivalences were described, or in which the same equivalence is described in different ways. As long as the syntax of the operators remains constant, problem definitions should be able to be used in any setting (unless defined tactics rely on a particular set of axioms). Of course, all possible laws/axioms could be included in one file, but this makes the file and the proof window somewhat unwieldy.

8.4 PAM Axioms for cred

The above section deals only with expressing axioms for the usual equivalence and congruence relations defined over LOTOS. However, it was noted in chapter 2 that equivalence and congruence relations are not the only relations of interest when comparing LOTOS specifications. In some situations, e.g. if the specifications are *partial*, i.e. early specifications ignore some aspects of the system, while later specifications provide a more full and accurate representation of the system, it may be inappropriate to use an equivalence relation to compare specifications. In these cases, where we wish to show that the implementation *approximates* the specification, preorders (sometimes also referred to as implementation relations) can be useful. For LOTOS, there are two important related preorders: **red** and **cred**. These preorders are based on those first presented in [DH84] and are used in the definition of testing equivalence and congruence. Some motivation for these preorders may be found in section 3.4.3; the LOTOS definitions of **red** and **cred** appear in section 3.5.3. Other preorders defined there, originally presented in [BSS87], are closely related, differing only in minor details.

We have chosen to examine the **cred** preorder for two reasons: mainly because it appears in the standard (although, as mentioned above, the preorders are all substantially the same), but also because **cred** is the congruent counterpart of **red** (and congruence is important if we consider that any system will typically be part of a larger system). Figure 8.1 gives the laws from [ISO88] for **cred**.

The problem we consider in this section is how a set of PAM axioms corresponding to these laws may be formed, allowing us to decide if two processes are related by the **cred** preorder (but note that **cred** is not decidable, therefore the best we can hope for is a semi-decision procedure).

Since **cred** is a preorder and therefore reflexive and transitive but not symmetric, the implementation of **cred** in PAM cannot be achieved in the same way as the implementation of the equivalences as described in the previous section. For example, consider the second law of

1. $\vdash B_1 \ \equiv_{wbc} \ B_2 \Longrightarrow B_1$ **cred** B_2

   Note: this means that **cred** inherits all the laws for weak bisimulation congruence.

2. B **cred** $\mathbf{i}$; B

3. $g; (B_1 \ [] \ B_2)$ **cred** $g; B_1 \ [] \ g; B_2$

4. $g; B_1$ **cred** $g; B_1 \ [] \ g; B_2$

5. B_1 **cred** B_2 & B_2 **cred** $B_3 \Longrightarrow B_1$ **cred** B_3

6. B_1 **cred** B_3 & B_2 **cred** $B_3 \Longrightarrow (B_1 \ [] \ B_2)$ **cred** B_3

Figure 8.1: The **cred** laws from [ISO88]

figure 8.1. Our first attempt at expressing this as a PAM axiom might be:

$$X \ = \ \mathbf{i}; X$$

where = stands for **cred**. However, the = relation is PAM in assumed to be an equivalence, therefore adding **cred** in this way is not sound. Although X **cred** $\mathbf{i}; X$ holds, $\mathbf{i}; X$ **cred** X does not.

The obvious way to get round this problem is to axiomatise **cred** as a predicate; then the equivalence we deal with in PAM will be the equivalence over truth values. For example, the law above will be added as:

$$(X \ \textbf{cred} \ \mathbf{i}; X) \ = \ \textit{true}$$

This approach to adding PAM axioms for **cred** is discussed in section 8.4.1. Unfortunately, in order to ensure the correctness of the PAM axioms formed using this approach, some strong conditions concerning the syntactic form of terms have to be made; these make this approach too restrictive in practice.

Given that axiomatising **cred** as a predicate is not really practical, we return to the possibility of axiomatising **cred** as if it were an equivalence in section 8.4.2. Although this is generally unsound we can find some restrictions on the axioms which guarantee soundness. The proof system generated by this approach is more powerful and flexible than the one in which **cred** is viewed purely as a predicate. Section 8.4.4 contains the theoretical work required to back up this claim. We emphasise here that this rather bizarre approach to axiomatising **cred** is necessary because we cannot express **cred** properly in the equational reasoning framework. At this stage, rather than adopting a new framework, we prefer to see how far we can push the equational reasoning paradigm.

8.4.1 Axiomatising cred as a Predicate

As **cred** is not an equivalence relation, we cannot treat it in the same way as the other LOTOS relations when considering how to add PAM axioms to allow proofs of the conjecture

implementation **cred** *specification*

The obvious way to deal with **cred** is to axiomatise it as a *predicate*. We now face another problem: to our knowledge, no such axiomatisation exists in the literature. Below we present our attempt to axiomatise **cred** as a predicate.

Given that **cred** has been declared appropriately in PAM, the first axioms we need are those expressing simple cases for which **cred** holds, i.e. base case axioms. Using the laws in figure 8.1 as a guide, we arrive at the following:

```
CBASE2 (A cred i.A)                    = true
CBASE3 (a.(B1 [] B2) cred a.B1 [] a.B2) = true
CBASE4 (a.B1 cred a.B1 [] a.B2)        = true
```

These are obviously correct as they are merely particular instantiations of the corresponding laws in figure 8.1. We now consider PAM axioms corresponding to the other laws of figure 8.1.

There is no PAM axiom relating to the first **cred** law of figure 8.1. Assuming the definition file for **cred** also includes the axioms for weak bisimulation congruence, this law says that we can use these axioms to reduce A and/or B in the conjecture A **cred** $B = true$ and the procedure is sound. This is implicit in the operation of PAM which allows subterms to be reduced separately, and the result to be substituted back into the main conjecture.

The fifth **cred** law of figure 8.1, transitivity of **cred**, also does not appear because it is impossible to express this in the PAM framework. This law would require being able to work on two conjectures at once: A **cred** $B = true$ and B **cred** $C = true$, and on being able to combine the results. Attempting to express a PAM axiom which would allow this gives:

$$(A \textbf{ cred } B \;=\; \textit{true})\ \textit{and}\ (B \textbf{ cred } C \;=\; \textit{true}) \;=\; (A \textbf{ cred } C \;=\; \textit{true})$$

but here we end up using the = operator four times in one axiom, which is not allowed by PAM. The effect of this law could be gained by the user decomposing and recombining the desired conjecture manually, doing separate proofs for each part of the conjecture, but this is not something we can add explicitly as a PAM axiom. We ignore the sixth **cred** law of figure 8.1 for similar reasons.

In addition to the PAM axioms above, we add the following:

```
CBASE7 (B cred B) = true
```

since we must define **cred** to be reflexive. This law is implicit in law 1 of figure 8.1.

Note that none of the base case laws above allow A **cred** $B = \mathit{false}$ to be derived, i.e. the proof system is at best a semi-decision procedure for **cred**. This is to be expected as **cred** is not decidable. If $\neg(A$ **cred** $B)$ then the proof must be completed by hand. Obviously we prefer a fully automated procedure; however, this inability to show $\neg(A$ **cred** $B)$ in PAM can be used to our advantage, see section 8.4.2 below.

The next sort of PAM axiom we consider is, given an arbitrary expression A **cred** $B = \mathit{true}$, where A and B do not fit any of the base case axioms, how do we reduce the expression until a base case axiom *does* apply?

The motivation for the form of these axioms is that the validity of **cred** depends on the tests passed, i.e. $(A$ **cred** $B)$ implies if B passes a test, then so does A (but not vice versa), and therefore if $\neg(A$ **cred** $B)$ there will exist at least one test which B passes but which A does not. The strategy we attempt to express in the PAM axioms is that we can reduce both A and B by removing process expressions which behave in the same way, i.e. pass the same test, retaining the parts of A and B which differ (if there are such parts). Eventually we reach an expression to which a base case axiom applies, implying $(A$ **cred** $B)$. If no base case axiom applies then it is possible that $\neg(A$ **cred** $B)$, and the proof is continued by hand. We hope that the reductions will have made the distinguishing behaviour of A and B more obvious and that a hand proof will be simplified. The axioms are as follows:

```
CRED2A  i.A cred i.B                = A cred B
CRED3A  a.A [] a.B cred a.A [] a.C  = a.(A [] B) cred a.(A [] C)
CRED4A  a.A [] a.B cred a.A [] a.C  = a.B cred a.C
```

These axioms suffer two problems; the first is that they are not sound, the second problem is that in the above form the axioms are only applicable to a small number of expressions, i.e. we have lost a lot of the power of the original laws. We consider the soundness problem first.

The problem with this approach to formulating PAM axioms for **cred** is that we cannot guarantee that removing a part of the process expression is the same as removing a particular set of tests. For example, in `CRED4A` the idea behind the form of the axiom is that branches with identical behaviour are removed, leaving behind the branches which may (or may not) have different behaviour. In this way, we hope to finally reduce the expression until either it is proved true by the base case axioms, or the user can prove by hand that the relationship does not hold. However, in `CRED4A` it is possible that $B = A$, in which case removing the $a.A$ branch will not remove all the tests which $a.A$ satisfies from the expression. This could mean that these tests have been removed from one side of the **cred** predicate, but not the other, i.e. we have artificially *created* a distinguishing test.

To correct this problem we must ensure that removing process behaviour is the same as removing tests. The problem can be formalised as follows: let L_1 be the largest set for which

$a.A$ **must** L_1 and let L_2 be the largest set for which $a.B$ **must** L_2. In order to apply CRED4A maintaining soundness we must ensure that all tests L_1 are removed by removing the expression $a.A$. The simplest way to do this is to insist that $L_1 \cap L_2 = \emptyset$. To ensure this condition holds we place strong constraints on the syntactic form of the expression A, B and C: we can insist that both sides be normal forms with respect to weak bisimulation congruence (which solves the example above), and also that the language of B be disjoint from the language of A (thus assuring that the set of tests relating to each branch do not overlap). This is stronger than necessary; we only require $L_1 \cap L_2 = \emptyset$, but this cannot be expressed in PAM. In general, it is also simpler to compute the language of a process than the set of tests it passes.

These constraints ensure soundness of the PAM axioms for **cred**. Leaving aside for a moment the fact that they are extremely restrictive, we move on to consider the second problem with our formulation of the **cred** preorder axioms: the loss of the power of the original laws.

Consider the following example: we might want to show

$$(a.A \,[]\, C \textbf{ cred } a.A \,[]\, a.B \,[]\, C) \;= \mathit{true}$$

but the relevant axiom (CRED4A) applies only to expressions with two choice branches, not three. Moreover, CRED4A must reduce both sides of the expression, whereas the conjecture above needs a PAM axiom which reduces only the right hand side of the **cred** predicate.

Obviously we can add axioms of the correct form for choice, but then we also have to consider the other operators of LOTOS and add axioms for them. Another example of a relation which can be proved to hold by the laws of the standard, but which cannot be shown to hold by the PAM axioms above is

$$a.b.B \textbf{ cred } a.(b.B \,[]\, b.C)$$

This highlights the problem: the PAM axioms can only be applied to the outermost level of an expression.

Normally in this situation we use the PAM functions to reduce the subterms of the expression (otherwise we would have had the same problem in axiomatising the equivalence relations), but since **cred** is axiomatised as a predicate and not as $=$ we have lost this ability. To overcome this problem we would have to add an infinite number of axioms, each dealing with different operators and different levels of nesting. Clearly this is not possible.

Our inability to express **cred** as a predicate, without either losing the power of the original laws, or having to place strong syntactic conditions on the axioms, indicates that we cannot accurately express **cred** in the equational reasoning framework. On the other hand, our proof system is considerably weakened if we ignore preorder relations. This encourages us to try to find an alternative formulation of the PAM axioms for **cred**.

8.4.2 Axiomatising cred as an Equivalence

Although originally rejected as a means of implementing **cred**, in this section we explore more fully the consequences of axiomatising **cred** as if it were an equivalence relation.

Some of the problems of the previous attempt to implement axioms for **cred** in PAM were generated because the built in functions of PAM cannot be fully utilised given the form of the axioms. In particular, subterms of expressions cannot be reduced independently, with the result being substituted back into the original conjecture. This sort of manipulation applies only if the relation being considered is modelled by the = of PAM, i.e. an equivalence relation. In order to regain this ability, we consider what happens if **cred** *is* axiomatised as an equivalence. The axioms are as given in figure 8.2. We also declare **cred** as a predicate and add the base case axioms, `CBASE2`, `CBASE3`, `CBASE4` and `CBASE7`, as in the previous section, allowing the derivation of *true*. The PAM input file for these definitions is given in appendix D.3.

```
CRED2   A            = i.A
CRED3   a.(B [] C) = a.B [] a.C
CRED4   a.C          = a.B [] a.C
```

Figure 8.2: PAM axioms for **cred** as an equivalence

We know that in general this method of axiomatising **cred** is not sound, but we can place some constraints on the application of the axioms which allow only sound reductions. Unfortunately, PAM cannot enforce these restrictions, therefore we rely on the restraint of the user. To help, we retain some of the features of the previous axiomatisation in order to remind the user that **cred** is not an equivalence and that care should be taken in the application of these axioms. This is why we add the base case axioms, `CBASE2`, `CBASE3`, `CBASE4` and `CBASE7`, although they are subsumed by the axioms of figure 8.2. We also declare **cred** as a predicate, even though it is also expressed in the axioms by =. This allows us to continue expressing conjectures involving **cred** as A **cred** $B\ = true$, which serves as a constant reminder in the proof of the true nature of **cred**.

Note that not all of the original laws are explicitly included in this axiomatisation; the first and fifth laws are implied by the properties of = and PAM's built-in substitution facility. The sixth law of figure 8.1 is again ignored, and for the same reason; we cannot express this law in the framework of PAM. Also note that axioms `CRED3` and `CRED4` hold for $a \in G \cup \{\mathtt{i}\}$, rather than $g \in G$ as in the original laws. The proofs that this is the case are in section 8.4.3.

Given the axioms of figure 8.2, in which cases is their application sound? Only if they are applied right to left to the expression on the right hand side of the **cred** predicate, i.e. given the conjecture A **cred** $B = true$, we may only reduce B. The proof of this is given in section 8.4.4, where we consider the implications of using the axioms of figure 8.2 as rewrite rules. If A is reduced in A **cred** $B = true$ then the reduction may not be sound.

Like the previous axiomatisation, this axiomatisation is obviously incomplete, as we do not provide a way to derive *false*. We also note that random applications of the axioms may lead to being unable to prove relations which hold. For example, CRED3 may be used to throw away the "wrong" branch of an expression, i.e. given

$$a.b.A \textbf{ cred } a.(b.A \ [] \ b.C) \ = \ true$$

we can reduce the right hand side of the **cred** predicate to $a.b.C$. However, whereas the original expression holds, $a.b.A$ **cred** $a.b.C \ = \ true$ does not. In such cases it is an advantage that we have no PAM axioms $(A \textbf{ cred } B) = false$, as otherwise we might not realise our mistake. For this example, we should have used the axiom to reduce the right hand side to $a.b.A$, then CBASE7 could be applied to show the conjecture holds.

This new approach is obviously more powerful than that of the previous section; the examples which were given there as impossible to derive in that system are straightforward to prove with the axioms of figure 8.2. Equally obviously, the axioms of this section are not as powerful as the original **cred** laws, as they may not be applied to the left hand side of the **cred** predicate.

An example of the use of these **cred** axioms is included in section 9.2.3. To some extent, the rules were designed with this example in mind, and are therefore tailored to make this particular proof work. Whether or not they can be applied to other examples remains to be seen.

The remaining sections of this chapter provide the theoretical basis for the axiomatisation of **cred** as in figure 8.2 and the investigation of the soundness of the use of those axioms as rewrite rules.

8.4.3 Proving that Axioms CRED3 and CRED4 hold for i

In axioms CRED3 and CRED4 of figure 8.2 we have used a for actions, thus including **i**, whereas the corresponding laws of the standard use g, implying that they do not hold for **i**. This is not the case and may be an oversight due to the inclusion of the law which allows all occurrences of **i** to be removed. In this section we show that the third and fourth **cred** laws *do* hold for **i**. These proofs were completed by hand.

Theorem 2 *The following hold:*

$$\mathbf{i}; B_1 \quad \textbf{cred} \quad \mathbf{i}; B_1 \ [] \ \mathbf{i}; B_2 \tag{8.1}$$

$$\mathbf{i}; (B_1 \ [] \ B_2) \quad \textbf{cred} \quad \mathbf{i}; B_1 \ [] \ \mathbf{i}; B_2 \tag{8.2}$$

The proofs that these laws hold proceed in two stages: first we prove the laws hold for **red**, and then we consider the validity of the relation in all contexts. In the second part of the proofs only the context of choice need be considered, as this is the only context which affects the substitution property (see for example [Mil89b] or [DH84]).

Proof. In order to show that law (8.1) holds, we must first show that it holds for **red**, i.e.

$$\forall t.\forall L.((\mathbf{i}; B_1 \; [] \; \mathbf{i}; B_2) \; \mathbf{after} \; t) \; \mathbf{must} \; L \Rightarrow ((\mathbf{i}; B_1) \; \mathbf{after} \; t) \; \mathbf{must} \; L \qquad (\clubsuit)$$

In other words, in all states, if $\mathbf{i}; B_1 \; [] \; \mathbf{i}; B_2$ can pass a test then so must $\mathbf{i}; B_1$.

We consider the state sets which result after various traces. The table in figure 8.3 lays out the sets P **after** t for left and right hand sides of the axiom for each possible case of t. In this figure, we use $tr(P)$ to denote the trace set of P, and, given a trace s, such that $s \in tr(P)$, we use P' to denote the set of states reached after s, i.e. $\{P'' \mid P \xrightarrow{s} P''\}$.

trace, t	state set of $\mathbf{i}; B_1$ after t	state set of $\mathbf{i}; B_1 \; [] \; \mathbf{i}; B_2$ after t
$t = \epsilon$	$\{\mathbf{i}; B_1, B_1\}$	$\{\mathbf{i}; B_1 [] \; \mathbf{i}; B_2, B_1, B_2\}$
$t \in tr(B_1) \cap tr(B_2)$	B_1'	$B_1' \cup B_2'$
$t \in tr(B_1), t \notin tr(B_2)$	B_1'	B_1'
$t \notin tr(B_1), t \in tr(B_2)$	$\{\}$	B_2'
$t \notin tr(B_1) \cup tr(B_2)$	$\{\}$	$\{\}$

Figure 8.3: State sets of law 8.1 after selected traces

The next step is to evaluate the effects of various tests on those state sets.

For each state set we consider four possible tests: the test belongs to the trace set of B_1 and the trace set of B_2, the test belongs to trace set of B_1 only, the test belongs to the trace set of B_2 only, and the test does not belong to the trace set of either B_1 or B_2. These results of these tests applied to the state sets of figure 8.3 are presented in figures 8.4 to 8.8 in a form similar to that of truth tables. In these tables, lhs denotes the process expression $\mathbf{i}$; B_1 and rhs denotes the process expression $\mathbf{i}$; $B_1 \; [] \; \mathbf{i}$; B_2.

We need not consider $L = \{\}$, as then all tests fail, regardless of the state set. We also do not have to explicitly consider instantiations of L such that L has more than one element. The expression P **must** $\{a, b\}$ is equivalent to the expression P **must** $\{a\} \vee P$ **must** $\{b\}$. Since $\vee$ is used to combine the results, it is enough to show that the implication ($\clubsuit$) holds for singleton sets.

These tables show that for every value of a, and hence all possible L, and for all traces, the implication ($\clubsuit$) holds, i.e.

$$\mathbf{i}; B_1 \; \mathbf{red} \; \mathbf{i}; B_1 \; [] \; \mathbf{i}; B_2$$

$L = \{a\}$	state set of rhs $\Rightarrow$ state set of lhs
$a \in tr(B_1) \cap\ tr(B_2)$	true $\Rightarrow$ true
$a \in tr(B_1) \wedge a \notin tr(B_2)$	false $\Rightarrow$ true
$a \notin tr(B_1) \wedge a \in tr(B_2)$	false $\Rightarrow$ false
$a \notin tr(B_1) \cup\ tr(B_2)$	false $\Rightarrow$ false

Figure 8.4: $t = \epsilon$.

$L = \{a\}$	state set of rhs $\Rightarrow$ state set of lhs
$a \in tr(B_1') \cap\ tr(B_2')$	true $\Rightarrow$ true
$a \in tr(B_1') \wedge a \notin tr(B_2')$	false $\Rightarrow$ true
$a \notin tr(B_1') \wedge a \in tr(B_2')$	false $\Rightarrow$ false
$a \notin tr(B_1') \cup\ tr(B_2')$	false $\Rightarrow$ false

Figure 8.5: $t \in tr(B_1) \cap\ tr(B_2)$.

$L = \{a\}$	state set of rhs $\Rightarrow$ state set of lhs
$a \in tr(B_1') \cap\ tr(B_2')$	true $\Rightarrow$ true
$a \in tr(B_1') \wedge a \notin tr(B_2')$	true $\Rightarrow$ true
$a \notin tr(B_1') \wedge a \in tr(B_2')$	false $\Rightarrow$ false
$a \notin tr(B_1') \cup\ tr(B_2')$	false $\Rightarrow$ false

Figure 8.6: $t \in tr(B_1), t \notin tr(B_2)$.

$L = \{a\}$	state set of rhs $\Rightarrow$ state set of lhs
$a \in tr(B_1') \cap\ tr(B_2')$	true $\Rightarrow$ true
$a \in tr(B_1') \wedge a \notin tr(B_2')$	false $\Rightarrow$ true
$a \notin tr(B_1') \wedge a \in tr(B_2')$	true $\Rightarrow$ true
$a \notin tr(B_1') \cup\ tr(B_2')$	false $\Rightarrow$ true

Figure 8.7: $t \notin tr(B_1), t \in tr(B_2)$.

$L = \{a\}$	state set of rhs $\Rightarrow$ state set of lhs
$a \in tr(B_1') \cap\ tr(B_2')$	true $\Rightarrow$ true
$a \in tr(B_1') \wedge a \notin tr(B_2')$	true $\Rightarrow$ true
$a \notin tr(B_1') \wedge a \in tr(B_2')$	true $\Rightarrow$ true
$a \notin tr(B_1') \cup\ tr(B_2')$	true $\Rightarrow$ true

Figure 8.8: $t \notin tr(B_1) \cup\ tr(B_2)$.

We must now show that the above statement holds when **cred** is substituted for **red**. This is done by considering the expressions $\mathbf{i}; B_1$ and $\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2$ in a choice context, i.e. we need to show

$$Z \;[]\; \mathbf{i}; B_1 \;\mathbf{red}\; Z \;[]\; \mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2$$

for some process expression Z. We only need to consider the state set resulting from $t = \epsilon$, i.e. the case in which the $\mathbf{i}$ action might pre-empt the choice; the other state sets are similar to those in figure 8.3. After the trace ϵ, the state sets of the left and right hand sides of the above statement are:

$$\{Z \;[]\; \mathbf{i}; B_1, B_1\} \qquad\qquad \{Z \;[]\; \mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2, B_1, B_2\}$$

Similarly, only one test need be considered, $a \in tr(Z) \wedge a \notin (tr(B_1 \cup tr(B_2))$. In this case, both state sets above fail the test and therefore the implication holds.

We therefore conclude that law 8.1 is valid. ∎

Proof. In order to show that law (8.2) holds, we must show

$$\forall t.\forall L.((\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2) \;\mathbf{after}\; t) \;\mathbf{must}\; L \Rightarrow (\mathbf{i}; (B_1 \;[]\; B_2) \;\mathbf{after}\; t) \;\mathbf{must}\; L \qquad (\spadesuit)$$

In other words, in all states, if $(\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2)$ can pass a test then so must $(\mathbf{i}; (B_1 \;[]\; B_2))$.

As above, we consider the state sets which result after various traces; these are given in figure 8.9. The state sets are identical for all traces except $t = \epsilon$, therefore they will pass the same tests and we need only consider tests for state sets of $t = \epsilon$.

trace, t	state set of $\mathbf{i}; (B_1 \;[]\; B_2)$ after t	state set of $\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2$ after t
$t = \epsilon$	$\{\mathbf{i}; (B_1 \;[]\; B_2), B_1 \;[]\; B_2\}$	$\{\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2, B_1, B_2\}$
$t \in tr(B_1) \cap tr(B_2)$	$B_1' \cup B_2'$	$B_1' \cup B_2'$
$t \in tr(B_1), t \notin tr(B_2)$	B_1'	B_1'
$t \notin tr(B_1), t \in tr(B_2)$	B_2'	B_2'
$t \notin tr(B_1) \cup tr(B_2)$	$\{\}$	$\{\}$

Figure 8.9: State sets of law 8.2 after selected traces

Assume $L = \{a\}$. Consider the possible cases of a and its membership of $tr(B_1)$ and $tr(B_2)$. If a belongs to both, then both sides of the law pass the test. If a belongs to one but not the other, then $\mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2$ fails the test, whereas $\mathbf{i}; (B_1 \;[]\; B_2)$ passes the test. Finally, if a belongs to the trace set of neither branch, then both sides fail the test.

We conclude that the implication $(\spadesuit)$ holds for each test, i.e.

$$\mathbf{i}; (B_1 \;[]\; B_2) \;\mathbf{red}\; \mathbf{i}; B_1 \;[]\; \mathbf{i}; B_2$$

We move on to consider how placing these expressions in a choice context might alter the outcome of the tests.

The statement we now need to consider is

$$Z \mathbin{[]} \mathbf{i}; (B_1 \mathbin{[]} B_2) \ \mathbf{red}\ Z \mathbin{[]} \mathbf{i}; B_1 \mathbin{[]} \mathbf{i}; B_2$$

for some process expression Z. Again only the case $t = \epsilon$ need be considered. The state sets are:

$$\{Z \mathbin{[]} \mathbf{i}; (B_1 \mathbin{[]} B_2), B_1 \mathbin{[]} B_2\} \qquad \{Z \mathbin{[]} \mathbf{i}; B_1 \mathbin{[]} \mathbf{i}; B_2, B_1, B_2\}$$

Similarly, only one test need be considered, $a \in tr(Z) \wedge a \notin (tr(B_1) \cup tr(B_2))$. In this case, both state sets above fail the test and therefore the implication holds.

We therefore conclude that law 8.2 is valid. ∎

We note that for each of these laws, the proof that the law holds in a choice context was not really necessary: since each side of the expression is prefixed by **i**, substituting one for the other in a context has no effect, i.e. the tests passed are not changed. We use this realisation to justify restricting the analysis of the PAM **cred** axioms in the next section to visible actions only; this makes the presentation simpler. The results are easily extended to include invisible actions.

8.4.4 Why cred as an Equivalence can be Dangerous

In this section we analyse the consequences of axiomatising **cred** as an equivalence and using the PAM axioms given in figure 8.2 as rewrite rules. Axiomatising **cred** in this way may lead to unsound reductions as PAM assumes the relation is an equivalence, i.e. reflexive, symmetric and transitive, but since **cred** is a preorder it is only reflexive and transitive.

Consider the conjecture

$$A \ \mathbf{cred}\ B = \mathit{true}$$

the question we address in this section is: under what circumstances may we use the **cred** axioms as rewrite rules to reduce the terms A and/or B without altering the validity of the conjecture? Since the PAM axioms never allow a conjecture to be disproved, we are more concerned with cases in which the reductions might turn a false statement into a true one, than in cases in which the reduction turns a true statement into a false one. In the latter case, the proof would be finalised by hand and we would expect the error to be detected then.

We consider the PAM axioms for **cred** as given in figure 8.2; note that only axioms relating to the second, third and fourth laws are given. The first law is ignored as this is implicit in the way PAM allows the reduction of subterms of a conjecture, assuming the language definition file also includes the PAM axioms for weak bisimulation congruence. Using the axioms of weak bisimulation congruence to reduce terms of a **cred** conjecture is correct since weak bisimulation congruence is a stronger relation, i.e. makes more distinctions, than **cred**. The validity of the conjecture is unaffected.

Similarly, we ignore the fifth law (transitivity) in our investigation as this is also implicit in the operation of PAM. Indeed the need to use this facility was one of the reasons for axiomatising **cred** as an equivalence. Application of the transitivity law does not change the validity of the conjecture. The sixth law, on the other hand, cannot be axiomatised in PAM: given the form of the conjecture there is no way to represent it in the PAM framework. If this law is required in a proof, the user must reason about the subparts separately and recombine the results manually. We analyse each of the other laws as PAM axioms below.

As PAM axioms may be used in either direction, we analyse the consequences of using the **cred** axioms both left to right and right to left. We note that the left to right direction is less likely to be used as this implies taking a deterministic expression and turning it into a nondeterministic one. In addition, using axiom CRED4 left to right is forbidden by PAM as the free variables of the right hand side of a rule must be included in the free variables of the left hand side of the rule. Also, using axiom CRED2 left to right may make the expression behave differently in a choice context, since an **i** action is added. Axiom CRED3 is the only one of the three likely to be used left to right.

Recall the definition of **red** as given in section 3.5.3.

$$A \;\mathbf{red}\; B \;\Leftrightarrow\; \forall t \in A^* \;\forall L \subseteq A.(B \;\mathbf{after}\; t) \;\mathbf{must}\; L \Rightarrow (A \;\mathbf{after}\; t) \;\mathbf{must}\; L$$

Our motivation for the analysis below is that by reducing either A or B the tests passed may be changed, thus changing the validity of A **red** B. The analysis below is really on A **red** B, but we write A **cred** B as the context does not affect the validity (as mentioned in the previous section).

For each PAM axiom we consider first how the tests passed differ in the left and right hand sides of the axiom; we then consider what that means in terms of increasing or decreasing tests passed when used left to right or right to left. After completing this analysis for each of the three axioms, we consider how using these axioms in a proof of (A **cred** B) $=$ *true* might affect the validity of the conjecture.

Axiomatising B cred i; B as B = i; B

The only distinction between B and $\mathbf{i}; B$ arises after the trace ϵ. B **after** $\epsilon \;=\; \{B\}$, whereas $\mathbf{i}; B$ **after** $\epsilon \;=\; \{\mathbf{i}; B, B\}$ The definition of **must** means that any leading **i** actions are ignored, therefore the set of states on each side is effectively $\{B\}$, and the tests passed by each side are the same.

Since the tests are not altered, we conclude that it is safe to use this axiom right to left on either A or B in (A **cred** B) $=$ *true*. As noted above, use of this axiom left to right is not appropriate as the introduction of an **i** action may cause the validity to change in a choice context, i.e. giving **red** instead of **cred**.

Axiomatising $g;(B_1 \mathbin{[]} B_2)$ **cred** $g;B_1 \mathbin{[]} g;B_2$ **as** $g;(B_1 \mathbin{[]} B_2) = g;B_1 \mathbin{[]} g;B_2$

For this axiom, several instantiations of t need to be examined. The table in figure 8.10 lays out the sets P **after** t for left and right hand sides for each possible t. As before, we use P' to denote the state set $\{P'' \mid P \xrightarrow{t} P''\}$.

trace, t	state set of $g;(B_1 \mathbin{[]} B_2)$ after t	state set of $g;B_1 \mathbin{[]} g;B_2$ after t
ϵ	$\{g;(B_1 \mathbin{[]} B_2)\}$	$\{g;B_1 \mathbin{[]} g;B_2\}$
g	$\{B_1 \mathbin{[]} B_2\}$	$\{B_1, B_2\}$
$gs, s \in tr(B_1) \cap tr(B_2)$	$B_1' \cup B_2'$	$B_1' \cup B_2'$
$gs, s \in tr(B_1), s \notin tr(B_2)$	B_1'	B_1'
$gs, s \notin tr(B_1), s \in tr(B_2)$	B_2'	B_2'
$gs, s \notin tr(B_1) \cup tr(B_2)$	$\{\}$	$\{\}$

Figure 8.10: States sets after selected traces in **cred** axiom CRED3

The figure shows that the state sets only differ when $t = \epsilon$ or $t = g$. We examine each case separately.

Case: $t = \epsilon$. Consider two possible values for L, either $L = \{g\}$ in which case both sides of the axiom pass the test, or $L = \{a\}$, where $a \neq g$, in which case both sides fail the test.

Case: $t = g$. Let $L = \{a\}$; there are four possible outcomes:

$a \in tr(B_1) \cap tr(B_2)$ Both sides of the axiom pass this test.

$a \in tr(B_1),\ a \notin tr(B_2)$ The right hand side of the axiom *fails* this test, as passing the test requires both states to pass. The left hand side *passes* the test.

$a \notin tr(B_1),\ a \in tr(B_2)$ As above.

$a \notin tr(B_1) \cup tr(B_2)$ Both sides of the axiom fail this test.

Axiom CRED3 right to left By considering the cases of t and L for CRED3 we see that using the axiom right to left may *increase* the number of tests passed, i.e. by combining B_1 and B_2 in a choice statement the possibility of passing a test is greater because the test is passed if either state passes the test, while before applying the axiom passing the test requires *both* states to pass the test.

Axiom CRED3 left to right Used in this direction, the axiom may *decrease* the number of tests passed. The reasoning is just the reverse of the above.

Axiomatising $g; B_1$ cred $g; B_1 \, [] \, g; B_2$ as $g; B_1 = g; B_1 \, [] \, g; B_2$

As for the previous axiom, we begin by analysing the states B **after** t, for left and right hand sides, and for different values of t. A table of these results is given in figure 8.11.

trace, t	state set of $g; B_1$ after t	state set of $g; B_1 \, [] \, g; B_2$ after t
ϵ	$\{g; B_1\}$	$\{g; B_1 \, [] \, g; B_2\}$
g	$\{B_1\}$	$\{B_1, B_2\}$
$gs, s \in tr(B_1) \cap tr(B_2)$	B_1'	$B_1' \cup B_2'$
$gs, s \in tr(B_1), s \notin tr(B_2)$	B_1'	B_1'
$gs, s \notin tr(B_1), s \in tr(B_2)$	$\{\}$	B_2'
$gs, s \notin tr(B_1) \cup tr(B_2)$	$\{\}$	$\{\}$

Figure 8.11: States sets after selected traces in **cred** axiom CRED4

The table shows that the state sets differ for $t = \epsilon$, $t = g$ and some cases of $t = gs$. We examine each case separately.

Case: $t = \epsilon$. Consider two possible values for L, either $L = \{g\}$ in which case both sides of the axiom pass the test, or $L = \{a\}$, where $a \neq g$, in which case both sides of the axiom fail the test.

Case: $t = g$. Let $L = \{a\}$, there are four possible outcomes:

$a \in tr(B_1) \cap tr(B_2)$ Both sides of the axiom pass this test.

$a \in tr(B_1)$, $a \notin tr(B_2)$ The left hand side of the axiom passes this test, but the right hand side fails the test as B_2 does not pass the test.

$a \notin tr(B_1)$, $a \in tr(B_2)$ Both sides of the axiom fail the test.

$a \notin tr(B_1) \cup tr(B_2)$ Both sides of the axiom fail this test.

Case: $t = gs, s \in tr(B_1') \cap tr(B_2')$. The results are similar to the above case analysis for $t = g$.

Case: $t = gs, s \in tr(B_1'), s \notin tr(B_2')$. For this value of t, the state sets of left and right hand sides of the axiom are the same, therefore the tests passed are the same.

Case: $t = gs, s \notin tr(B_1'), s \in tr(B_2')$. Let $L = \{a\}$, there are two cases:

$L = \{a\}, a \in tr(B_2')$ Both sides of the axiom pass this test, but note that the left hand side passed only because the state set after gs is empty.

$L = \{a\}, a \notin tr(B_2')$ The left hand side passes the test but the right hand side fails.

Case: $t = gs, s \notin tr(B_1') \cup tr(B_2')$. Again, the state sets are the same for both sides of the axiom, therefore the tests passed are the same.

Axiom CRED4 right to left By considering the cases of t and L in the above analysis, we see that the number of test may be *increased*, i.e. by throwing away a branch of the choice, we throw away all information about the tests which that branch may *fail*, and assume instead that all of those tests are passed.

Axiom CRED4 left to right As mentioned above, PAM does not allow use of this axiom left to right as a new free variable is introduced on the right hand side of the axiom.

Effect of axioms as reductions

Summarising, use of axioms CRED3 or CRED4 right to left may *increase* the number of tests passed by a behaviour expression, while use of CRED3 left to right may decrease the number of tests passed. We now consider how use of these axioms in reductions may affect the validity of a conjecture. Assume we are trying to show

$$A \textbf{ cred } B = \textit{true}$$

and we have, by applying the axioms as rewrite rules, $A \longrightarrow A'$ and $B \longrightarrow B'$. Under what circumstances can we substitute A' for A and B' for B in A **cred** $B = \textit{true}$?

Consider the use of the axioms right to left, which may increase the tests passed, i.e. A' passes more tests than A, and similarly for B' and B. Since, if $\neg(A \textbf{ cred } B)$, there is some test which B passes but which A does not, replacing A by A' may turn a false statement into a true one, as A' may pass the test which previously distinguished A and B. On the other hand, replacing B by B' might artificially create a distinguishing test, thereby turning a true statement into a false one.

From this analysis we may conclude that using the axioms in figure 8.2 right to left to reduce either term in a **cred** conjecture is not sound; however, while it is totally unacceptable to reduce A, as this may allow us to prove a false statement true in PAM, we may consider reducing B. Since we know the axioms are not complete and that we therefore cannot always prove a statement holds in PAM even though it holds in the model, and also that we can never prove in PAM that a conjecture does not hold, using the axioms to reduce B to B' is acceptable. The worst thing which can happen is that a bad reduction makes us unable to prove a true statement holds in PAM in which case we revert to a hand proof employing the technique of finding a distinguishing test.

Conversely, if we use the axioms from left to right, possibly decreasing the tests passed, it is acceptable to reduce A, but not B, for similar reasons as above.

In most proofs therefore, we use the **cred** axioms right to left, and on B *only*, in the conjecture A **cred** $B = true$. Occasionally we may also use the axiom CRED3 left to right on A. These restrictions will prevent us from making unsound reductions when using PAM for proofs involving the **cred** preorder.

8.5 Summary

In this chapter we have shown how our proof technique may be transplanted to a different equational reasoning tool, PAM. This change was necessary to allow us to reason about recursive processes. We first considered a number of ways in which rules for recursive processes could be added in a normal term rewriting framework, but without success.

We have also considered PAM formulations of the laws/axioms of other equivalences and congruences, and also for the preorder **cred**. We have shown that, while axiomatising **cred** as a predicate might be sound, we have to place such great restrictions on the rules to ensure soundness that the resulting proof system is considerably weaker than that given by the original laws of **cred** in the LOTOS standard. In order to regain some of that power, we axiomatise **cred** as an equivalence relation; again, some constraints must be placed on the axioms, but the system is more powerful than the one resulting from axiomatising **cred** as a predicate. In order to show the possible dangers of axiomatising **cred** as an equivalence, and also to show which uses of the axioms were safe, we analysed each of the axioms in terms of tests passed.

We note that really this is an unsatisfactory situation brought about by the inappropriateness of the equational reasoning paradigm as a setting for the **cred** preorder. One possible view is that we should ignore the preorder relations and stick to equivalence and congruence relations; however, this is unacceptable as preorders are valuable in expressing the verification requirements of a system. We have at least shown a way in which **cred** can be expressed within equational reasoning if necessary, even if it is a rather ugly solution which becomes dangerous if the recommended restrictions are ignored.

Having defined our proof system, we must next demonstrate its utility by application to a range of examples. This is the purpose of chapter 9.

Chapter 9

Further Studies using PAM

In this chapter we present a number of studies in verifying that an implementation of a system satisfies a specification of the same system. The purpose of these studies is to perform proofs in the system developed in the previous chapter, i.e. PAM in conjunction with the various sets of axioms, in order to demonstrate the applicability of the system to real examples.

When developing a proof system of any kind, it is always a worry that the examples carried out are somehow tailored, even subconsciously, to the quirks of the system. We have tried to avoid this by taking the examples, with one exception, from the papers of other authors. We note that in addition to taking the LOTOS or CCS descriptions of the systems, we also used the other authors' interpretation of the verification requirements, typically *specification* $\equiv_{wbc}$ *implementation.* The only example of our own is a repeat of the Login case study of chapter 7, this time with recursive behaviour. This is described in section 9.1.

The other examples presented are: a radiation machine [Tho94], section 9.2, the Readers and Writers problem, section 9.3, and the Candy machine, section 9.4, both from [DIN91], and the Scheduler [Mil89b, DIN89], section 9.5. The radiation machine is a fair-sized example, which introduces the verification technique of *property testing* and explores the difficulties of proving a specification does or does not exhibit a particular behaviour. The other examples are all relatively small and straightforward.

9.1 Login Case Study

With the system as described in the previous chapter, we now have the power necessary to complete a proof of correctness of the login case study *where the processes and the protocols are described using recursion.*

9.1.1 Reformulating the Example for PAM

The verification requirement is as before: to prove that the *implementation* satisfies the *specification*. The new process descriptions are obtained largely by exchanging all occurrences of **exit** by the appropriate (recursive) process call. Some modifications are required; this is described below. Since the example is substantially the same as the final, constraint-oriented, version of chapter 7, the details of the descriptions are not given here: see appendix D.4 for the PAM input files.

It is the nature of the constraint-oriented style of specification to use lots of parallelism; it is this parallelism in conjunction with recursion that causes us some difficulties when trying to express satisfaction using the new version of the Login example. The problem is that, as the processes are now recursive, and definitions can be unfolded repeatedly, extra opportunity arises for the expansion of a parallel statement to produce independent actions. For example, given the process descriptions $A = a; b; \textbf{exit}$ and $B = b; c; \textbf{exit}$, then $A \mid[b]\mid B = a; b; c; \textbf{exit}$. However, if we replace the occurrences of **exit** by A or B as appropriate and make the processes recursive, $A \mid[b]\mid B$ becomes:

$$\begin{aligned}(A \mid[b]\mid B) = a; b; (\ & c; (A \mid[b]\mid B) \\ & [\,]\ a; (b; A \mid[b]\mid c; B))\end{aligned}$$

Similar problems arise with the login example. Consider two definitions relating to the timer (in their original form):

```
timer    = set; (tcancel; exit [] timeout; exit)
timer_on = p4; set; exit
```

Combining these processes in parallel, we get

```
timer |[set]| timer_on  = p4; set; (tcancel; exit [] timeout; exit)
```

Now consider the recursive forms obtained by replacing **exit** by `timer` or `timer_on`:

```
timer    = set; (tcancel; timer [] timeout; timer)
timer_on = p4; set; timer_on
```

The result of trying to combine the processes `timer` and `timer_on` in parallel is as follows, using only unfolding and expansion:

```
  timer |[set]| timer_on
= p4; set; ((tcancel; timer [] timeout; timer) |[set]| timer_on)
= p4; set; ((tcancel; timer [] timeout; timer) |[set]| (p4; set; timer_on))
= p4; set; (   p4; ((tcancel; timer [] timeout; timer) |[set]| (set; timer_on))
            [] tcancel; (timer |[set]| timer_on)
            [] timeout; (timer |[set]| timer_on))
  ⋮
```

Although the trace we want appears, i.e. the trace produced by the original expression, we also get a trace which allows p4 to occur again before either `tcancel` or `timeout` occurs. This possible trace does not match with our intuitions about how the protocols should behave. We might expect some other constraint to restrict this possible behaviour, but in this particular case we know that no other process describes the relationship between these events. To rectify this situation, a new *timer* process is described which incorporates activation of the timer and the events which occur after the operation of the timer. In some ways this is an unsatisfactory solution as the specification is now less modular than it was previously; however, there seems to be no other way around the above problem.

9.1.2 Proof of the Verification Requirement

We now present the proof of satisfaction for the Login case study as carried out in PAM on the conjecture `PROCESSES = PROTOCOLS`, where `=` stands for $\equiv_{wbc}$.

The only proof techniques required for the proof are the ones built into PAM: the expansion law, folding and unfolding of definitions (including recursive definitions), substitution of a subterm reduced in an earlier section, and unique fixed-point induction (ufi); none of the axioms defined for weak bisimulation congruence are required.

The details of the proof are uninteresting, consisting of expanding each parallel expression to end up with an expression using only choice and sequencing in the initial part and calling the original process expression recursively. For example,

```
    P2 |[second]| DEALLOC_C
≡wbc
    m3;(n3;P2 [] p3;(P2 [] m6;p6;P2))
    |[second]| m3;(p3;m6;p6;DEALLOC_C [] n3;DEALLOC_C)
≡wbc
    m3;(n3;(P2 |[second]| DEALLOC_C)
      [] p3;((P2 [] m6;p6;P2) |[second]| m6;p6;DEALLOC_C))
≡wbc
    m3;(n3;P2' [] p3;((P2 [] m6;p6;P2) |[second]| m6;p6;DEALLOC_C))
≡wbc
    m3;(n3;P2' [] p3;((m3;(n3;P2 [] p3;(P2[]m6;p6;P2)) [] m6;p6;P2)
                      |[second]| m6;p6;DEALLOC_C))
≡wbc
    m3;(n3;P2' [] p3;m6;p6;(P2 |[second]| DEALLOC_C))
≡wbc
    m3;(n3;P2'[]p3;m6;p6;P2')
```

where `P2'` is a new name defined to be `P2 |[second]| DEALLOC_C`. This serves to make the proof less cluttered. The other expansions proceed in a similar fashion.

We note that in `PROCESSES`, `A`, `C` and `D` are all absorbed by `B`, i.e.

```
PROCESSES = ((B |[first]| A) |[second]| C) |[third]| D = B
```

where

```
    first     = m1, p1, n1
    second    = m3, n3, p3, m6, p6
    third     = m4, n4, p4, m5, p5, m7, p7
```

Having unfolded both PROCESSES and PROTOCOLS we are able to use unique fixed point induction via the built-in function ufi to conclude that the conjecture PROCESSES $\equiv_{wbc}$ PROTOCOLS holds, i.e. the expressions for PROCESSES and PROTOCOLS are syntactically identical modulo occurrences of the process name.

Completion of this example shows that in the new proof system we have achieved the goal stated in the discussion of chapter 7 of having more control over the proof process and of being able to perform proofs of equivalence of recursive processes.

9.2 A Simple Radiation Machine

In [Tho94], a simple radiation machine, based on the Therac 25, is presented in order to demonstrate the use of formal specification and verification techniques. The aim of the exercise is to prove the safety, or otherwise, of several variants of the machine. The specifications are given in Basic LOTOS (later specifications use full LOTOS, but these are ignored at present) and several different approaches to verification (trace analysis, property testing and temporal logic) are used to reason about the safe and unsafe behaviour of three variants of the machine. In this section, the property testing part of the experiment is repeated using the system developed in chapter 8. We explore the safety or otherwise of *four* variants of the `Therac` machine. Three of these are the ones given in [Tho94]; the fourth, called `SimpleTherac` here, is developed for *this* experiment to provide a simpler example on which to explore techniques for proving the safety of the machine. All specifications, unless noted otherwise, are as in [Tho94].

In [Tho94] the original approach to the verification (property testing, automated by the LOLA tool [LITE]) was only able to confirm *unsafe* behaviours of a machine, and could not be used to prove a machine safe. The approach relies on building an Extended Finite State Machine (EFSM) representing the process and analysing the transitions of this EFSM. In an unsafe machine, this analysis can reveal the "bad" behaviours, usually fairly quickly. However, although LOLA can recognise states which it has visited before, the EFSM for one of the safe machines (`Therac1b`) is infinite, and LOLA can continue forever in an attempt to find bad behaviours. This problem occurs because of the way in which `Therac1b` is defined (a problem with the `[>` operator). In the experiments of [Tho94] the tool runs out of memory, therefore although the confidence of the user in the correctness of the specification is increased, there is always the possibility that the tool ran out of memory just before a bad behaviour was discovered. Here, the strange behaviour of the `[>` is noted, and we are able to re-express `Therac1b` without `[>`; the new process is equivalent to the original version of Therac1b, and can be shown to be safe. Note however, that, in contrast to [Tho94], our proofs are not fully automated. Although PAM can be used to obtained a reduced form of the specification, the proof must be finished by hand.

While the main results of our experiment are the successful proofs of the safety or otherwise of the four versions of the radiation machine, another result is to "debug" the specifications originally given in an unpublished draft of [Tho94]. Our findings have been noted by the author and incorporated in the published version.

The basic specification of the radiation machine, Therac1, is presented in the next section. The verification requirement, i.e. how the safety of the machine is measured, follows in section 9.2.2. The proof that Therac1 is unsafe is given in section 9.2.3 and the following two sections present the variations on the basic Therac specification and the proofs of safety (or otherwise, as the case may be) of those machines.

9.2.1 LOTOS Specification of the Radiation Machine

The LOTOS specification of the radiation machine is given in figure 9.1. The computer controlled radiation machine provides two forms of treatment: electron and xray. The electron treatment is produced by firing a low intensity electron beam directly at the patient, while for xray treatment the intensity of the beam is higher and the beam is scattered through a shield to produce xrays rather than being fired directly at the patient. In the LOTOS processes the setting of the position of the shield and the intensity of the beam is modelled by the events ls (low shield), hs (high shield), lb (low beam) and hb (high beam). The beam and shield are initially both set to low in the process STARTUP. The two forms of treatment are reflected by corresponding processes in the LOTOS specification. The operator chooses the form of treatment by selecting the event el or xr and doses the patient by selecting the fire event. The XRAY process sets the beam and shield to high (using the auxiliary process SETUP), fires, and then resets the beam and shield to low (again using SETUP). The ELECTRON process assumes the shield and beam are set to low, therefore the only event this process performs is the fire event. Once a particular form of treatment is selected the operator may abort the treatment and return to the initial choice between xray and electron treatment. The ability to abort the treatment and begin again is reflected by the use of the special LOTOS disable operator, written [>, which was designed specifically for this sort of behaviour.

9.2.2 Expressing the Verification Requirements

As stated in [Tho94], the most important verification/safety requirement for the radiation machine is that it should not fire when the beam strength is high and the shield is low as this would result in a lethal dose of radiation for the patient. A further safety requirement would be to also consider

```
specification Theracl [fire, lb, hb, ls, hs, xr, el] :exit

behaviour
STARTUP [fire, lb, hb, ls, hs, xr, el]
where

process STARTUP [fire, lb, hb, ls, hs, xr, el] :exit :=
    SETUP [lb, ls] >> TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc

process SETUP [ev1, ev2] :exit :=
    (ev1; exit) ||| (ev2; exit)
endproc

process TREATMENT [fire, lb, hb, ls, hs, xr, el] :exit :=
        xr; XRAY [fire, lb, hb, ls, hs, xr, el]
    []  el; ELECTRON [fire, lb, hb, ls, hs, xr, el]
    []  exit
endproc

process ELECTRON [fire, lb, hb, ls, hs, xr, el] :exit :=
    (fire; TREATMENT [fire, lb, hb, ls, hs, xr, el])
        [> TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc

process XRAY [fire, lb, hb, ls, hs, xr, el] :exit :=
    (SETUP [hb, hs] >> (fire; SETUP [lb, ls])
        >> TREATMENT [fire, lb, hb, ls, hs, xr, el])
    [> TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc
endspec
```

Figure 9.1: Theracl Specification

the situation in which the beam is low and the shield is high as dangerous, as this results in the patient getting insufficient radiation to treat their illness. For simplicity, we consider only the former requirement.

In [Tho94] the verification requirement is formalised by specifying the traces which precede a lethal dose of radiation as a regular expression:

```
(not (hb | hs))*; hb; (not (lb | hs | fire))*; fire
```

where * denotes zero or more occurrences, | denotes choice, ; sequencing, and **not (x | y)** denotes the choice of all events excluding x and y. The given regular expression says that a bad trace is one in which a `fire` event is preceded by a `hb` event but not a `hs` event. The trace will always start with `((lb; ls) | (ls; lb))`, representing the initial setting of the beam and shield. A minimal example of a bad behaviour is `lb; ls; xr; hb; el; fire`.

In [Tho94] the proof technique is based on showing the ability or inability of the `Therac` specification to perform traces of the above form. In other words, we use a bad trace as a *test.* If `Therac` can pass the test, then we know that the machine is not safe. This technique is generally known as *Property Testing.* This approach is used because it is simpler to give samples of possible bad sequences of behaviour than it is to specify all possible good behaviours. Another reason for specifying bad behaviour rather than good behaviour is that the good behaviour has already been specified once, i.e. in the description of the machine itself. If a mistake was made in that specification, it is possible that the same mistake will be repeated when specifying *good* tests. In order to avoid this potential problem, the machine is specified from a different angle the second time, namely the *bad* behaviours.

In general, a proof using the property testing technique proceeds by describing the test (which may be a good or a bad sequence of events) as a LOTOS process. The last event in the test is the user-defined special event **testok**, which indicates successful completion of the test. The test process is then composed in parallel with the process under test, synchronising on all events in the test except **testok**, i.e. $P \,|[\mathcal{L}(P)]|\, TEST$, where P is the system under test. Due to the multi-way synchronisation of the LOTOS parallel operator, if the process under test can perform the behaviour in the test process, then eventually the **testok** event will be observed, and we say that the process passes the test. Typically, the proof is automated by employing a simulation tool to execute the process, although specialised testing tools also exist. The original exercise of [Tho94] used the testing tool LOLA, which is part of the LITE toolkit [LITE]. The LOLA tool is essentially a simulator, and operates by simulating the process $P \,|[\mathcal{L}(P)]|\, TEST$ until either the **testok** action is observed, a duplicate state is reached, or the tool runs out of memory.

We may also express property testing in terms of a relation between processes. For our experiment with the radiation machine we use the following basic form:

```
    testok; exit cred THERACTEST
where
    THERACTEST = hide therac_events in
                        STARTUP |[therac_events]| lb; ls; (TEST >> testok; exit)
```

The use of **cred** expresses the notion that at least one trace of THERACTEST has the behaviour we are looking for (although there may be lots of other behaviours). An equivalence relation would also take account of these other behaviours and is therefore too strong. The **therac_events** must be hidden in the combined test process, THERACTEST, to make comparison with **testok; exit** possible.

The full definition of the TEST process is given in figure 9.2. The TEST process is preceded by setting the beam and shield to low (which occurs in the STARTUP process), and when we exit from the TEST process the **testok** action is performed and `Therac` passes the test. For full generality we should give the option to perform `ls` before `lb`, but this would make no difference to the validity of

```
process TEST[fire, lb, hb, ls, hs, xr, el] :exit :=
    Nothbhs[fire, lb, ls, xr, el]
        >> (hb; Notlbhs[fire, hb, ls, xr, el])
            >> (fire; exit)
endproc

process Nothbhs[fire, lb, ls, xr, el] :exit :=
       fire; Nothbhs[fire, lb, ls, xr, el]
    [] lb; Nothbhs[fire, lb, ls, xr, el]
    [] ls; Nothbhs[fire, lb, ls, xr, el]
    [] xr; Nothbhs[fire, lb, ls, xr, el]
    [] el; Nothbhs[fire, lb, ls, xr, el]
    [] exit
endproc

process Notlbhs[hb, ls, xr, el] :exit :=
       ls; Notlbhs[hb, ls, xr, el]
    [] hb; Notlbhs[hb, ls, xr, el]
    [] xr; Notlbhs[hb, ls, xr, el]
    [] el; Notlbhs[hb, ls, xr, el]
    [] exit
endproc
```

Figure 9.2: The Unsafe Test Process

the proof and is therefore unnecessary. For all of the `Therac` examples, the test process describes a bad behaviour, therefore if the specification passes the test it means that it is *unsafe*.

As mentioned in the introduction, use of the LOLA tool for the radiation machine example is appropriate in the case that the test is passed (within a small number of unfoldings), but less so for the safe machines, which potentially require infinite unfolding. As the power of the tool is limited by the number of unfoldings it can perform, and therefore by the physical limitations of the machine on which it executes, this can result in inconclusive test results, i.e. the test has not been passed, but the unfolding is aborted due to lack of memory. This should not occur in the `Therac` examples, as LOLA can recognise states which have been visited before, e.g. recursive calls of a process. Unfortunately, the `Therac1b` process is more complicated, and LOLA is unable to recognise duplicate states and therefore cannot show that the test is rejected, i.e. that the machine is safe. We also encounter the same problem, but the close interaction required of the user by PAM makes it easier to identify the cause of the problem. The problem and our solution is described in section 9.2.5.

In the following sections, proofs of the safety of several variants of the example are attempted as follows: in section 9.2.3 the original machine of [Tho94], referred to here as `Therac1`, is shown to be unsafe, in section 9.2.4 our machine `SimpleTherac` is shown to be safe, and finally, in section 9.2.5 two modified versions of machine, also presented in [Tho94], are shown to be unsafe and safe respectively. In all cases the main part of the proof is carried out in PAM, with the proofs being finished by hand where necessary.

Note that some modifications to the specification are necessary because of limitations of PAM. These are merely syntactic rather than semantic and do not affect the proof results. In particular, parameter passing is not supported by PAM. The main effect of this is that instead of having a parameterised process SETUP as in [Tho94] we have two processes SETUPH and SETUPL which are hardwired versions of SETUP [hb, hs] and SETUP [lb, ls] respectively. The other processes of the specification are also parameterised, but the list of actual parameters matches the list of formal parameters, therefore having no effect on the process behaviour.

9.2.3 Proving Therac1 is not safe

To show that the machine, as described in figure 9.1, is unsafe, we attempt to prove the following conjecture holds using PAM:

```
((testok; exit) cred THERACTEST) = true                                    (*)
```

The proof begins by reducing THERACTEST as much as possible by unfolding the definitions of the processes, applying various laws preserving weak bisimulation congruence and applying the laws which allow the process to be expressed using only the ;, [] and **hide** operators. The **hide** operator could also be removed, but is retained to allow the reduction to be followed more easily. Below we give a condensed version of the proof procedure, showing important intermediate stages. In performing this unfolding we use the bad trace lb; ls; xr; hb; el; fire to guide us to an occurrence of the testok action.

```
    THERACTEST
≡wbc
    hide therac_events in
        lb; ls; (   i; stop
                 [] el; ELTEST
                 [] xr; (   el; ELTEST
                         [] xr; XRTEST
                         [] i; hb; (   i; stop
                                    [] xr; XRTEST'
                                    [] el; (   i; fire; testok; exit
                                            [] xr; XRTEST'
                                            [] el; ELTEST'))))
```

where

```
    ELTEST  = ELECTRON |[therac_events]| (TEST >> testok; exit)
    XRTEST  = XRAY |[therac_events]| (TEST >> testok; exit)
    ELTEST' = ELECTRON |[therac_events]|
                  (Notlbhs >> fire; exit >> testok; exit)
    XRTEST' = XRAY |[therac_events]|
                  (Notlbhs >> fire; exit >> testok; exit)
```

We stop here because the `testok` action has appeared in the unfolding.

The reduced form of THERACTEST given above may be further simplified by applying laws which push the `hide` operator further into the expression, turning all hidden events into occurrences of **i**. Once this is done the laws of weak bisimulation and testing congruence can be applied to further simplify the expression by removing sequences of internal actions and duplicated branches. Finally we obtain

```
    THERACTEST
≡tc
        i; testok; exit
    []  i; stop
    []  i; hide therac_events in ELTEST
    []  i; hide therac_events in XRTEST
    []  i; hide therac_events in ELTEST'
    []  i; hide therac_events in XRTEST'
```

By substituting this reduced form of THERACTEST for THERACTEST in (∗) the expression which must be proved to hold can now be expressed as:

```
    (testok; exit    cred       i; testok; exit                        = true
                             [] i; stop
                             [] i; hide therac_events in ELTEST
                             [] i; hide therac_events in XRTEST
                             [] i; hide therac_events in ELTEST'
                             [] i; hide therac_events in XRTEST')
```

which can be shown to hold in PAM by application of the **cred** axioms, CRED4, CRED2 and CRED7, developed in section 8.4.

9.2.4 Proving `SimpleTherac` is safe

It is relatively easy to prove that the `Therac1` specification is unsafe since all that is required is to show that at least one bad trace is possible. If a particular example of a bad trace, preferably the shortest one, is known in advance, the proof is less tedious as there is some notion of where the proof is going and what is being aimed for in unfolding definitions and applying axioms. In this section we consider the case in which the specification must be proved safe, and therefore no such bad trace exists.

By examining the original `Therac1` we see that it is the ability to interrupt the machine while the beam and shield are being set for xray treatment which leads to the firing of the high beam with a low shield. As a simple preliminary exercise in proving a specification safe, we look at `Therac1` with the disable sequences omitted, i.e. we remove the ability to abort the treatment. This particular modification was not discussed in [Tho94]; the safe modification presented there is considered in section 9.2.5.

```
specification SimpleTherac [fire, lb, hb, ls, hs, xr, el] :exit

behaviour
STARTUP [fire, lb, hb, ls, hs, xr, el]
where

process STARTUP [fire, lb, hb, ls, hs, xr, el] :exit :=
    SETUP [lb, ls] >> TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc

process SETUP [ev1, ev2] :exit :=
    (ev1; exit) ||| (ev2; exit)
endproc

process TREATMENT [fire, lb, hb, ls, hs, xr, el] :exit :=
        xr; XRAY [fire, lb, hb, ls, hs, xr, el]
    []  el; ELECTRON [fire, lb, hb, ls, hs, xr, el]
    []  exit
endproc

process ELECTRON [fire, lb, hb, ls, hs, xr, el] :exit :=
    fire; TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc

process XRAY [fire, lb, hb, ls, hs, xr, el] :exit :=
    (SETUP [hb, hs] >> (fire; SETUP [lb, ls])
        >> TREATMENT [fire, lb, hb, ls, hs, xr, el])
endproc
endspec
```

Figure 9.3: Simplified Therac Specification

The new `SimpleTherac` description is as in figure 9.3. The only changes are to the processes `ELECTRON` and `XRAY`. The test process is as before.

Whereas in the proof of safety of `Therac1` all that was required was to exhibit an occurrence of a bad trace, showing the machine was not safe, here we must show that *none* of the traces of `SimpleTherac` is a bad trace and that the machine is therefore safe. In the previous section, the main part of the proof consists of unfolding the behaviour until a `testok` action is reached. Here there should be no occurrence of `testok`, and, since the processes of `SimpleTherac` are recursive, the unfolding procedure is potentially infinite. The `SimpleTherac` proof proceeds by unfolding the behaviour once or twice as necessary and analysing the behaviour of the initial portion of the unfolding and using that analysis to deduce properties of the machine as a whole. In more abstract terms, if we start with a process X, we attempt to unfold this process to give an expression which involves only X (and some events), but no other process names, e.g. $X = a; b; c; X$. An equation of this form tells us that all unfoldings of X will result in traces of the form $(abc)^*$, allowing us to conclude, for example, that no d event will occur.

Consider the initial unfolding of THERACTEST, produced in PAM using the weak bisimulation congruence PAM axioms:

```
    THERACTEST
≡wbc
    hide therac_events in
    STARTUP |[therac_events]| (lb; ls; TEST) >> testok; exit
≡wbc
    hide therac_events in
    lb; ls; (TREATMENT |[therac_events]| (TEST >> testok; exit))
≡wbc
    i; (hide therac_events in
    TREATMENT |[therac_events]| (TEST >> testok; exit))
```

Two important expressions in the above unfolding are:

```
hide therac_events in (TREATMENT |[therac_events]| (TEST >> testok; exit))
```

which we will refer to as TREATTEST, and

```
TREATMENT |[therac_events]| (TEST >> testok; exit)
```

i.e. the above without the **hide** operator, which we will refer to as TREATTEST′.

Instead of trying to find an occurrence of testok in TREATTEST we try to unfold TREATTEST, i.e. **hide** therac_events **in** TREATTEST′, to get a process expression in which TREATTEST′ (and no others) is referred to again. Unfolding in PAM:

```
    hide therac_events in TREATTEST′
≡wbc
    hide therac_events in
    (   i; stop
     [] xr; hb; stop
     [] el; (   i; stop
             [] fire; TREATTEST′))
```

By the uniqueness of solutions to equations we can show that this process is equivalent to one which does not even have the testok event in its language, allowing us to conclude that this version of the machine is safe.

Note that often we do not push the **hide** operator through the process expression as this can obscure the origins of expression by hiding the identity of events which cause those actions.

For example, the expression above, TREATTEST is weak bisimulation congruent to

```
i; (hide therac_events in TREATTEST [] i; stop)
```

With the actions hidden, it is harder to see what is happening, but it *is* obvious that neither branch leads to an occurrence of a testok.

In this example we are really reasoning about the state of the beam and shield, which could be regarded as an underlying state of the process. The actions `lb`, `ls`, `hb` and `hs` are the operations which allow this state to be altered, but we have no way of explicitly examining the current value of the state, which is what makes reasoning about it so difficult. In section 10.3.2 a version of the machine which makes this state explicit is considered, thus allowing reasoning about the state.

9.2.5 Proving the Modified `Therac1` is safe

The above version, `SimpleTherac`, of the machine cannot be used as a serious alternative to the original radiation machine as the operator is committed to a course of action as soon as the xray or electron button is pressed. This is bad because the operator has no facility for changing the dosage: once either of the processes `XRAY` or `ELECTRON` has started we must continue to the `fire` event and we may not abort the processes. In the original paper [Tho94] a modification is given which allows the machine to maintain the ability to abort from a treatment, but which is also safe. We now consider this modification here.

The problem in the `Therac1` specification is that the beam and shield may be set incorrectly because of the ability to abort a treatment. Rather than remove the ability to abort the treatment, as in `SimpleTherac`, we look at how the beam and shield are set. The process `SETUP` allows the beam and shield to be set in any order, but the ordering of events could be fixed so that the beam is only made high if the shield is already up, and the shield is lowered only after the beam is set to low. This adjustment should rectify the problem of firing the high beam when the shield is down.

An alternative solution to this problem would be to introduce a language construct to wrap up sequences of events which should be performed as atomic, thus ensuring that the beam and shield are set together. The need for such a construct was already mentioned in section 7.6.1. We shall only study the former solution to the problem here.

The LOTOS for the modified `SETUP` process is as in figure 9.4. As before, we modify this for PAM to give hardwired versions of the parameterised processes.

```
process SETUP [ev1, ev2] :exit :=
   ev1; ev2; exit
endproc
```

Figure 9.4: The New SETUP Process

In addition to the new definition of `SETUP` there is another important alteration to `Therac1`. For the machine to be safe, the calls to `SETUP` in `XRAY` must also be altered (the arguments have to be in the right order). The new version of `XRAY` is as in figure 9.5. The rest of the machine is as in figure 9.1.

```
process XRAY [fire, lb, hb, ls, hs, xr, el] :exit :=
    (SETUP [hs, hb] >> (fire; SETUP [lb, ls])
        >> TREATMENT [fire, lb, hb, ls, hs, xr, el])
    [> TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc
```

Figure 9.5: The New XRAY Process

If this change to the parameters of SETUP is not made, then the machine can still deliver lethal doses of radiation.

We shall refer to the first version (with the parameters in the wrong order) as Therac1a and the second version (with the parameters in the correct order) as Therac1b. We prove that Therac1a is still unsafe, but that Therac1b is safe. In the draft of [Tho94] this change to the order of the parameters was not mentioned, but it was claimed that the new version did not pass the test, i.e. was safe. In fact, LOLA ran out of memory before the test could pass or fail, thus the conclusion is that confidence in the correctness of the system is increased, even though the system may still not be safe.

Therac1a

As with the original Therac1 proof, we proceed by trying to follow a trace which we know leads to testok being performed. The trace is the same as before, as is the final process expression obtained (allowing for the modification of XRAY). Our conclusion is that Therac1a is *not* safe.

Therac1b

Since we believe this process to be safe, we attempt to use the same approach as for the SimpleTherac proof of section 9.2.4. We unfold THERACTEST to try to obtain a recursive expression referring only to TREATTEST, or TREATTEST′ if hide is not pushed through the expression. Although this approach was successful for the SimpleTherac proof, it is not successful here. Below we give some stages in the unfolding of THERACTEST.

```
    THERACTEST
≡wbc
    hide therac_events in
        lb; ls; TREATMENT |[therac_events]| lb; ls; (TEST >> testok; exit)
≡wbc
    hide therac_events in
        lb; ls; TREATTEST′
≡wbc
    i; hide therac_events in TREATTEST′
≡wbc
```

```
i; hide therac_events in
    lb; ls; (   i; stop
             [] xr; XRTEST
             [] el; ELTEST)
```

where

```
XRTEST = XRAY |[therac_events]| (TEST >> testok; exit)
       = i; stop [] xr; XRTEST [] el; ELTEST
ELTEST = ELECTRON |[therac_events]| (TEST >> testok; exit)
       = i; stop [] xr; XRTEST [] el; ELTEST
         [] fire; ((TREATMENT [> TREATMENT)
                  |[therac_events]| (TEST >> testok; exit))
```

In ELTEST, instead of getting TREATTEST′ after a `fire` event, we get

```
((TREATMENT [> TREATMENT) |[therac_events]| (TEST >> testok; exit))
```

Unfolding *this* expression results in similar expression to the unfolding of THERACTEST, but with extra occurrences of the [> operator in various places. In particular, after `fire` in the electron phase of treatment we get

```
(((TREATMENT [> TREATMENT) [> TREATMENT
    |[therac_events]| (TEST >> testok; exit))
```

We can deduce from this that each unfolding will have the same effect, giving larger process expressions each time, none of which appear to be the same as TREATTEST′. In the original paper [Tho94] this is why the tests are inconclusive, because LOLA cannot recognise TREATMENT as the same process as TREATMENT [> TREATMENT. What we would like is to have an axiom which allows the application of the disable operator to be eliminated, for example $P\,[>\ P\ =\ P$, but this conjecture does not hold. This can be proved by counter example: let $P\ =\ a;\ b;\ P$, then $P\,[>\ P\ =\ P\ []\ a;\ (P\ []\ b;\ P\,[>\ P)$ which is obviously not the same as P since a possible trace of $P\,[>\ P$ is $\langle a, a, b\rangle$ which can never occur as a trace of P.

The situation is not hopeless, as it is possible that the full power of such a general law is not required. The problem in the unfolding is that the disable operator, if not activated, gets passed on to the next recursion of the process, which is not what we want in the proof. Perhaps the form of the above conjecture is incorrect: what we really want to say is, given the following definitions and unfoldings:

$$\begin{array}{lll} P & \overset{def}{=} & a; b; P \\ P_1 & \overset{def}{=} & P\ []\ a; (P\ []\ b; P_1) \\ P\,[>\ P & \equiv_{wbc} & P\ []\ a; (P\ []\ b; (P\,[>\ P)) \end{array}$$

where $P\,[>\ P$ is generated by the disable expansion law, $P\,[>\ P$ is a solution for P_1, i.e. if we substitute $P\,[>\ P$ for P_1 in the second equation above we get the third equation. In other words, by uniqueness of solutions $P_1 \equiv_{wbc} P$ [> P. Although we use a particular example above, what we are saying is that the disable expansion law can be used to unfold occurrences of [> , and the expression P [> P can be replaced by a variable name. This equivalence is taken for granted when the disable operator is used.

In PAM we are unable to redefine operators in this way, so instead we rewrite the Therac1b specification, replacing occurrences of the disable operator in the ELECTRON and XRAY processes by the equivalent expression involving only choice and sequencing; see figures 9.6 and 9.7. These new processes are equivalent to the original ELECTRON and XRAY processes by uniqueness of solutions to recursive equations.

```
process ELECTRON [fire, lb, hb, ls, hs, xr, el] :exit :=
       TREATMENT [fire, lb, hb, ls, hs, xr, el]
    [] fire; TREATMENT [fire, lb, hb, ls, hs, xr, el]
endproc
```

Figure 9.6: ELECTRON with no occurrence of [>

```
process XRAY [fire, lb, hb, ls, hs, xr, el] :exit :=
   TREATMENT [fire, lb, hb, ls, hs, xr, el]
[] hs; (   TREATMENT [fire, lb, hb, ls, hs, xr, el]
        [] hb; (   TREATMENT [fire, lb, hb, ls, hs, xr, el]
                [] fire; (   TREATMENT [fire, lb, hb, ls, hs, xr, el]
                          [] lb; (   TREATMENT [fire, lb, hb, ls, hs, xr, el]
                                  [] ls; TREATMENT [fire, lb, hb, ls, hs, xr, el]))))
endproc
```

Figure 9.7: XRAY with no occurrence of [>

The unfolding of THERACTEST is now as follows:

```
   THERACTEST
≡wbc
   i; hide therac_events in TREATTEST'
≡wbc
   i; hide therac_events in
          (   i; stop
           [] xr; (   TREATTEST'
                   [] i; stop)
           [] el; (   TREATTEST'
                   [] fire; TREATTEST'))
```

Again, we have produced an expression which refers to itself recursively, so to complete the proof of safety we use the uniqueness of solutions to recursive equations theorem to show that the above process is equivalent to a process in which the event testok does not occur. We conclude that the machine Therac1b is safe.

We note that in the original presentation [Tho94], Thomas is unable to prove the safety of this machine using LOLA. Although LOLA has the ability to recognise previously explored states, like us it cannot identify `TREATMENT [> TREATMENT` and `TREATMENT`. Since LOLA is a simulation tool, it is not possible to carry out any manipulation of the term `TREATMENT [> TREATMENT`, which is why the original paper does not investigate this problem further. Using PAM, however, we are able to manually identify an equivalent expression, without the `[>` operator, which allows us to successfully reason about the safety of the machine.

We also note that if the original experiment with LOLA is repeated with the modified versions of `ELECTRON` and `XRAY`, as in figures 9.6 and 9.7, LOLA is able to identify loops in the process and to state that the unsafe test is rejected, i.e. the machine `Therac1b` is safe.

9.2.6 Summary and Discussion

In this section we have examined four specifications of the radiation machine and proved their safety or otherwise. This was largely a repetition of the experiment presented in [Tho94], using a modification of the proof technique and a different method of automation. We note that we were able to conclude with certainty that `Therac1b` was safe, whereas this was not possible in [Tho94]. Although the proofs that two of the machines were unsafe were straightforward, the same could not be said of the proofs that the other two machines were safe. In the original presentation the method was incomplete, in that the proof tool seemed geared towards finding bad behaviours, or of proving finite processes safe. In an infinite process the tool ran out of memory and aborted before finding a bad behaviour. Here we used additional proof techniques (applied by hand) to *mathematically* prove the safety of two of the machines. The lack of automation is the price which has to be paid for a more powerful proof technique.

An example of an additional proof technique required in the radiation machine verification is the reference to the underlying state of the beam and shield in the analysis of `SimpleTherac`. Although this state is implicit here, it can be made explicit by the introduction of *abstract data types* modelling the state of the beam and shield. This part of LOTOS has been ignored up till now, but is considered in chapter 10. In section 10.3.2 the modified `Therac` example is analysed in the setting of full LOTOS.

Another interesting problem uncovered during this study is the difficulty of reasoning about expressions involving the disable operator. In fact, the verification of `Therac1b` could not be completed until the expression `TREATMENT [> TREATMENT` had been replaced by an equivalent expression involving only sequence and choice.

Our main aim in studying this example was to demonstrate the PAM axioms developed in chapter 8. This particular example provides the opportunity to use the **cred** axioms developed in the previous chapter, as well the more commonly used weak bisimulation congruence axioms. Our aim has therefore been achieved. In addition, we have also gained more experience of the process of verification, and uncovered some more problems in that process.

9.3 Readers and Writers

This example is taken from [DIN91]. It is based on the problem of ensuring that a shared resource is accessed mutually exclusively by two processes (a reader and a writer). Two LOTOS descriptions are given: one which specifies the mutual exclusion property (the *specification*), and one which implements the access of the reader and the writer (the *implementation*). The verification requirement is then to prove that the specification is satisfied by the implementation. Minor changes have been made to the example from the original presentation in [DIN91]; these are detailed in the next section.

9.3.1 The LOTOS Descriptions

The specification of the readers and writers problem is given in figure 9.8. This says that either the reader has access to the resource or the writer does, but never both at the same time. If we model reading and writing by the actions `r` and `w`, then scheduling these activities and ensuring mutual exclusion would be trivial (because of the interleaving semantics of LOTOS). Instead, we model the beginning and ending of these activities separately, i.e. we have four actions: `rb` (reader begin), `wb` (writer begin), `re` (reader end) and `we` (writer end). This also models the fact that the reading and writing activities have some duration.

```
specification Readers_And_Writers_Spec [rb, re, wb, we] :noexit

behaviour
Spec [rb, re, wb, we]
where

process Spec [rb, re, wb, we] :noexit :=
   i; rb; re; Spec [rb, re, wb, we] [] i; wb; we; Spec [rb, re, wb, we]
endproc
endspec
```

Figure 9.8: The Readers and Writers Specification

A proposed implementation of the specification is given in figure 9.9. The actions of the reader and the writer are interleaved. Both processes are forced to synchronise with a *semaphore*, which ensures mutual exclusion. The reader and writer are created by instantiating the general `Proc`, i.e. the reader is the process `Proc [p, v, rb, re]` and the writer the process `Proc [p, v, wb, we]`. This is different from [DIN91] where the reader and the writer were defined individually. Another minor alteration is that the semaphore is more general (using only p and v rather than specific pr and vr actions for the reader and pw and vw actions for the writer). The semaphore actions are hidden in the implementation.

```
specification Readers_And_Writers_Impl [rb, re, wb, we] :noexit

behaviour
Impl [rb, re, wb, we]
where

process Impl [rb, re, wb, we] :noexit :=
    hide p, v in S [p, v] |[p, v]| (Proc [p, v, rb, re] ||| Proc [p, v, wb, we])
endproc

process S [p, v] :noexit :=
    p; v; S [p, v]
endproc

process Proc [p, v, b, e] :noexit :=
    p; b; e; v; Proc [p, v, b, e]
endproc
endspec
```

Figure 9.9: The Readers and Writers Implementation

The problem could potentially have more processes accessing the resource, but it is sufficient to show that the semaphore preserves mutual exclusion for the two process case.

9.3.2 Proving the Verification Requirement Holds

The aim of the verification is to show that the implementation of figure 9.9 satisfies the specification of figure 9.8, i.e. we have to prove the following conjecture in PAM:

```
    Spec = Impl
```

where = stands for weak bisimulation congruence. This was the relation used in [DIN91].

Some minor syntactic alterations are made to the LOTOS descriptions of figures 9.8 and 9.9 in order to conform with PAM syntax, also, gate parameters in the LOTOS description are implemented by relabelling in PAM. See appendix D.6 for the PAM input file.

The above conjecture can be shown by the axioms as described in appendix D.1, the relabelling axioms of appendix D.2 and the built-in unique fixpoint induction. As usual, most of the proof consists of unfolding process identifiers, expanding parallel statements, and folding process identifiers. Tactics are used to push **hide** and relabelling through a sequence of actions, and the PAM axioms of weak bisimulation congruence are used to remove occurrences of the silent action

i. In the following presentation of the proof we simplify process expressions by removing gate parameters. We also combine several simple (tedious) steps, but note the rules used in each step at the right hand side.

```
    Impl
≡wbc                                                        (unfold)
    hide p, v in S |[p, v]| ((Proc [rb/b]) [re/e] |||(Proc [wb/b]) [we/e])
≡wbc                                       (unfold Proc, REL tactic)
    hide p, v in S |[p, v]|
              (p; rb; re; v; (Proc [rb/b]) [re/e]
              ||| p; wb; we; v; (Proc [wb/b]) [we/e])

≡wbc                                           (unfold S, expansion)
    hide p, v in
          p; rb; re; v; (S |[p, v]| ((Proc [rb/b]) [re/e] ||| (Proc [wb/b]) [we/e]))
       [] p; wb; we; v; (S |[p, v]| ((Proc [rb/b]) [re/e] ||| (Proc [wb/b]) [we/e]))

≡wbc                                                  (HIDE tactic)
      i; rb; re; i; (hide p, v in
                     (S |[p, v]| ((Proc [rb/b]) [re/e] ||| (Proc [wb/b]) [we/e]))
    [] i; wb; we; i; (hide p, v in
                     (S |[p, v]| ((Proc [rb/b]) [re/e] ||| (Proc [wb/b]) [we/e]))
≡wbc                                                (i laws, fold)
    i; rb; re; Impl [] i; wb; we; Impl
```

By application of the built-in `ufi` function, PAM is able to conclude `Spec` $\equiv_{wbc}$ `Impl`.

Completing this example builds our confidence in the applicability of our proof technique. We continue with another example from the same paper.

9.4 A Nondeterministic Candy Machine

This problem is also taken from [DIN91] and is again an exercise in proving that two LOTOS descriptions of a system are equivalent, i.e. the *implementation* satisfies the *specification*. The specification gives the observable behaviour of the machine whereas the implementation builds the machine from simple components. We feel that the original specification presented was somewhat contrived, and therefore ours is slightly different.

9.4.1 The LOTOS Descriptions

The specification of a nondeterministic candy machine is given in figure 9.10. This machine takes a ten pence and returns either the message "try again", a ten pence, or a piece of candy. After this it behaves like a candy machine again.

An implementation of the Candy machine is given in figure 9.11. The implementation of the machine is the parallel composition of the processes `Slot`, `Fair` and `Turn`.

```
specification Candy_Spec [in10p, message, candy, out10p] :noexit

behaviour
Spec [in10p, message, candy, out10p]
where

process Spec [in10p, message, candy, out10p] :noexit :=
    in10p; (   i; message; Spec [in10p, message, candy, out10p]
            [] i; out10p; Spec [in10p, message, candy, out10p]
            [] i; candy; Spec [in10p, message, candy, out10p])
endproc
endspec
```

Figure 9.10: The Candy Machine Specification

```
specification Candy_Impl [in10p, message, candy, out10p] :noexit

behaviour
Candy [in10p, message, candy, out10p]
where

process Candy [in10p, message, candy, out10p] :noexit :=
    hide t25p, t10p in
        ((Slot [in10p, message, t10p, t25p]
        |[t25p, t10p]| Fair [t10p, t25p, candy, out10p])
        |[in10p, message, candy, out10p]| Turn [in10p, message, candy, out10p])
endproc


process Slot [in10p, message, out10p, out25p] :noexit :=
    in10p; (   (i; message; Slot [in10p, message, out10p, out25p])
            [] (i; out10p; Slot [in10p, message, out10p, out25p])
            [] (i; out25p; Slot [in10p, message, out10p, out25p]))
endproc

process Fair [in10p, in25p, candy, out10p] :noexit :=
    (   in25p; candy; Fair [in10p, in25p, candy, out10p]
     [] in10p; out10p; Fair [in10p, in25p, candy, out10p])
endproc

process Turn [in10p, message, candy, out10p] :noexit :=
    in10p; (   out10p; Turn [in10p, message, candy, out10p]
            [] candy; Turn [in10p, message, candy, out10p]
            [] message; Turn [in10p, message, candy, out10p])
endproc
endspec
```

Figure 9.11: The Candy Machine Implementation

The processes `Slot` and `Fair` are as given in [DIN91]. The machine `Slot` takes ten pence and nondeterministically returns either a message, ten pence or twenty-five pence. The machine `Fair` will return a candy if given twenty-five pence, but rejects ten pence. The object of the original exercise of [DIN91] was to study the process resulting from the parallel combination of `Slot` and `Fair`. The link between the two processes was established by joining the output of `Slot` to the input of `Fair` via the new gates `t25p` and `t10p` (through twenty-five pence and through ten pence).

Since there is no restriction on putting ten pence in before the result of the last ten pence is known, the specification derived in [DIN91] is rather complex and unintuitive. We have written a new, simpler, specification, as given in figure 9.10, and altered the original implementation by adding a new constraint in the form of `Turn`, which forbids putting in another ten pence until the result of the previous input is known.

9.4.2 Proving the Verification Requirement Holds

The aim of the verification is to show that the implementation of figure 9.11 satisfies the specification of figure 9.10, i.e. we have to prove the following conjecture in PAM:

```
Spec = Candy
```

where = stands for $\equiv_{wbc}$.

Some minor syntactic alterations are made to the LOTOS descriptions of figures 9.10 and 9.11 in order to conform with PAM syntax; see appendix D.7 for the PAM input file. Gate parameters in the LOTOS are implemented by means of relabelling in PAM.

The above conjecture can be shown to hold by the axioms as described in appendix D.1, the relabelling axioms of appendix D.2 and unique fixpoint induction. The proof below follows the format of the Readers and Writers proof of the previous section:

```
    Candy
```
$\equiv_{wbc}$ *(unfold)*
```
    hide t10p, t25p in (((Slot[t10p/out10p]) [t25p/out25p]
                         |[t10p, t25p]| (Fair[t10p/in10p]) [t25p/in25p])
                         |[in10p, message, out10p, candy]| Turn)
```
$\equiv_{wbc}$ *(unfold,* REL *tactic)*
```
    hide t10p,t25p in
        in10p; (   i; message; (Slot [t10p/out10p]) [t25p/out25p]
                [] i; t10p; (Slot [t10p/out10p]) [t25p/out25p]
                [] i; t25p; (Slot [t10p/out10p]) [t25p/out25p]))
      |[t10p, t25p]| ( t25p; candy; (Fair [t10p/in10p]) [t25p/in25p]
                     [] t10p; out10p; (Fair [t10p/in10p]) [t25p/in25p])
      |[in10p, message, out10p, candy]| in10p; ( out10p; Turn [] candy; Turn
                                               [] message; Turn)
```
$\equiv_{wbc}$ *(expansion)*

```
  hide t10p,t25p in
    in10p; (    i; message; (((Slot [t10p/out10p]) [t25p/out25p]
                              |[t10p, t25p]| (Fair [t10p/in10p]) [t25p/in25p])
                              |[in10p, message, out10p, candy]| Turn)
            [] i; t10p; out10p; (((Fair [t10p/in10p]) [t25p/in25p]
                              |[t10p, t25p]| (Slot [t10p/out10p]) [t25p/out25p])
                              |[in10p, message, out10p, candy]| Turn)
            [] i; t25p; candy; (((Fair [t10p/in10p]) [t25p/in25p]
                              |[t10p, t25p]| (Slot [t10p/out10p]) [t25p/out25p])
                              |[in10p, message, out10p, candy]| Turn))
```

$\equiv_{wbc}$ *(*HIDE *tactic)*

```
    in10p; (    i; message; (hide t10p,t25p in
                              ((Slot [t10p/out10p]) [t25p/out25p]
                              |[t10p, t25p]| (Fair [t10p/in10p]) [t25p/in25p])
                              |[in10p, message, out10p, candy]|Turn)))
            [] i; i; out10p; (hide t10p,t25p in
                              ((Fair [t10p/in10p]) [t25p/in25p]
                              |[t10p, t25p]| (Slot [t10p/out10p]) [t25p/out25p])
                              |[in10p, message, out10p, candy]| Turn)
            [] i; i; candy; (hide t10p,t25p in
                              ((Fair [t10p/in10p]) [t25p/in25p]
                              |[t10p, t25p]| (Slot [t10p/out10p]) [t25p/out25p])
                              |[in10p, message, out10p, candy]| Turn)
```

$\equiv_{wbc}$ *(*i *laws, fold)*

```
    in10p; (i; message; Candy [] i; out10p; Candy [] i; candy; Candy)
```

By using the built-in `ufi` function PAM is able to conclude `Spec` $\equiv_{wbc}$ `Candy`. This was the conclusion of [DIN91] on the original version of the candy machine.

9.5 The Scheduler

Our final example in this chapter is the well-known Scheduler problem of [Mil89b]. This example is often used as a benchmark test for proof systems. The example was also presented in [DIN89], which is our main source. In [DIN89] the problem was simplified from the n process case to the two process case.

Since the original example was presented in CCS some major changes to the process behaviours have been made in recasting the descriptions in Basic LOTOS. These changes relate to the different approaches to synchronisation of parallel actions in CCS and LOTOS, and also the use of hiding rather than restriction. The example is described in the next section.

9.5.1 The LOTOS Descriptions

Consider two processes, `C1` and `C2`, which each perform some activity. As with the readers and writers example, we do not model the actual activities, only the start and end of them. The actions `a1` and `b1` are the start and stop actions for `C1`, with `a2` and `b2` being the corresponding actions for C2. The scheduler must ensure that the start activity actions occur cyclically, i.e. if only a actions are observed, the system produces traces of the form (a1 a2)*. The end activity actions are ignored by the scheduler specification. The LOTOS description of this specification is given in figure 9.12.

```
specification Sched_Spec [a1, a2] :noexit

behaviour
Spec [a1, a2]
where

process Spec [a1, a2] :noexit :=
    a1; a2; Spec [a1, a2]
endproc
endspec
```

Figure 9.12: The Scheduler Specification

The implementation of the scheduler, given in LOTOS in figure 9.13, must then ensure that this specification is satisfied. This is achieved by using extra actions which model a baton being passed back and forth between C1 and C2, where possession of the baton indicates permission to start. The b actions (end activity) may occur in any order, in particular, the scheduler must not force an ordering on them as it does with the a actions.

Note that the translation from CCS to LOTOS is not trivial. For example, in the CCS implementation, the processes C1 and C2 are identical, modulo renaming of events, rather like our C1' and C2. However, these processes need a prompt from another source to start them off, since initially both are waiting for their turn; for the "baton" to be passed to them. In the CCS description, this prompting is performed by an extra process which performs a g1 action and then stops; the process 0 can then be eliminated by using the laws of CCS to rewrite the expression. However, in LOTOS, due to the different nature of synchronisation, there is no corresponding law. To solve this problem, we have two versions of C1. Initially C1 can start straight away, but control is then passed to C1', which must wait for the g1 action before it may proceed.

9.5.2 Proving the Verification Requirement Holds

The aim of the verification is to show that the implementation of figure 9.13 satisfies the specification of figure 9.12, i.e. we have to prove the following conjecture in PAM:

```
    Spec = hide b1, b2 in Sch
```

where = stands for $\equiv_{wbc}$.

```
specification Sched_Impl [a1, a2, b1, b2] :noexit

behaviour
hide b1, b2 in Sch [a1, a2, b1, b2]
where

process Sch [a1, a2, b1, b2] :noexit :=
    hide g1, g2 in (C1 [a1, b1, g1, g2] |[g1, g2]| C2 [a2, b2, g1, g2])
endproc

process C1 [a1, b1, g1, g2] :noexit :=
    a1; ((b1; g2; C1' [a1, b1, g1, g2]) [] (g2; b1; C1' [a1, b1, g1, g2]))
endproc

process C1' [a1, b1, g1, g2] :noexit :=
    g1; a1; ((b1; g2; C1' [a1, b1, g1, g2]) [] (g2; b1; C1' [a1, b1, g1, g2]))
endproc

process C2 [a2, b2, g1, g2] :noexit :=
    g2; a2; ((b2; g1; C2 [a2, b2, g1, g2]) [] (g1; b2; C2 [a2, b2, g1, g2]))
endproc
endspec
```

Figure 9.13: The Scheduler Implementation

Some minor syntactic alterations are made to the above LOTOS descriptions in order to conform with PAM syntax. See appendix D.8 for the PAM input file.

In this case the PAM axioms as described in section D.1 and unique fixpoint induction are not sufficient to complete the proof; two other laws (derived from those in the LOTOS standard and given as part of the supplementary set of PAM axioms in appendix D.2) are also required.

```
hide A in hide A' in B    = hide A' in hide A in B
hide A in (B1 |[A']| B2) = ((hide A in B1) |[A']| (hide A in B2))
                            if {} eq (A inter A')
```

These allow occurrences of **hide** to be flipped, and **hide** to be distributed through parallel statements (as long as the hidden actions are not required for synchronisation). Again, the proof follows the format of the Readers and Writers proof.

```
    hide b1, b2 in Sch
≡wbc                                                        (unfold)
    hide b1, b2 in hide g1, g2 in (C1 |[g1, g2]| C2)
≡wbc                                         (flip and distribute hide)
    hide g1, g2 in ((hide b1, b2 in C1) |[g1, g2]| (hide b1, b2 in C2))
≡wbc                                                        (unfold)
    hide g1, g2 in
        ((hide b1, b2 in (a1; ((b1; g2; C1') [] (g2; b1; C1'))))
        |[g1, g2]| (hide b1, b2 in (g2; a2; ((b2; g1; C2) [] (g1; b2; C2)))))
≡wbc                                                   (HIDE tactic)
```

```
    hide g1, g2 in
        ((a1; ((i; g2; hide b1, b2 in C1') [] (g2; i; hide b1, b2 in C1')))
        |[g1, g2]|
        (g2; a2; ((i; g1; hide b1, b2 in C2) [] (g1; i; hide b1, b2 in C2))))
≡wbc                                                          (i laws)
    hide g1, g2 in
        ((a1; g2; hide b1, b2 in C1') |[g1, g2]| (g2; a2; g1; hide b1, b2 in C2))
≡wbc                                                          (expansion)
    hide g1, g2 in a1; g2; a2;
        ((hide b1, b2 in C1') |[g1, g2]| (g1; hide b1, b2 in C2))
≡wbc                                                          (unfold C1')
    hide g1, g2 in a1; g2; a2;
        ((hide b1, b2 in (g1; a1; ((b1; g2; C1') [] (g2; b1; C1'))))
        |[g1, g2]| (g1; hide b1, b2 in C2))
≡wbc                              (push hide through g1, expansion, fold)
    hide g1, g2 in a1; g2; a2; g1;
        ((hide b1, b2 in C1) |[g1, g2]| (hide b1, b2 in C2))
≡wbc                                                (HIDE tactic, i laws)
    a1; a2; hide g1, g2 in ((hide b1, b2 in C1) |[g1, g2]| (hide b1, b2 in C2))
≡wbc                                              (reverse 2nd step, fold)
    a1; a2; (hide b1, b2 in Sch)
```

By using the built-in `ufi` function PAM is able to conclude `Spec` $\equiv_{wbc}$ `Impl`.

9.6 Summary

In this chapter we have presented five examples of the use of the proof system developed in the previous chapter: the login case study, now with recursive processes, one more major example (a simple radiation machine) and three minor examples (the readers and writers problem, the nondeterministic candy machine, and the scheduler). All examples were drawn from the papers of other authors, together with the verification requirement of showing the implementation satisfies the specification. The main aim of this chapter was to show that the proof system developed on PAM is applicable to other examples; this has been successful. It is worth noting that most of the proofs presented here were completed very quickly, also showing that the proof system is easy to use, and that this method of proof is practical. Difficulties only occur when new approaches to the proof are required, as in the radiation machine example of section 9.2.

With the exception of the radiation machine, no new results were presented here; we merely use the examples of others to show that our proof system is applicable more generally. In the radiation machine example, we obtained a result (that a particular version of the system was safe), where the original presentation failed. Conclusions specific to this example were discussed in section 9.2.6.

Chapter 10

Full LOTOS

Up to this point, verification of only a subset of the LOTOS language, namely the process algebra Basic LOTOS, has been considered; however, full LOTOS incorporates an abstract data type (adt) language, also known as ACT ONE [EM85]. The addition of ACT ONE to the language enables the specification of guards which may further determine the flow of control within a process, and also data values which may be passed between processes. Since, in practical applications of LOTOS, specifiers do not restrict themselves only to Basic LOTOS, we must rethink the approach to verification presented so far to incorporate specifications including ACT ONE data types.

In the literature, most authors have done as we have: concentrated on verification of Basic LOTOS, ignoring data. For example, in the LITE toolset [LITE] the tools which operate over full LOTOS are the testing and simulation tools, whereas the verification tools, i.e. those which evaluate equivalences between processes and perform model checking, deal only with Basic LOTOS. Similarly, there are also tools which deal only with ACT ONE, ignoring the process algebra part of LOTOS. Although there is work on the verification of adts, we prefer to concentrate on the verification of *full* LOTOS, i.e. considering how the addition of adts to LOTOS affects the verification approach we have adopted up to now. Obviously, since data can affect the flow of control within a process and hence its behaviour, ignoring the addition of ACT ONE constructs to a specification may have serious implications for any correctness results.

We begin by introducing the data type part of LOTOS, noting that some features of Basic LOTOS need to be modified to include data types. To introduce the problems that may arise with verification of full LOTOS specifications we give three LOTOS descriptions of the well-known adt example, the *stack*. Each description is written with differing emphasis on the process part and the adt part. These descriptions were first presented by Gotzhein in [Got87]. In the original paper the author claimed that the descriptions were equivalent, but was unable to prove this. We try to prove equivalence using the satisfaction approach to verification, but with limited success. We go on to consider approaches by other authors to the problem of verification of full LOTOS. To our knowledge there are only a few works in this area: [Led87], [Bri92, BK91] and [Bol92]. We discuss the strengths and weaknesses of these approaches. The latter two approaches seem quite promising, and we illustrate their use via application to examples of our own: the stack of the early part of the chapter, and the simple radiation machine example of chapter 9 respectively. Finally, we conclude that the most suitable approach to verification is a composite one.

10.1 ACT ONE and LOTOS

The syntax and semantics of full LOTOS is presented in appendix B. Here we present a brief overview of the main features of ACT ONE and alterations to Basic LOTOS which give full LOTOS.

10.1.1 ACT ONE Syntax

Abstract data types in LOTOS are specified in an algebraic fashion. A basic specification consists of the name of the type, a list of *sorts* used by the type, declarations of *operators* over the type, and *equations* specifying the behaviour of those operations. There are no built in types, although there is a standard library of commonly used types. One of the standard library types is *Boolean.* A portion of the definition is given in figure 10.1 to illustrate the ACT ONE style of definition. The full definition of *Boolean* also includes the operators `=>`, `iff`, `ne`, `eq` and their associated equations.

```
type    Boolean is
sorts   Bool
opns    true, false   :  -> Bool
        not           :  Bool -> Bool
        _and_, _or_   :  Bool, Bool -> Bool

eqns    forall x : Bool
        ofsort Bool
        not(true)   = false;
        not(false)  = true;
        x and true  = x;
        x and false = false;
        x or true   = true;
        x or false  = x;
endtype
```

Figure 10.1: The Library Type `Boolean`

In ACT ONE, complex specifications may be built up from simpler specifications in a hierarchical manner; parameterisation of specifications also aids specification of large systems.

10.1.2 Adding ACT ONE to Basic LOTOS

To incorporate ACT ONE into LOTOS, much more needs to be done than simply providing the ability to specify data types; some modification to the existing process algebra, to allow manipulation of the data, is also required.

- The actions of Basic LOTOS are enhanced by allowing values to be passed between processes at the gates. An action in full LOTOS is a gate plus an *event offer*, i.e. a value passing/receiving construct. There are two types of event offers: variable offering and value offering, which can be thought of as input and output. For example,

  ```
  wire?x:Nat
  ```

  receives a value `x` of type `Nat` at the gate `wire`, and

  ```
  wire!1
  ```

  outputs the value one at the gate `wire`. There can be more than one value passed at any gate, and there are no restrictions on direction (unlike CSP, where it is recommended, although not enforced, that channels are unidirectional). For example, in LOTOS the following is also an action:

  ```
  wire?x:Int !3 ?y:Char
  ```

  receiving values in `x` and `y`, and outputting the value 3, all at the gate `wire`.

  As before, actions synchronise by matching gate names, but now values/variables must also be matched. The standard form of value passing is to match a value offering to a variable offering, but, in addition, offerings of the same type may synchronise, i.e. value with value and variable with variable. This latter form essentially restricts the possible range of the value received/sent, bearing in mind that LOTOS has multi-way synchronisation, and that `wire?x:Nat`, for example, is really a shorthand for a *set* of choices, i.e. `wire?0 [] wire?1 [] ....` The synchronisation mechanism allows the actions of one process to be restricted by another (or several others) and is utilised by the constraint-oriented style of specification.

- Processes may be prefixed by *guards*, e.g.

  ```
  [x > 0] -> P
  ```

  which means `P` is only performed if `x` has value greater than zero, i.e. if the guard evaluates to true. Typically several such statements are composed using the choice operator, giving the effect of the case statement of programming (although the guards may not be mutually exclusive, in which case the choice is made nondeterministically).

- A process may produce values on successful termination. Whereas in Basic LOTOS successful termination is denoted by the δ action, in full LOTOS successful termination is denoted by $\delta\ v_1 \ldots v_n$, where $v_1 \ldots v_n$ is a string of data values.

  In Basic LOTOS the functionalities **exit** and **noexit** are added to processes to indicate whether or not they terminate. Given the above modification of successful termination, data types can also be added to **exit** functionality to indicate the types of values produced when the process terminates.

  This addition to functionality can be used in an extension of sequential composition of processes to pipeline results from one process to another. For example,

  ```
  P >> accept n:Nat in Q
  ```

  Here P will terminate with, for example, $\delta\,5$, and Q will use the value of n, instantiated by the value 5, in its body. A special value is **any** which can be used to give a parameter any value from the specified type. This is required because in a parallel composition, the exit values of a process must match; use of **any** gives flexibility.

- Data values may be present as parameters to processes, just as gate names may be used as parameters in Basic LOTOS.

Note that, according to the full LOTOS syntax, a specification may *optionally* have data type definitions, whereas it *must* have process definitions. Within these constraints, the balance between the two components may be altered, giving, for example, a specification which has no data types at all, or a specification which has lots of data types, but only a minimal process providing an interface to the data types. The stack descriptions of section 10.2 demonstrate this ability to describe the same object in totally different ways while staying within the full LOTOS language.

10.1.3 Full LOTOS Semantics

Having introduced the syntax of full LOTOS, it is important to discuss how the addition of data types to the language affects the semantics. The semantics of an ACT ONE type is a multi-sorted algebra composed of (one or more) carriers of data elements, and a set of operations (total functions) over the carriers. ACT ONE data types are given *initial algebra semantics.* An algebra is initial iff there exists a homomorphism between it and any other algebra which gives the same properties for the signature. The formal definition of the ACT ONE syntax and semantics is discussed more fully in appendix B.

The semantics of a full LOTOS process is given by a *structured labelled transition system.* Instead of being labelled by a gate name, a transition is labelled by a gate name and a string of values from the ACT ONE algebra. For example, transitions are of the form

$$p \xrightarrow{gw} q$$

where $g \in$ *Gates* as before, and $w = v_1 \ldots v_n, v_i \in$ *Value*. Obviously the interpretation of *Value* will differ from specification to specification, depending on the data types used. No variables are included in w, the semantics of full LOTOS ensures that the variables of the specification are all instantiated, i.e. only ground terms appear in labelled transition systems and uninstantiated terms have no meaning.

Whereas in Basic LOTOS the set *Act* was composed of gate names and the internal action, now actions are of the form gw as above. All definitions of equivalence/congruence/preorder relations are modified to use the new type of actions simply by using the new definition of *Act.* For example, for actions to match in a bisimulation relation, the gate names *and* the data values of the transition must match.

Bearing in mind that the semantics of every LOTOS specification is a structured labelled transition system, it easy to see now why the data component of the specification is optional while the process part is compulsory. Since the equivalences are based on observable behaviour, the specifications must have at least *some* behaviour to observe. Obviously if a specification consists of only data types then there will be no transitions; the process part is equivalent to **stop**. As with many other formalisms, this is the result of a design decision by the creators of the language, and is not the only way to describe objects.

Now we have discussed how a data type may be used in a LOTOS specification, and how the meaning of that specification is given, we can go on to consider the verification of full LOTOS specifications.

10.2 Three Views of a Stack

Early in the development of LOTOS, [Got87] considered the problems of verification of full LOTOS specifications. Four descriptions of the same example, the stack, are presented in [Got87], each with a different emphasis on the process algebra or the abstract data type parts of LOTOS. Although the descriptions look quite different, the author claims they all represent the same object, and are in fact weak bisimulation congruent. No proof is presented to support this conjecture, although the author used testing and simulation tools to gain confidence in his claim.

In order to gain intuition about what adjustments to the verification process may be necessary when a full LOTOS specification is under consideration, we re-examine three of the four stack descriptions of [Got87] with a view to *formally* proving their equivalence. We have attempted to stick as closely as possible to the original descriptions, making only essential changes. Most changes were the result of what were assumed to be typographical errors; however, we did also remove the *error* values of the abstract data type as these were not necessary and in fact created some problems. The proofs are completed by hand, using a combination of equational reasoning and bisimulation techniques.

10.2.1 The ACT ONE Stack

We begin by giving a description of the stack data type in ACT ONE; see figure 10.2. This forms the basis for the first LOTOS stack, `Stack_1`, in the next section, but may also be viewed as the conceptual model of behaviour we have in mind when giving the process algebra style descriptions of `Stack_2` and `Stack_3`. Note that the description in figure 10.2 does not constitute a LOTOS specification itself, as it has no process algebra behaviour.

```
type Element is (* ACT ONE definition *) endtype

type Stack_Type(Data) is Boolean
  formal_sorts Data
  sorts        Stack_Type
  opns         New :  -> Stack_Type
               Push :  Stack_Type × Data -> Stack_Type
               Pop :  Stack_Type -> Stack_Type
               Top :  Stack_Type -> Data
               Empty :  Stack_Type -> Bool
  eqns         forall s:Stack_Type, d:Element
               ofsort Bool
               Empty (New) = true
               Empty (Push (s, d)) = false
               ofsort Data
               Top (Push (s, d)) = d
               ofsort Stack_Type
               Pop (Push (s, d)) = s
endtype

type Stack_Type (Element) is
     Stack_Type (Data) actualizedby Element
          using sortname Element for Data
endtype
```

Figure 10.2: Abstract Data Type: Stack

In the ACT ONE stack the usual operations are defined: `top`, `push`, `pop` and `empty`. Error values are ignored as the process guards provide the means to avoid constructing data values which are undefined, such as `Top (New)`.

This specification also demonstrates the parameterisation mechanism used in ACT ONE whereby the stack type is described independently of the type of the data elements. These are instantiated later. Comments are bracketed by (* and *). Note that our use of comments to indicate portions of the specification which have been omitted is not correct concrete LOTOS, but is rather a convenient method of abbreviating the presentation.

In the proof of section 10.2.4 that `Stack_1` and `Stack_2` are equivalent we will find it convenient to be able to refer to the bottom part of the stack, i.e. we define a function `rest` which takes a stack and returns the stack obtained by removing the top element.

```
rest :  Stack_Type -> Stack_Type
rest (Push (s, d)) = s
```

Note that `rest (New)` is undefined; we will never apply `rest` to `New`.

10.2.2 The First LOTOS Stack

The first stack will be described with the emphasis on ACT ONE. The ACT ONE description of the stack is as given in figure 10.2. The process part given in figure 10.3 supplies the external interface to the abstract data type stack.

```
specification Stack_Data_Type_1

library Boolean endlib
type Element is (* ACT ONE definition *) endtype
type Stack_Type (Data) is (* ACT ONE definition in figure 10.2 *) endtype
type Stack_Type (Element) is (* ACT ONE definition in figure 10.2 *) endtype

behaviour
process Stack_1 [push, pop, top, empty] :noexit :=
        push?x:Element; Used_Stack_1 [push, pop, top, empty] (Push (New, x))
    [] empty!Empty (New); Used_Stack_1 [push, pop, top, empty] (New)
where
process Used_Stack_1 [push, pop, top, empty] (s:Stack_Type) :noexit :=
        push?x:Element; Used_Stack_1 [push, pop, top, empty] (Push (s, x))
    [] [not (Empty (s))] -> pop; Used_Stack_1 [push, pop, top, empty] (Pop (s))
    [] [not (Empty (s))] -> top!Top (s); Used_Stack_1 [push, pop, top, empty] (s)
    [] empty!Empty (s); Used_Stack_1 [push, pop, top, empty] (s)
    endproc
endproc
endspec
```

Figure 10.3: The First Stack

This stack is referred to as Version 1 in [Got87].

10.2.3 The Second LOTOS Stack

Here the data type is coded within the process part, i.e. the stack mechanism is implemented by the processes, rather than being described by adt equations. The specification still retains approximately the same structure as before, the main differences being the substitution of the ACT ONE terms in the process by constants, the change in the parameter to `Used_Stack_2` from the whole stack to just one element of the stack, the *top* element, and the use of $\gg$ to connect separate cells of the stack, i.e. instances of `Used_Stack_2 (x)`. Note that the process `Used_Stack_2 (x)` always deals with a stack with something in it, whereas the process `Used_Stack_1 (s)` may also have to deal with an empty stack.

```
specification Stack_Data_Type_2

library Boolean endlib
type Element is (* ACT ONE definition *) endtype

behaviour
process Stack_2 [push, pop, top, empty] :noexit :=
    (   push?x:Element; Used_Stack_2 [push, pop, top, empty] (x)
    []  empty!true; exit)
        >> Stack_2 [push, pop, top, empty]
where
process Used_Stack_2 [push, pop, top, empty] (x:Element) :exit :=
        push?y:Element; Used_Stack_2 [push, pop, top, empty] (y)
            >> Used_Stack_2 [push, pop, top, empty] (x)
    []  pop; exit
    []  top!x; Used_Stack_2 [push, pop, top, empty] (x)
    []  empty!false; Used_Stack_2 [push, pop, top, empty] (x)
    endproc
endproc
endspec
```

Figure 10.4: The Second Stack

This stack is referred to as Version 3 in [Got87].

10.2.4 Proving Stack One Equivalent to Stack Two

Having described our stacks, how can it be verified that the two specifications, `Stack_Data_Type_1` and `Stack_Data_Type_2`, denote the same object?

In the previous examples, where we were restricted to Basic LOTOS, such verification was achieved by proving that the process expressions of the specifications were related by one of the equivalence, congruence or preorder relations. For full LOTOS we have the added complication of expressions relating to data. The definitions of the relations in previous chapters are easily modified to take account of the shift from gate names to structured actions, as mentioned in section 10.1.3. We can therefore still attempt to prove that Stack_1 and Stack_2 denote the same object by showing that the process expression corresponding to Stack_1 is equivalent in some sense to the one corresponding to Stack_2, although the proof technique for this is not as well researched as that for Basic LOTOS expressions. Here we identify the top level process Stack_1 with the specification Stack_Data_Type_1 by an abuse of notation, and similarly for Stack_2. The relation chosen for the comparison is weak bisimulation congruence as this was the relation used in [Got87].

The proof method we use is a combination of equational reasoning and construction of a bisimulation relation; this proof method is employed in [Mil89b] for CCS with simple data types. We also note that a sketch of an apparently similar style of proof for equivalence of two queue specifications described in full LOTOS appears in [Bri88b]. At the time our proof was completed we were unaware of this sketch. To our knowledge, no such proof has ever been fully worked out for full LOTOS specifications.

The proofs are carried out by hand, partly to gain intuition as to the proof process, and partly because no tools currently exist which can prove equivalences between full LOTOS specifications. As observed in the introduction, there are several tools which operate over full LOTOS, but these are simulators or testers, and our aim is to produce a formal, mathematical proof of correctness. It is possible to use a tool to produce a labelled transition system which can then be fed into a tool such as the Concurrency Workbench; this technique is used in [EFJ90]. However, this process requires intervention by the user to rename the actions to make them acceptable for input to the Concurrency Workbench, and on even a small example like the stack, this seems more tedious than carrying out the proof by hand in the first place.

The tool used in the last two sections, PAM v1.0, cannot be used for the proofs as PAM only deals with basic process algebras. However, a new version called VPAM is under development in which processes can have data type parameters, and we hope to use this tool at some point in the future. Addition of data types would be straightforward in a term rewriting system framework, but we are reluctant to go back to using RRL, as this would mean losing the ability to define recursive processes. It might be possible to use the data value parameters of the processes to control the rewriting, i.e. primitive recursion, but the stacks in this chapter are unbounded data types, therefore the recursion would be unbounded for open terms. For the time being we are resigned to carrying out the proof by hand.

Formally, we wish to prove the following theorem holds:

Theorem 3 Stack_1 $\equiv_{wbc}$ Stack_2

Proof. Since full LOTOS is rather verbose, the first step in the proof is to simplify the processes as much as possible. This is done in two ways: syntactically and semantically. The former includes the omission of gate parameters to the processes because the formal and actual parameters of the processes are the same in both descriptions, and omission of types of values because these may be deduced from the context. Semantic manipulations include application of the laws of weak bisimulation congruence to the behaviour expressions, and application of the equations of the data type to the abstract data type expressions. The descriptions we give below are therefore LOTOS-like, but do not conform to the concrete syntax of LOTOS.

We identify significant states in the processes `Stack_1` and `Stack_2` and present their simplified forms.

```
Stack_1 =      push?x; Used_Stack_1 (Push (New, x))
          []   empty!true; Used_Stack_1 (New)

Stack_2 = (     push?x; Used_Stack_2 (x)
          []    empty!true; exit) >> Stack_2
        ≡wbc    push?x; (Used_Stack_2 (x) >> Stack_2)
          []   empty!true; Stack_2
```

We now do the same for `Used_Stack_1` and `Used_Stack_2`.

`Used_Stack_1 (New)` and `Used_Stack_1 (Push (s, x))` are considered as separate cases. There is no need to do this for `Used_Stack_2` as `Used_Stack_2 (New)` is not a state of `Used_Stack_2`; it is blocked by the process algebra behaviour. In `Used_Stack_1 (New)` the expression `Empty(New)` is replaced by `true`, and two of the four choices are removed because their guards evaluate to `false`.

```
Used_Stack_1 (New)          =      push?x; Used_Stack_1 (Push (New, x))
                                []  empty!true; Used_Stack_1 (New)

Used_Stack_1 (Push (s, x)) =      push?y; Used_Stack_1 (Push (Push (s, x), y))
                                []  pop; Used_Stack_1 (s)
                                []  top!x; Used_Stack_1 (Push (s, x))
                                []  empty!false; Used_Stack_1 (Push (s, x))

Used_Stack_2 (x)            =      push?y; Used_Stack_2 (y) >> Used_Stack_2 (x)
                                []  pop; exit
                                []  top!x; Used_Stack_2 (x)
                                []  empty!false; Used_Stack_2 (x)
```

It is illuminating to look at the labelled transition systems for these processes, given in figure 10.5. We can see from these diagrams that `Stack_1` and `Stack_2` look very similar, except that the state `Stack_1` is visited only once, at the start of execution, `Used_Stack_1` being used in its place in the remainder of the execution. As a first step towards proving `Stack_1` $\equiv_{wbc}$ `Stack_2` we show `Stack_1` $\equiv_{wbc}$ `Used_Stack_1 (New)`. The proof proceeds by unique fixed point induction: if `Used_Stack_1 (New)` can be substituted for `Stack_1` in the defining equations of `Stack_1` with no change in the behaviour, then `Used_Stack_1 (New)` is equivalent to `Stack_1`. This is obvious from the equations above.

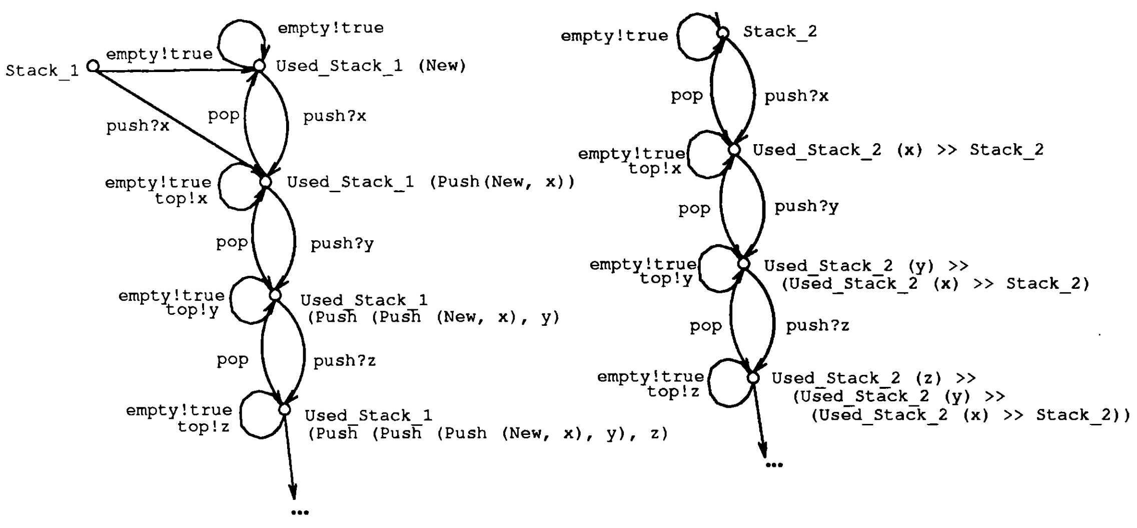

Stack_1
empty!true
push?x
empty!true
Used_Stack_1 (New)
pop
push?x
empty!true top!x
Used_Stack_1 (Push(New, x))
pop
push?y
empty!true top!y
Used_Stack_1 (Push (Push (New, x), y)
pop
push?z
empty!true top!z
Used_Stack_1 (Push (Push (Push (New, x), y), z)
...
empty!true
Stack_2
pop
push?x
empty!true top!x
Used_Stack_2 (x) >> Stack_2
pop
push?y
empty!true top!y
Used_Stack_2 (y) >> (Used_Stack_2 (x) >> Stack_2)
pop
push?z
empty!true top!z
Used_Stack_2 (z) >> (Used_Stack_2 (y) >> (Used_Stack_2 (x) >> Stack_2))
...

Figure 10.5: **Stack_1** and **Stack_2**

The next stage of the proof is to show that **Used_Stack_1** $\equiv_{wbc}$ **Stack_2**. We construct a relation, $\mathcal{R}$, which includes pairs of process expressions (which represent states) which we claim are equivalent; we must then show that $\mathcal{R}$ is a weak bisimulation. Using the process expressions in figure 10.5 as a guide, we give the first few pairs of $\mathcal{R}$ below. Note that since >> is associative, we have omitted brackets.

$\mathcal{R}$ = {(**Used_Stack_1 (New), Stack_2**),
  (**Used_Stack_1 (Push (New, x)), Used_Stack_2 (x) >> Stack_2**),
  (**Used_Stack_1 (Push (Push (New, x), y)),**
    Used_Stack_2 (y) >> Used_Stack_2 (x) >> Stack_2),
  (**Used_Stack_1 (Push (Push (Push (New, x), y), z)),**
    Used_Stack_2 (z) >> Used_Stack_2 (y) >> Used_Stack_2 (x) >> Stack_2),
  ...}

Since the stacks are unbounded $\mathcal{R}$ will contain an infinite number of pairs, so to express $\mathcal{R}$ finitely we have to find a generalisation of the pairs in $\mathcal{R}$. We define functions which "translate" from states in **Stack_1** to states in **Stack_2**, using the adt stack given in figure 10.2 as an intermediate step. These functions are merely syntactic manipulations of process expressions.

Given a stack **s** of **Stack_Type**, we assume that **s** is in canonical form, i.e. composed only of applications of **Push** and **New** (applications of **Pop** can be removed by the equations of **Stack_Type**, and **Top** gives values from **Data**, rather than **Stack_Type**). We define f and g which take **s** of **Stack_Type** and return a process expression, i.e. a member of $\mathcal{P}$, in **Stack_1** or **Stack_2** respectively.

f	: **Stack_Type** $\rightarrow \mathcal{P}$	g	: **Stack_Type** $\rightarrow \mathcal{P}$
f(**s**)	= **Used_Stack_1 (s)**	g(**Push (s, d)**)	= **Used_Stack_2 (d) >>** g(**s**)
		g(**New**)	= **Stack_2**

Both functions are bijective with respect to syntactic equality, since both processes take a stack

s, which is in canonical form, and merely perform syntactic manipulations of the representation of that stack to give a process expression. For the function f this manipulation is trivial, prefixing the stack s by `Used_Stack_1`, therefore each stack s has a unique representation in terms of process expressions relating to states in the labelled transition system of `Stack_1`. The function g performs a more complicated manipulation, but it is clear that no elements of the stack s are lost or duplicated in the transformation into process expressions, and that each stack s is related to precisely one process expression relating to states in the labelled transition system of `Stack_2`.

Given these definitions, we now define $\mathcal{R}$ by:

$$\mathcal{R} = \{(f(\mathtt{s}), g(\mathtt{s})) \mid \forall \mathtt{s}.\mathtt{s} : \mathtt{Stack\_Type}\}$$

Having now defined $\mathcal{R}$ so that each pair consists of two states which we claim are bisimilar, we have to prove that $\mathcal{R}$ is a weak bisimulation. We proceed by showing that for any pair in $\mathcal{R}$, the definition of weak bisimulation is satisfied (see definition 4 of chapter 3 for the formal definition), i.e. for each pair, if one state can perform some transition, then the other state must also be able to perform that transition and the pair of states resulting from these transitions is also in $\mathcal{R}$. We consider two cases of pairs: $\mathtt{s} = \mathtt{New}$ and $\mathtt{s} \neq \mathtt{New}$.

Case: $s = \mathtt{New}$. The pair under consideration is $(f(\mathtt{New}), g(\mathtt{New}))$, i.e. `(Used_Stack_1 (New), Stack_2)`. From the defining equation of `Used_Stack_1 (New)` two transitions are possible: `push?x` and `empty!true`.

1. `push?x` can also be performed by `Stack_2`. The resultant pair of states is

   `(Used_Stack_1 (Push (New, x)), Used_Stack_2 (x) >> Stack_2)`

   By the definition of f and g this pair of states can be expressed as $(f(\mathtt{s}'), g(\mathtt{s}'))$, where $\mathtt{s}' = $ `Push (New, x)`, which is in $\mathcal{R}$.

2. `empty!true` can also be performed by `Stack_2`. The resultant pair is

   `(Used_Stack_1 (New), Stack_2)`

   which we know to be in $\mathcal{R}$ since this is the pair originally under consideration.

The argument is similar for the second clause of the bisimulation definition.

Case: $s \neq \mathtt{New}$. Here we proceed much as in the first case, except this time we choose an arbitrary pair from $\mathcal{R}$ and therefore an arbitrary underlying stack `s`. Let `s = Push(...(Push(New,` $\mathtt{x}_1$`), ...),` $\mathtt{x}_n$`)` for some $n \geq 1$. The pair in $\mathcal{R}$ under consideration is $(f(\mathtt{s}), g(\mathtt{s}))$, i.e.

(`Used_Stack_1 (s)`, P_n) where P_k is defined to be (`Used_Stack_2 (x`$_k$`)` >> P_{k-1}) for $k > 0$ and `Stack_2` for $k = 0$.

Four actions are possible:

1. `push?y`. After `push?y` the state pair evolves to the result pair

   (`Used_Stack_1 (Push (s, y))`, `Used_Stack_2 (y)` >> P_n)

   By using the definition of f and g and the states of this pair we can see the underlying stack is `s'` `= Push(s, y)`, allowing this pair to be alternatively expressed as (f(s'), g(s')), which is therefore in $\mathcal{R}$.

2. pop. To make the calculation of the resultant pair easier, we first partially unfold the definitions of f and g in the original state pair giving

   (`Used_Stack_1 (Push (rest(s), x`$_n$`)`, `Used_Stack_2 (x`$_n$`)` >> P_{n-1})

   where `rest(s)` is as defined in section 10.2.1. The pair which results after a **pop** action is

   (`Used_Stack_1` (`rest`(s)), **exit** >> P_{n-1})

   The right hand element of the pair, **exit** >> P_{n-1}, can be rewritten using the laws of weak bisimulation congruence to give **i**; P_{n-1}, giving the pair

   (`Used_Stack_1` (`rest`(s)), **i**; P_{n-1})

   Since we know that this is not the pair at the root of the labelled transition system, we can use weak bisimulation *equivalence* laws to remove the occurrence of **i**, giving

   (`Used_Stack_1` (`rest`(s)), P_{n-1})

   which can be expressed as (f(`rest`(s)), g(`rest`(s))), which is in $\mathcal{R}$.

3. `top!`x_n. This transition starts and ends in the same state, so the resultant pair is just the same as the original state, i.e.

   (`Used_Stack_1 (s)`, P_n)

   which is in $\mathcal{R}$.

4. `empty!false`. As with `top!`x_n the resultant pair is

   (`Used_Stack_1 (s)`, P_n)

which is in $\mathcal{R}$.

Again, the argument is similar for the second clause.

Since the pair in the second case above was chosen arbitrarily, it represents all pairs in $\mathcal{R}$ except the pair (f(New), g(New)), which was considered in the first case. The two cases above therefore cover all possible forms of pairs in $\mathcal{R}$, and hence $\mathcal{R}$ satisfies the definition of weak bisimulation.

Since $\mathcal{R}$ is a weak bisimulation, any pair of processes in $\mathcal{R}$ are bisimilar. In particular, the pair (Used_Stack_1 (New), Stack_2) belongs to $\mathcal{R}$, and hence Used_Stack_1 (New) $\approx_{wbe}$ Stack_2. Since the first branches of the labelled transition systems involved no internal actions, this relation is also a congruence, see the definition of observation congruence (weak bisimulation congruence), definition 4, section 3.4.3. ∎

Having successfully shown that Stack_1 and Stack_2 are weak bisimulation congruent using a combination of equational reasoning and construction of a bisimulation relation, we now consider a third description of the stack.

10.2.5 The Third LOTOS Stack

Although described using only process algebra and simple data types, Stack_2 retained some of the structure and flavour of Stack_1. In the third stack specification, shown in figure 10.6, the adt style is left behind, concentrating on process algebra to give a more operational definition of the stack.

Similarly to Stack_2, the process Used_Stack_3 is parameterised on the top element of the stack; the rest of the stack is modelled by an auxiliary data structure made up of linked cells, each cell containing one element of the stack.

This description is referred to as version 4 in [Got87].

10.2.6 Proving Stack One Equivalent to Stack Three

In Stack_2, each instantiation of Used_Stack_2 only has information about *one* element of the stack; as soon as that element is popped from the top of the stack, the process Used_Stack_2 exits and control is taken up by the instantiation of Used_Stack_2 modelling the next element down in the stack. In this procedure, it is only because we know how control is passed from one process to another via the >> operator that we can deduce that Stack_2 behaves as expected.

In Stack_3 we have a similar situation; this time the Used_Stack_3 process behaves like the top element of the stack with a pointer to the rest of the stack. Although Used_Stack_3 can perform the stack events of push, pop, etc., it is really only an interface to the stack-like structure of linked instantiations of Cell. After a pop action, Used_Stack_3 obtains the new top value from the Cell structure.

```
specification Stack_Data_Type_3

library Boolean endlib
type Element is (* ACT ONE definition *) endtype

behaviour
process Stack_3 [push, pop, top, empty] :noexit :=
(New_Stack [push, empty]
    >> accept x:Element, n:Int in
         (hide up_1, up_2, down_1, down_2 in
            Used_Stack_3 [push, pop, top, empty, up_1, up_2] (x, n)
            |[up_1, up_2]| Cell [up_1, up_2, down_1, down_2] (x, 0)))
    >> Stack_3 [push, pop, top, empty]

where
process New_Stack [push, empty] :exit (Element, Int) :=
    push?x:Element; exit (x, 1)
[] empty!true; New_Stack [push, empty]
endproc

process Used_Stack_3 [push, pop, top, empty, up_1, up_2] (x:Element, n:Int) :exit :=
    push?y:Element; up_1!x; Used_Stack_3 [push, pop, top, empty, up_1, up_2] (y, n+1)
[] pop; up_2?z:Element;
       ([n = 1] -> exit
       [] [n > 1] -> Used_Stack_3 [push, pop, top, empty, up_1, up_2] (z, n-1)
[] top!x; Used_Stack_3 [push, pop, top, empty, up_1, up_2] (x, n)
[] empty!false; Used_Stack_3 [push, pop, top, empty, up_1, up_2] (x, n)
endproc

process Cell [up_1, up_2, down_1, down_2] (x:Element, n:Int) :exit :=
hide down_1', down_2' in
(up_1?y Element;
    (   [n > 0] -> down_1!x; Cell [up_1, up_2, down_1, down_2] (y, n+1)
    []  [n = 0] -> ( Cell [up_1, up_2, down_1, down_2] (y, n+1)
                   |[down_1, down_2]| Cell [down_1, down_2, down_1', down_2'] (x, 0)))
[] up_2!x;
    (   [n > 0] -> down_2?z:Element; Cell [up_1, up_2, down_1, down_2] (z, n-1)
    []  [n = 0] -> exit))
    endproc
endproc
endspec
```

Figure 10.6: The Third Stack

Consider trying to show `Stack_1` $\equiv_{wbc}$ `Stack_3` using the same method as in the previous section. In the case analysis of possible transitions from an arbitrary state, it is clear that both stacks can perform a `pop` action; it is less clear that `Stack_3` then behaves like the rest of the stack. Since `Cell` actions are hidden, we cannot use weak bisimulation congruence to analyse their behaviour; `Cell` could *reverse* the values for all we know, or generate random values. In other words, we can never be sure that `Cell` returns the right values and in the right order. To try to gain understanding of the workings of `Stack_3` we take a particular stack and simulate the process for the values of that stack.

Equational Proof for a Particular Stack

We begin by considering the way `Stack_3` behaves with the sequence of actions `push!1; push!2; pop; top?1; pop; exit`. A test that `Stack_3` really behaves like a stack will be whether or not it can synchronise on the `top!1` action. Below we unfold `Stack_3` in parallel with the above sequence of actions by hand, in order to gain some confidence in the correctness of the `Cell` processes. We begin with the expression

```
    Stack_3 |[stack_events]| push!1; push!2; pop; top!1; pop; exit                (*)
```

and attempt to obtain something of the form

```
    push!1; push!2; pop; top!1; pop; Stack_3
```

In the following unfoldings we know which event we want to occur next, therefore we only unfold the appropriate part of the expression. This helps make the unfolding more readable. Since all other actions in the choice expression are blocked by our choice of synchronisation list this is correct. To further aid readability we abbreviate the set of actions `up_1, up_2` by `up`, `down_1, down_2` by `down`, `up_1, up_2, down_1, down_2` by `updown`, and `push, pop, top, empty` by `stack_events`.

We begin by unfolding some subsidiary expressions required for the unfolding of (∗).

Lemma 1 *Our first unfolding is to push an occurrence of* `up_1!1` *through the stack expression.*

```
    hide updown in (up_1!1; Used_Stack_3 (2, 2) |[up]| Cell (1, 0))
```
$\equiv_{wbc}$
```
    i; hide updown in (Used_Stack_3 (2, 2) |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
```

We proceed by unfolding and expansion

```
    hide updown in (up_1!1; Used_Stack_3 (2, 2) |[up]| Cell (1, 0))
```
$\equiv_{wbc}$ *(unfold)*
```
    hide updown in
        (up_1!1; Used_Stack_3 (2, 2)
        |[up]| up_1?y; (Cell (y, 1) |[down]| Cell (1, 0)))
```

$\equiv_{wbc}$ *(expansion)*

```
    hide updown in
        up_1!1; (Used_Stack_3 (2, 2) |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
```

$\equiv_{wbc}$ *(f5a)*

```
    i; hide updown in (Used_Stack_3 (2, 2) |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
```

Lemma 2 *Next we unfold the expression which pushes* up_2?z *through the stack expression.*

```
    hide updown in
        ((up_2?z; Used_Stack_3 (z, 1)) |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
```

$\equiv_{wbc}$

```
    i; hide updown in (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
```

Again we proceed by unfolding and expansion. Note that the second Cell *process cannot interact with* Used_Stack_3 *as its events were renamed at instantiation.*

```
    hide updown in
        ((up_2?z; Used_Stack_3 (z, 1)) |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
```

$\equiv_{wbc}$ *(unfold)*

```
    hide updown in
        ((up_2?z; Used_Stack_3 (z, 1))
        |[up]| (up_2!1; down_2?z; Cell (z, 0) |[down]| Cell (1, 0)))
```

$\equiv_{wbc}$ *(expansion, unfold)*

```
    hide updown in up_2?1;
        (Used_Stack_3 (1, 1)
        |[up]| (down_2?z; Cell (z, 0) |[down]| down_2!1; exit))
```

$\equiv_{wbc}$ *(expansion)*

```
    hide updown in up_2?1;
        (Used_Stack_3 (1, 1) |[up]| down_2?1; (Cell (1, 0) |[down]| exit))
```

$\equiv_{wbc}$ *(expansion)*

```
    hide updown in up_2?1; down_2?1;
        (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
```

$\equiv_{wbc}$ *(f5a, m1)*

```
    i; hide updown in (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
```

We now proceed with the main proof of (∗), i.e.

Conjecture 1 *Simulation of the stack with a particular sequence of events.*

```
    Stack_3 |[stack_events]| push!1; push!2; pop; top!1; pop; exit        (*)
```

$\equiv_{wbc}$

```
    push!1; push!2; pop; top!1; pop; Stack_3
```

As with the lemmata we proceed mainly by unfolding and expansion.

```
    Stack_3 |[stack_events]| push!1; push!2; pop; top!1; pop; exit
```

$\equiv_{wbc}$ *(unfold)*

```
    (New_Stack >> accept x, n in
        (hide updown in Used_Stack_3 (x, n) |[up]| Cell (x, 0)))
    >> Stack_3 |[stack_events]| push!1; push!2; pop; top!1; pop; exit
```

Let us consider just the initial portion of the right hand side, bearing in mind that this process will eventually exit, at which point we need to re-introduce the >> Stack_3 part of the process.

```
    (New_Stack >> (accept x, n in
        (hide updown in Used_Stack_3 (x, n) |[up]| Cell (x, 0)))
    |[stack_events]| push!1; push!2; pop; top!1; pop; exit
```
$\equiv_{wbc}$ *(unfold)*
```
    ((push?x; exit (x, 1) [] empty!true; New_Stack)
        >> accept x, n in (hide updown in Used_Stack_3 (x, n) |[up]| Cell (x, 0)))
    |[stack_events]| push!1; push!2; pop; top!1; pop; exit
```
$\equiv_{wbc}$ *(expansion)*
```
    push!1;
    ((exit (1, 1) >> accept x, n in
        (hide updown in Used_Stack_3 (x, n) |[up]| Cell (x, 0)))
    |[stack_events]| push!2; pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(d2b, m1)*
```
    push!1;
        ((hide updown in Used_Stack_3 (1, 1) |[up]| Cell (1, 0))
        |[stack_events]| push!2; pop; top!1; pop; exit)
```

Since we know that we want the action **push!2** to proceed, and that all others are restricted by the synchronisation list of the parallel operators, we can unfold this expression further.

```
    push!1;
        ((hide updown in push?y; up_1!1; Used_Stack_3 (y, 2) |[up]| Cell (1, 0))
        |[stack_events]| push!2; pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(f5b)*
```
    push!1; ((push?y; (hide updown in up_1!1; Used_Stack_3 (y, 2) |[up]| Cell (1, 0)))
    |[stack_events]| push!2; pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(expansion)*
```
    push!1; push!2;
        ((hide updown in (up_1!1; Used_Stack_3 (2, 2) |[up]| Cell (1, 0)))
        |[stack_events]| pop; top!1; pop; exit)
```

From the result of the first lemma we substitute for the expression
hide updown in (up_1!1; Used_Stack_3 (2, 2) |[up]| Cell (1, 0))), giving:

```
    push!1; push!2;
        (i; hide updown in (Used_Stack_3 (2, 2)
                            |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
    |[stack_events]| pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(expansion, m1)*
```
    push!1; push!2;
        (hide updown in (Used_Stack_3 (2, 2)
                            |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
    |[stack_events]| pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(unfold)*
```
    push!1; push!2;
        (hide updown in (   pop; up_2?z; Used_Stack_3 (z, 1)
                            |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
        |[stack_events]| pop; top!1; pop; exit)
```
$\equiv_{wbc}$ *(expansion)*
```
    push!1; push!2; pop;
        (hide updown in (   (up_2?z; Used_Stack_3 (z, 1))
                            |[up]| (Cell (1, 1) |[down]| Cell (1, 0)))
        |[stack_events]| top!1; pop; exit)
```

Using the second lemma we can substitute the unfolding of **hide** updown **in** ((up_2?z; Used_Stack_3 (z, 1)) |[up]| (Cell (1, 1) |[down]| Cell (1, 0))) into the current unfolding

```
    push!1; push!2; pop;
        (i; hide updown in (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| top!1; pop; exit)
≡wbc                                                        ( expansion, unfold)
    push!1; push!2; pop; i;
        (hide updown in (top!1; Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| top!1; pop; exit)
≡wbc                                                        ( m1, expansion)
    push!1; push!2; pop; top!1;
        (hide updown in (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| pop; exit)
```

Here we see that Used_Stack_3 does indeed have the correct value for the top of the stack, since it is able to perform the top!1 action. We continue with the unfolding to check that we return to the original state.

```
    push!1; push!2; pop; top!1;
        (hide updown in (Used_Stack_3 (1, 1) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| pop; exit)
≡wbc                                                        ( unfold)
    push!1; push!2; pop; top!1;
        (hide updown in ((pop; up_2?z; exit) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| pop; exit)
≡wbc                                                        ( expansion)
    push!1; push!2; pop; top!1; pop;
        (hide updown in ((up_2?z; exit) |[up]| (Cell (1, 0) |[down]| exit))
        |[stack_events]| exit)
≡wbc                                                        ( unfold)
    push!1; push!2; pop; top!1; pop;
        (hide updown in ((up_2?z; exit) |[up]| ((up_2!1; exit) |[down]| exit))
        |[stack_events]| exit)
≡wbc                                                        ( expansion)
    push!1; push!2; pop; top!1; pop;
        (hide updown in up_2?1; (exit |[up]| (exit |[down]| exit))
        |[stack_events]| exit)
≡wbc                                                        ( c3a, f5a, m1, expansion)
    push!1; push!2; pop; top!1; pop;
        (hide updown in (exit |[up]| exit) |[stack_events]| exit)
```

At this point we should remember that a part of the Used_Stack_3 process was stripped away for convenience; we put it back now.

```
    push!1; push!2; pop; top!1; pop;
        (hide updown in exit >> Stack_3 |[up]| exit) |[stack_events]| exit
≡wbc                                                        ( d2a, f5a, expansion, m1)
    push!1; push!2; pop; top!1; pop;
        ((hide updown in Stack_3 |[up]| exit) |[stack_events]| exit)
```

$\equiv_{wbc}$ *(c5)*

```
    push!1; push!2; pop; top!1; pop;
        ((hide updown in Stack_3 |[stack_events]| exit) |[stack_events]| exit)
```

$\equiv_{wbc}$ *(c2, c3a)*

```
    push!1; push!2; pop; top!1; pop; (hide updown in Stack_3 |[stack_events]| exit)
```

The occurrence of **hide** will eventually be eliminated as at some point in the unfolding we get two occurrences of **hide**, and one can be removed. In the last step above, we can remove one occurrence of **exit** by manipulation of the parallel operators, but not all. Since `Stack_3` is nonterminating, this **exit** cannot be removed, as it has no other **exit** process with which it may synchronise; however, it will not interfere with the actions of `Stack_3`. This means that we are in fact unable to prove the conjecture ($*$) holds.

It can be seen that this unfolding is a very tedious process, and that, although we may have gained some understanding of the behaviour of `Cell` and `Used_Stack_3`, and hence some extra confidence in the correctness of the behaviour, this cannot be taken as a formal proof that `Stack_3` behaves like a stack.

We go on to consider other possible approaches to proving that `Stack_3` has the same behaviour as `Stack_1` or `Stack_2`.

Other Approaches to the Verification

An alternative approach to the proof that `Stack_3` has the same behaviour as `Stack_1` or `Stack_2` might be to try to prove something about the `Cell` structure, for example, that it has the same behaviour as `Stack_1` or `Stack_2`, and to use that proof to conclude that `Stack_3` therefore behaves like a stack. Informally, the structure built of `Cell` processes behaves similarly to a stack, but with the bottom element held twice, and the actions renamed. Although we can relate the action **up_1** to **push**, **up_2** is more like a combination of `top` and `pop`; there is no `Cell` action which relates to `empty`. These differences mean that we cannot say anything about `Cell` in relation to `Stack_1`, since we cannot express that two actions are combined into one. Similarly, we cannot attempt a proof of `Stack_3` $\equiv_{wbc}$ `Stack_1` with the occurrence of **hide** removed as the `Cell` actions have no counterparts in `Stack_1`, therefore the equivalence will certainly not hold.

It seems that we cannot perform a proof of behavioural equivalence between two specifications to show that `Stack_3` has the correct behaviour; are there any other ways of expressing the requirement that `Stack_3` behaves like a stack? In section 9.2 we used the notion of property testing to express the bad behaviours of the radiation machine, perhaps we can use a similar technique to express good behaviours of the stack. We can use the equations of the abstract data type stack in figure 10.2, and also some information about which parts of the process model empty stacks, to give specifications of properties which the stack may be expected to display. For example:

1. A **push** action followed by a **pop** action should leave the stack unchanged,

2. A `push?x` action followed by a `top!z` action should satisfy $x = z$,

3. `New_Stack_3` can never perform `empty!false`, only `empty!true`,

4. `Used_Stack_3 (x, n)` can never perform `empty!true`,

5. `New_Stack_3` can never perform `top` or `pop` events,

6. `Used_Stack_3 (x, n)` performs `top!x`, not `top!z` where $x \neq z$.

We consider formulating the first property above in more detail. The abstract data type equation we use as our specification is `Pop(Push(s, x)) = s`; this property can be formulated in terms of transitions as follows:

$$\texttt{Used\_Stack\_3 (z, n)} \xrightarrow{\text{push?x}} \texttt{P} \xrightarrow{\text{pop}} \texttt{Used\_Stack\_3 (z, n)}$$

Given that so far we have working on equivalence proofs, we would like to be able to express this as a relation between processes, e.g.

```
push?x:Element; pop; Used_Stack_3 (z, n) = Used_Stack_3 (z, n)
```

The question is: which relation should be used in place of =? It seems that the above equation is too strict if a relation such as weak bisimulation is used as then it says that `Used_Stack_3 (z, n)` may *only* perform **push** then **pop** actions. What we really want to say is that this is one possible course of action, suggesting the use of the **cred** relation. If we try to prove the conjecture above with **cred** for = we quickly encounter another problem: `Used_Stack_3` does not operate in isolation. In the conjecture there is no mention of the `Cell` processes. Including `Cell` processes gives an expression of the form:

```
    push?x:Element; pop;
        (Used_Stack_3 (z, n) |[...]| Cell (xn, n) |[...]| ...|[...]| Cell (x1, 0))
cred
    (Used_Stack_3 (z, n) |[...]| Cell (xn, n) |[...]| ...|[...]| Cell (x1, 0))
```

assuming that the synchronisation lists are properly defined.

This formulation of the property has two problems. The first is that the proof will be extremely tedious, and the second is that if we have to specify each `xi` then we are making the proof for a particular stack, rather than for an arbitrary stack. On the other hand, if we do not specify the `xi` we must at least specify `n` and then perform an induction proof over `n` to show that the equivalence holds for all `n`. Again, this analysis will be tedious. The problem is that really that we are attempting to prove something about a process containing uninstantiated variables.

Technically, such a process has no meaning in the LOTOS semantics, unless the specification is parameterised over those variables, although we can think of it as representing a *class* of processes. In the previous proof, of `Stack_1` $\equiv_{wbc}$ `Stack_2`, we ignored this fact and performed the proof for an arbitrary stack. This did not matter as the laws of the $\gg$ allowed us to move the variables around appropriately. Here the problem is exaggerated because we cannot rely on the laws of weak bisimulation congruence since the actions of `Cell` are hidden, and we have to rely instead on trying to investigate the values of the variables (therefore we must first assign values to those variables).

The problems with `Stack_3` arise because we are using LOTOS both as a specifications language and as the meta-language in which we formulate desirable properties. While this was acceptable for simple properties, we now encounter properties which require a more powerful meta-language incorporating, for example, variables and quantifiers. The LOTOS formalism is more suited to describing the *construction* of a system, than expressing a property of that system at a more abstract level. The property we try to describe above can be easily expressed in terms of transitions: if we are at a state s in a labelled transition system we can perform some actions to move to state s', and additionally we might be able to say something about a relationship between s and s', such as $s = s'$. A more natural formalism to use for this sort of property is a modal or temporal logic; the use of such a logic in conjunction with LOTOS is discussed in chapter 11.

10.2.7 Summary and Discussion

We have studied three versions of the stack originally given in [Got87] and tried to prove them equivalent in some sense. While we were able to prove that `Stack_1` $\equiv_{wbc}$ `Stack_2` by using a combination of equational reasoning and bisimulation construction, we were unable to prove anything about the relationship between `Stack_3` and the other two, mainly because we were unable to formulate the conjecture to be proved. The problem of verification of full LOTOS requires more study.

Other interesting observations made in [Got87] include some comments about the suitability of LOTOS as a tool for specification. Some deficiencies mentioned are: the lack of a mechanism to specify groups of events as atomic, i.e. to be performed without interleaving. This was something we also encountered in the Login case study of chapter 7. Another possible problem identified by Gotzhein is that LOTOS is a constructive language and that this might lead a specifier towards a more implementation influenced design than if a more abstract language had been used. The author does also praise LOTOS; in particular the ability to make specifications more readable (and hence understandable) by modularisation, especially the hierarchical structuring of processes.

Given that the approach to verification which we have been using up till now, i.e. proving the implementation satisfies the specification, is not as appropriate for full LOTOS as it was for

Basic LOTOS, we now consider the approaches of other authors to the problem of verification of full LOTOS specifications which do not require the addition of other operators or adoption of other formalisms, and which will allow us to remain within the equational reasoning paradigm, specifically the proof system developed in chapter 8. These approaches work on the principle that if full LOTOS is hard to deal with, then we must somehow separate the data types from the process algebra; this includes totally removing the adt part of the specification. We present the main ideas in more detail of two approaches, and consider how we might use them in practice. Both approaches are illustrated by an example; one the stack of this section, and the other a full LOTOS extension of the radiation machine example of chapter 9.

10.3 A Unified Framework

Part of the problem with verifying full LOTOS specifications is the apparent gap between objects described using adts and objects described using process algebra. One method of narrowing that gap is to create a unified framework for specification and verification by describing both parts of the language in the same framework, e.g. in [Ple87] a CCS like language with adts is given a semantics in terms of term rewriting systems. Another approach, which will be discussed in chapter 11, is to use a more powerful formalism to describe system properties. Yet another alternative, which is explored here, is to give the semantics of one part of the language in terms of the other part by way of a translation. In the LOTOS literature this has been done in both directions: process algebra in ACT ONE and data types as processes.

The semantics of Basic LOTOS in terms of ACT ONE can be found in [Raf92, EBB$^+$86]. However, [Raf92] states that it is impossible to have a full and complete axiomatisation of LOTOS in a many-sorted equational logic. This is because LOTOS contains operators expressing temporal ordering of events and nondeterminism. Therefore, there is never a guaranteed transformation from Basic LOTOS to ACT ONE. For our purposes, it is more useful to look at how data types can be coded into processes, since we already have techniques for dealing with Basic LOTOS processes. There are two main approaches at present to translating data types into processes: *abstract interpretation* and *context equations*. These are examined below.

10.3.1 Abstract Interpretation and LOTOS

Abstract Interpretation and LOTOS

The problem with proofs involving full LOTOS is that they are too complicated. On the other hand, proofs involving Basic LOTOS are much more straightforward, and a variety of proof techniques and tools for such proofs exist. For some types of analyses, not all of the information in the full LOTOS specification will be used; we need to get rid of the information we want to ignore, retaining only the information necessary to complete the analysis. Abstract interpretation is a well-known technique in partial evaluation of functional programs whereby a large/complex data type is replaced in the program by a smaller/simpler data type. This is done in order to allow evaluation of properties of the program, such as termination. By throwing away some information, analysis of the program behaviour becomes more tractable. The difficult part is retaining enough of the right information to make the evaluation meaningful. The same approach can be used to translate full LOTOS specifications into Basic LOTOS specifications. This was investigated as part of the LOTOSPHERE project, and is reported in chapter 13 of the Catalogue of LOTOS Correctness Preserving Transformations [Bol92] produced by the project.

In this section we describe two transformations from full to Basic LOTOS from [Bol92]. In the first, all data information is thrown away, leaving only the process skeleton, while in the second, the data type information is retained by coding it into the process expressions. The correctness of these transformations is expressed using modified versions of the simulation and bisimulation relations. We give informal descriptions of the transformations and the relations; the full formal descriptions may be found in [Bol92], illustrating the transformations by our own examples.

Obviously, transforming a full LOTOS specification into a Basic LOTOS specification in order to simplify verification proofs is only justifiable if we can infer properties of the full LOTOS specification based on the Basic LOTOS specification. An important question not addressed in [Bol92] is: given two full LOTOS descriptions and their Basic LOTOS transformations, if we prove something about the relationship between the Basic LOTOS specifications, can we then conclude anything about the relationship between the full LOTOS specifications? We consider this question in detail, with specific reference to the equivalence/congruence/preorder relations we have used so far. Finally, we illustrate the use of one of the transformations by applying it to a full LOTOS version of the radiation machine example of chapter 9, and completing a proof of safety of the Basic LOTOS transformation of the machine in PAM.

Remove Data Type Information Completely

The first transformation we consider throws away all the information related to the data type. Given any full LOTOS specification, we can always totally remove the data types to give a Basic LOTOS specification by syntactically removing the adt part of the specification and any reference to adts in the process expressions. In performing this transformation, information about data which is important to the flow of control in the process, if there is any such data, will be lost, and hence the process behaviour may be altered. For example, given the process P and its Basic LOTOS transformation P',

```
P = in?n:Int; (    [n > 0] -> a; P          P' = in; (   a; P'
                [] [n = 0] -> b; P                    []  b; P'
                [] [n < 0] -> c; P)                   []  c; P')
```

consider the consequences of executing P or P′ in parallel with the processes Q and R, given below.

```
Q = b; Q        R = (a; R [] b; R [] c; R)
```

Both Q and R can be successfully run in parallel with P′, synchronising on a, b and c in both cases. However, it is clear that, with the same synchronisation list, while R runs successfully with P, Q in parallel with P may deadlock, since P may be forced to take either the a or the c action because of the value of n, whereas Q may only perform b. Even in these very simple processes, the data makes a considerable difference. Having said that, this transformation may be of some use in reachability analysis. Consider the process S which performs only d actions. Even with all the data removed, it is obvious that P |[a, b, c, d]| S will deadlock.

The correctness of this transformation can be expressed in terms of a *simulation* relation over processes. A simulation relation is one half of a bisimulation relation, i.e. only one clause of definition 1 of chapter 3 need be satisfied. We write Q simulates P, $P \leq Q$, if Q can perform all the actions of P (but P need not perform all the actions of Q). For this transformation, the simulation relation is actually a *family* of relations, indexed by a coding function on events. The coding function throws away all data values, i.e. if gw is a transition label, the coding function returns g. The coding function is injective with respect to g, therefore distinctions between gate names are preserved. The definitions of these relations may be found in [Bol92].

Using these simulation relations, the correctness of the abstract interpretation which removes all the data type information is given by FL $\leq_{\phi}$ BL, where ϕ is the coding function, FL the full LOTOS specification and BL the Basic LOTOS specification.

Initially this preorder may seem to be written the wrong way round, but bear in mind that the Basic LOTOS translation may be able to perform *more* actions than the Full LOTOS specification can (because some of the restrictions enforced by the data type have been removed). What this preorder says is that the Basic LOTOS process can perform all the actions the Full LOTOS process can, i.e. it does not refuse any actions allowed by the Full LOTOS process.

As mentioned above, this coding is useful for reachability analyses. It is also useful in cases where the data does not affect the flow of control within the process. Unfortunately, there are lots of examples of processes in which the data does affect the flow of control. One such is the radiation machine example presented in section 10.3.2, where data types are added to indicate the level of radiation and the position of the shield. These values are then used to decide whether or not it is safe to fire the electron beam. Obviously, it would be inappropriate to use the above transformation on this example since the data types carry important information about the flow of control of the process.

The next section presents a transformation which retains the data type information.

Retain Data Type Information Completely

At the other end of the spectrum, we may take a full LOTOS process and transform it into a Basic LOTOS process retaining all the data type information. There are two provisos: either the data types involved must be finite, or the target language must provide a means of expressing infinite choice over a data range. Although full LOTOS has an operator which gives a choice over a possibly infinite range of data, Basic LOTOS has only finite choice, therefore we must ensure that the data types we are dealing with are finite.

The basic idea of this transformation is to code the values and variables of the data types into the process expressions. For example, a gate with an output variable is replaced by a new gate name which incorporates the old gate name and the value represented by the output variable. Gates with input variables are replaced by a choice over all the values of the appropriate type; again the values are coded into the new gate name. As a concrete example, given a type `Three` which denotes the set {1, 2, 3},

```
P = in?n:Three; P          becomes          P =      in.1; P
                                                 []  in.2; P
                                                 []  in.3; P
```

This is of course the way in which value passing is implemented in CCS.

In practice, it is not enough for the original data types to merely be finite, ideally each type should have only a small number of values, i.e. probably fewer than 10. This requirement is not as restrictive as it may seem at first; the types of several specifications in the literature satisfy this constraint, for example [Tho94, FLS90, EFJ90, DP91] and the OSI Session layer specifications [Sco89, vS89, Aju89] all use enumerated types over a small range (but see below).

In addition to the basic transformation of event offers given above, other expressions involving data types must also be evaluated. A record of which values are assigned to which variables (the environment) must be carried around to allow the evaluation of guards and instantiation of process parameters. This makes this transformation far more complex than the previous one. We do not give the transformation details here, but illustrate its use in section 10.3.2 via the radiation machine example. A full description of this transformation may be found in [Bol92].

The correctness of the previous transformation was expressed using a simulation *preorder* in conjunction with a coding function; we can use something a little stronger here: a ***bisimulation*** relation. This time the coding function must be bijective, because the definition of the bisimulation relation requires the application of ϕ^{-1} to transform Basic LOTOS labels into full LOTOS labels.

The correctness of this transformation is expressed by `BL` $\simeq_\phi$ `FL` where the coding function ϕ performs the sort of transformations described earlier in this section. The full definition of the coding function, and the bisimulation relation may be found in [Bol92].

While this transformation does retain all the data type information, which is good, it may also give a huge explosion in the size of the specification. For example, if the processes in the full LOTOS specification have data parameters, then each process is transformed into many processes, one for each instantiation of the data parameters. Similarly, one variable event offer results in several choice branches, one for each possible value of the variable.

A further problem of this transformation is the constraint that all types must be finite; although some of the types of a specification may satisfy this constraint, the same specification may also include infinite data types (typically the natural numbers or the integers). Several examples of such specifications may be found in [vEVD89], including the OSI Session layer specifications mentioned above. Many specifications include the integers or naturals as an unlimited source of unique identifiers; if the process control relies in any way on comparison of these identifiers, then neither of the abstract interpretation techniques described so far can be used.

An alternative transformation is given by choosing some sort of intermediate step between throwing away all the data information and retaining all the data information. For example, the Integers may be represented by the type {negative, zero, positive}. Another alternative is to retain the *type* of values, but not the values themselves, so at least some sort of type checking can be carried out. As with the first transformation, correctness of these transformations is expressed using the simulation relation, $\leq_\phi$, as some information is being lost.

Assuming that we can use these transformations on a given specification, it is important to know what implications results proved for the Basic LOTOS transformed specification will have for the original full LOTOS specification. This question was not considered in [Bol92]; we consider it below.

Properties of the Simulation Relations

Since the reason for using abstract interpretation is to allow verification to be carried out on the simpler Basic LOTOS specifications rather than the full LOTOS specifications, it is important to know what implications equivalences, or orderings, proved between Basic LOTOS specifications have for the relationships between the full LOTOS specifications they represent. Obviously, any relations depend heavily on the coding function used, so we will consider the two main transformations given above separately.

Coding Function Throws Away All Data Information Consider the coding which throws away all data type information. Assume we have two full LOTOS specifications, $\mathtt{FL}_1$ and $\mathtt{FL}_2$. From these we can derive two Basic LOTOS specifications, $\mathtt{BL}_1$ and $\mathtt{BL}_2$, so we have $\mathtt{FL}_1 \leq_\phi \mathtt{BL}_1$ and $\mathtt{FL}_2 \leq_\phi \mathtt{BL}_2$. Now assume that we can prove something about the relationship between

BL₁ and BL₂, say BL₁ $\sim$ BL₂, i.e. BL₁ is strongly bisimilar to BL₂. What does this imply for the relationship between FL₁ and FL₂? Unfortunately, the answer is that *nothing* can be implied about this relationship. As a counter example, take the case in which the strongest possible relation, and hence all other relations, holds between BL₁ and BL₂: let them be the same process. The processes FL₁ and FL₂ are defined as follows, the transformed processes are denoted BL₁ and BL₂:

```
FL1 = a?n:Int; (   [false] -> b; FL1         BL1 = a; (b; BL1 [] c; BL1)
                   [true] -> c; FL1)
FL2 = a?n:Int; (   [true] -> b; FL2         BL2 = a; (b; BL2 [] c; BL2)
                   [false] -> c; FL2)
```

Obviously FL₁ $\leq_\phi$ BL₁ and FL₂ $\leq_\phi$ BL₂, remembering that the BL$_i$ may do more than the FL$_i$. It also plain that FL₁ bears no relation to FL₂; they are not even trace equivalent. It is not even true to say that FL₁ **cred** FL₂ as FL₂ **after a must** b while $\neg$ (FL₁ **after a must** b). Throwing away all the data type information means that while some general properties, like reachability and language as mentioned earlier, are preserved, equations concerning the equivalence or ordering of the full LOTOS specifications cannot be inferred from the results proved about the equivalence or ordering of the Basic LOTOS processes.

Fortunately, this is not the case for the coding function which retains all information.

Coding Function Retains All Data Information Given the transformation which preserves the data information of a full LOTOS specification by coding it into the gate name of a Basic LOTOS specification, any relation which holds of the two (transformed) Basic LOTOS specifications can be shown to hold, modulo the composition of the coding functions, for their corresponding full LOTOS specifications. The relations weak bisimulation congruence, testing congruence and **cred** can all be defined modulo a coding function in the obvious way.

Below we present results concerning the relations strong bisimulation equivalence, weak bisimulation congruence and **cred** (and therefore testing congruence). We give only sketches of the proofs.

Theorem 4 $(\mathtt{FL}_1 \simeq_{\phi_1} \mathtt{BL}_1 \wedge \mathtt{FL}_2 \simeq_{\phi_2} \mathtt{BL}_2 \wedge \mathtt{BL}_1 \sim \mathtt{BL}_2) \Rightarrow \mathtt{FL}_1 \simeq_{\phi_3} \mathtt{FL}_2$

Proof. We know that the following relations hold: FL₁ $\simeq_{\phi_1}$ BL₁ and FL₂ $\simeq_{\phi_2}$ BL₂. Notice that different coding functions are used here to allow renaming of gates and data elements since we expect that FL₁ and FL₂ use different data types. If, in addition, BL₁ $\sim$ BL₂, then we can deduce FL₁ $\simeq_{\phi_3}$ FL₂, where $\phi_3 = \phi_1 \circ \phi_2^{-1}$ and $\circ$ denotes function composition.

Informally, the proof proceeds by showing that a bisimulation, R_{ϕ_3}, exists between FL₁ and FL₂, i.e. we show that

$$\forall \alpha \in Act\, \mathtt{fl}_1 \xrightarrow{\alpha} \mathtt{fl}_1{}' \Rightarrow \exists \mathtt{fl}_2 . \mathtt{fl}_2 \xrightarrow{\phi_3(\alpha)} \mathtt{fl}_2{}' \wedge (\mathtt{fl}_1', \mathtt{fl}_2') \in R_{\phi_3}$$

and vice versa.

If a particular element, α, of *Act* is chosen, it remains to be proven that a state $\mathtt{fl}_2'$ exists such that $\mathtt{fl}_2 \xrightarrow{\phi_3(\alpha)} \mathtt{fl}_2' \wedge (\mathtt{fl}_1', \mathtt{fl}_2') \in R_{\phi_3}$. The state $\mathtt{fl}_2'$ can be found by tracing through the other bisimulations:

$$\begin{array}{ll}
\mathtt{FL}_1 \simeq_{\phi_1} \mathtt{BL}_1 \wedge \mathtt{fl}_1 \xrightarrow{\alpha} \mathtt{fl}_1' & \Rightarrow \exists\ \mathtt{bl}_1'.\mathtt{bl}_1 \xrightarrow{\phi_1(\alpha)} \mathtt{bl}_1' \ \wedge\ (\mathtt{fl}_1',\ \mathtt{bl}_1') \in R_{\phi_1} \\
\mathtt{BL}_1 \sim \mathtt{BL}_2 \wedge \mathtt{bl}_1 \xrightarrow{\phi_1(\alpha)} \mathtt{bl}_1' & \Rightarrow \exists\ \mathtt{bl}_2'.\mathtt{bl}_2 \xrightarrow{\phi_1(\alpha)} \mathtt{bl}_2' \ \wedge\ (\mathtt{bl}_1',\ \mathtt{bl}_2') \in R \\
\mathtt{BL}_2 \simeq_{\phi_2} \mathtt{FL}_2 \wedge \mathtt{bl}_2 \xrightarrow{\phi_1(\alpha)} \mathtt{bl}_2' & \Rightarrow \exists\ \mathtt{fl}_2'.\mathtt{fl}_2 \xrightarrow{\phi_2^{-1}(\phi_1(\alpha))} \mathtt{fl}_2' \ \wedge\ (\mathtt{bl}_2',\ \mathtt{fl}_2') \in R_{\phi_2}
\end{array}$$

Since, by definition, a bisimulation equivalence contains all bisimulation relations, we know that $\simeq_{\phi_3}$ must contain the relations R, R_{ϕ_1} and R_{ϕ_2}. We also know that we can construct R_{ϕ_3} by the transitive closure of these relations, therefore $(\mathtt{fl}_1',\ \mathtt{fl}_2') \in R_{\phi_3}$. Note that the coding functions ϕ_1 and ϕ_2 are bijective, as required by the definition of the bisimulation relation.

The proof for the second clause is similar. ∎

So actually what we get from this chain of relations is a bisimulation modulo variations in the names of the data types, i.e. the application of ϕ_1 and ϕ_2. This is important because, according to the LOTOS semantics, even if two algebras A and B are isomorphic, and we have two processes which are identical except that one process uses data values from A and the other uses data values from B, these processes are not bisimilar; however, they will be related by $\simeq_\phi$.

Theorem 5 $(\mathtt{FL}_1 \simeq_{\phi_1} \mathtt{BL}_1 \wedge \mathtt{FL}_2 \simeq_{\phi_2} \mathtt{BL}_2 \wedge \mathtt{BL}_1 \equiv_{wbc} \mathtt{BL}_2) \Rightarrow \mathtt{FL}_1 \equiv_{wbc_{\phi_3}} \mathtt{FL}_2$

Proof. The proof is similar to that for strong bisimulation equivalence above, therefore we do not present it here. We remark that the only differences occur when considering **i** actions, but that since the coding function is defined to map **i** to itself, the transition relation $\Longrightarrow$ behaves as expected. The only difficult point might be in requiring congruence, rather than equivalence, but the only **i** transition which affects congruence is one from the root of the lts, and since ϕ maps **i** to itself and the Basic LOTOS processes are congruent, there can be no **i** transitions which affect the behaviour of the processes in different contexts. ∎

Theorem 6 $(\mathtt{FL}_1 \simeq_{\phi_1} \mathtt{BL}_1 \wedge \mathtt{FL}_2 \simeq_{\phi_2} \mathtt{BL}_2 \wedge \mathtt{BL}_1 \equiv_{tc} \mathtt{BL}_2) \Rightarrow \mathtt{FL}_1 \equiv_{tc_{\phi_3}} \mathtt{FL}_2$

Proof. We prove this theorem by showing that the result holds for **cred**, rather than $\equiv_{tc}$, first in one direction and then the other. Below we give the proof for one direction only; the other direction is similar. ∎

Theorem 7 $(\mathtt{FL}_1 \simeq_{\phi_1} \mathtt{BL}_1 \wedge \mathtt{FL}_2 \simeq_{\phi_2} \mathtt{BL}_2 \wedge \mathtt{BL}_1\ \mathbf{cred}\ \mathtt{BL}_2) \Rightarrow \mathtt{FL}_1\ \mathbf{cred}_{\phi_3}\ \mathtt{FL}_2$

Proof. The proof proceeds by contradiction. In addition to the assumptions $\texttt{FL}_1 \simeq_{\phi_1} \texttt{BL}_1$ and $\texttt{FL}_2 \simeq_{\phi_2} \texttt{BL}_2$, we also assume $\neg$ ($\texttt{FL}_1$ **cred**$_{\phi_3}$ $\texttt{FL}_2$), i.e.

$$\exists t \exists L. \neg(\texttt{FL}_2 \textbf{ after } \phi_3(t) \textbf{ must } \phi_3(L) \Rightarrow \texttt{FL}_1 \textbf{ after } t \textbf{ must } L)$$

We try to derive $\texttt{BL}_1$ **cred** $\texttt{BL}_2$, the third assumption of the theorem above.

We know that $\texttt{FL}_2$ must pass a test which $\texttt{FL}_1$ does not, so let t and L denote that particular trace and test set, modulo the coding functions (there may be other traces and tests, but only one representative is required). Since $\texttt{FL}_1$ does not pass the test, we know that $t \in tr(\texttt{FL}_1)$, because if $t \notin tr(\texttt{FL}_1)$ then $\texttt{FL}_1$ would pass any test. This means that the set $\texttt{FL}_1$ **after** t must have at least one member, call it $\texttt{fl}_1$. We also know that $\neg(\texttt{fl}_1 \textbf{ must } L)$.

Since $\texttt{FL}_1 \simeq_{\phi_1} \texttt{BL}_1$, then $\texttt{BL}_1$ may perform the trace $\phi_1(t)$, and, moreover, may reach a state, call it $\texttt{bl}_1$, which is bisimilar to $\texttt{fl}_1$. As these states are bisimilar, they must be able to perform the same actions, in particular, they are both *unable* to perform the actions in $\phi_1(L)$ and L respectively. We therefore have $\neg(\texttt{BL}_1 \textbf{ after } \phi_1(t) \textbf{ must } \phi_1(L))$.

We now consider the state $\texttt{FL}_2$, the trace $\phi_3(t)$ and the test set $\phi_3(L)$. We know that $\texttt{FL}_2$ passes this test. We must now try to determine whether or not $\texttt{BL}_2$ passes the corresponding test.

We have two cases, either $\phi_3(t) \notin tr(\texttt{FL}_2)$, or $\phi_3(t) \in tr(\texttt{FL}_2)$. In the first case there is no $\texttt{fl}'_2$ such that $\texttt{fl}_2 \xrightarrow{\phi_3(t)} \texttt{fl}'_2$, and therefore there can be no $\texttt{bl}'_2$ such that $\texttt{bl}_2 \xrightarrow{\phi_1(t)} \texttt{bl}'_2$, and $\texttt{BL}_2$ passes the test $\phi_1(L)$ vacuously. We use ϕ_1 here because $\phi_3 \circ \phi_2 = \phi_1$.

In the second case we can select a specific state $\texttt{fl}_2$ from $\texttt{FL}_2$ **after** $\phi_3(t)$ since we know that the set is not empty. We can then identify a state $\texttt{bl}_2$ in $\texttt{BL}_2$ **after** $\phi_1(t)$ which is bisimilar to $\texttt{fl}_2$, and can therefore perform the same actions as $\texttt{fl}_2$. In particular, $\texttt{bl}_2$ may perform the actions in $\phi_1(L)$, and therefore pass the test, i.e. $\texttt{BL}_2$ **after** $\phi_1(t)$ **must** $\phi_1(L)$.

From the above cases we may deduce that $\texttt{BL}_2$ passes a test which $\texttt{BL}_1$ does not, and $\texttt{BL}_1$ **cred** $\texttt{BL}_2$ does not hold, contradicting our original assumptions, therefore theorem 7 holds. ∎

These results allow us to take two full LOTOS specifications, to transform them into Basic LOTOS specifications, to prove some relation holds between the Basic LOTOS specifications, and to then deduce that a similar relation, modulo the coding function, holds of the full LOTOS specifications.

The next section demonstrates the use of the second transformation by applying it to the radiation machine example already discussed in chapter 9.

10.3.2 Abstract Interpretation and the Radiation Machine

In section 9.2 the simple radiation machine was introduced, and we concluded there that in order to have better control of the shield and beam, or rather, to have better *information about* the state of the shield and beam, data types should be introduced to the specification. This modification was originally presented in [Tho94]. We present it here in order to illustrate the full LOTOS to Basic LOTOS transformation discussed in the previous section. Having obtained an equivalent specification in Basic LOTOS, we attempt to prove the safety of the machine using the proof system, PAM together with sets of axioms for weak bisimulation congruence and **cred**, developed in section 9.2.

The Full LOTOS Description of the Radiation Machine

The full LOTOS specification is presented in figure 10.7, with auxiliary definitions in figures 10.8 and 10.9. There are several changes from the original presentation in [Tho94].

In [Tho94] the possibility that the electron beam may drift from the correct setting (and need recalibration) is modelled by the inclusion of a `mid` constant in the `beam` data type which is neither `high` nor `low`. We have decided to ignore this possibility, remaining with the simpler data type including only `high` and `low`, since the actual recalibration is not modelled anyway, and inclusion of the `mid` value would merely add to the complexity of the proof (unnecessarily).

One main alteration is made to the behaviour part of the radiation machine specification. In [Tho94] the hiding of events is not uniform in the `ELECTRON` and `XRAY` processes; this means that unfolding some terms gives **hide** `lb, ls` **in** `TREATMENT`, while unfolding of other terms gives **hide** `lb, ls, hb, hs, xr, el` **in** `TREATMENT`. These two process expressions cannot be identified by PAM as equivalent, and thus the proof becomes impossible. The hiding of events is uniform in our processes as this eases the proof process, and make no difference to the eventual behaviour of a given process.

Translating to Basic LOTOS

We apply the translation as given in [Bol92] to turn the full LOTOS specification into a Basic LOTOS specification, preserving the data type information. The main steps are:

- replacing a parameterised process by several new process descriptions; one for each instantiation of the data parameters,
- converting event offers such as `fire!Low!Down` into gate names `fire_Low_Down`,
- converting δ *Low Down* events to gates `d_Low_Down`,
- removing choices whose guards evaluate to `false`,
- updating the synchronisation lists of parallel operators to take account of the new events.

```
specification Therac2

library Boolean endlib
type SHIELD is (* as in figure 10.8 *) endtype
type BEAM is (* as in figure 10.8 *) endtype
type ERROR is (* as in figure 10.9 *) endtype

behaviour
STARTUP [fire]
where

process STARTUP [fire]:exit (beam, shield) :=
hide lb, ls in lb; ls; TREATMENT [fire] (low, down)
endproc

process TREATMENT [fire] (b:beam, s:shield):exit (beam, shield) :=
    (xr; XRAY [fire] (b, s))
[]  (el; ELECTRON [fire] (b, s))
[]  exit (b, s)
endproc

process ELECTRON [fire] (b:beam, s:shield):exit (beam, shield) :=
hide lb, hb, ls, hs in
        (FIRE [fire](b, s) >> TREATMENT [fire] (b, s))
    []  TREATMENT [fire] (b, s)
endproc

process XRAY [fire] (b:beam, s:shield):exit (beam, shield) :=
hide lb, hb, ls, hs in
   TREATMENT [fire] (b, s)
[] hb; (    TREATMENT [fire] (high, s)
       []   hs; (    TREATMENT [fire] (high, up)
                []   (FIRE [fire] (high, up) >>
                         (    TREATMENT [fire] (high, up)
                         []  lb;  (    TREATMENT [fire] (low, up)
                                  []  ls; TREATMENT [fire] (low, down)))))
endproc

process FIRE [fire] (b:beam, s:shield):exit :=
hide err in
        [(b eq high) and (s eq down)] -> ERROR [err] (err54)
    []  [(b eq high) and (s eq up)] -> ZAP [fire] (b, s)
    []  [(b eq low) and (s eq down)] -> ZAP [fire] (b, s)
    []  [(b eq low) and (s eq up)] -> ERROR [err] (err55)
endproc

process ZAP [fire] (b:beam, s:shield):exit := fire!b!s; exit endproc

process ERROR [err] (e:errnum):exit := err!e; exit endproc
endspec
```

Figure 10.7: Therac Specification II

```
type SHIELD is Boolean
sort shield
opns up, down :  -> shield
     _eq_ :  shield, shield -> Bool
eqns ofsort Bool
     up eq down   = false;
     up eq up     = true;
     down eq down = true;
     down eq up   = false;
endtype

type BEAM is Boolean
sort beam
opns high, low :  -> beam
     _eq_ :  beam, beam -> Bool
eqns ofsort Bool
     high eq high = true;
     high eq low  = false;
     low eq high  = false;
     low eq low   = true;
endtype
```

Figure 10.8: Beam and Shield Data Types

```
type ERROR is Boolean
sorts errnum
opns err53, err54 :  -> errnum
endtype
```

Figure 10.9: Error Data Type

The translated specification is given in figures 10.10, 10.11, 10.12 and 10.13. We note that after performing the transformation the specification becomes four times longer, corresponding to the four permutations of the data parameters.

```
TREATMENTLowDown  =
    xr; XRAYLowDown [] el; ELECTRONLowDown [] d_Low_Down; stop
TREATMENTLowUp    =
    xr; XRAYLowUp [] el; ELECTRONLowUp [] d_Low_Up; stop
TREATMENTHighDown =
    xr; XRAYHighDown [] el; ELECTRONHighDown [] d_High_Down; stop
TREATMENTHighUp   =
    xr; XRAYHighUp [] el; ELECTRONHighUp [] d_High_Up; stop
```

Figure 10.10: Therac Specification II in Basic LOTOS: Treatment

```
ELECTRONLowDown   = hide lb, hb, ls, hs in
                    (FIRELowDown >> TREATMENTLowDown) [] TREATMENTLowDown
ELECTRONLowUp     = hide lb, hb, ls, hs in
                    (FIRELowUp >> TREATMENTLowUp) [] TREATMENTLowUp
ELECTRONHighDown = hide lb, hb, ls, hs in
                    (FIREHighDown >> TREATMENTHighDown) [] TREATMENTHighDown
ELECTRONHighUp    = hide lb, hb, ls, hs in
                    (FIREHighUp >> TREATMENTHighUp) [] TREATMENTHighUp
```

Figure 10.11: Therac Specification II in Basic LOTOS: Electron

```
    FIRELowDown  = ZAPLowDown
    FIRELowUp    = ERROR55
    FIREHighDown = ERROR54
    FIREHighUp   = ZAPHighUp

    ZAPLowDown   = fire_Low_Down; exit
    ZAPLowUp     = fire_Low_Up; exit
    ZAPHighDown  = fire_High_Down; exit
    ZAPHighUp    = fire_High_Up; exit

    ERRORerr_54  = err_54; exit
    ERRORerr_55  = err_55; exit
```

Figure 10.12: Therac Specification II in Basic LOTOS: Remainder

Proving the Radiation Machine is Safe

Now that the state of the beam and the shield is explicit, a bad trace is simply expressed as one which contains the event `fire_High_Down`. The conjecture to be proved in PAM (if the machine

```
XRAYLowDown =
  hide lb, hb, ls, hs in
  (   TREATMENTLowDown
   [] hb; (   TREATMENTHighDown
           [] hs; (   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb; (   TREATMENTLowUp
                                              [] ls; TREATMENTLowDown))))))

XRAYLowUp =
  hide lb, hb, ls, hs in
  (   TREATMENTLowUp
   [] hb; (   TREATMENTHighUp
           [] hs; (   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb; (   TREATMENTLowUp
                                              [] ls; TREATMENTLowDown))))))

XRAYHighDown =
  hide lb, hb, ls, hs in
  (   TREATMENTHighDown
   [] hb; (   TREATMENTHighDown
           [] hs; (   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb; (   TREATMENTLowUp
                                              [] ls; TREATMENTLowDown))))))

XRAYHighUp =
  hide lb, hb, ls, hs in
  (   TREATMENTHighUp
   [] hb; (   TREATMENTHighUp
           [] hs; (   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb; (   TREATMENTLowUp
                                              [] ls; TREATMENTLowDown))))))
```

Figure 10.13: Therac Specification II in Basic LOTOS: Xray

is unsafe) is

```
(testok; exit cred hide events in UNSAFETESTLowDown) = true
```

where

```
UNSAFETESTLowDown = TREATMENTLowDown
                    |[fire_High_Down, d_Low_Down, d_Low_Up, d_High_Down, d_High_Up]|
                    TEST
TEST              = fire_High_Down; testok; exit
events            = lb, hb, ls, hs, xr, el, err, d_Low_Down, d_Low_Up,
                    d_High_Down, d_High_Up, fire_Low_Down, fire_Low_Up,
                    fire_High_Down, fire_High_Up
```

This test process is slightly different from the original Basic LOTOS test of section 9.2: here the `STARTUP` process is dispensed with, as we can specify that treatment begins with the beam low and the shield down by using the process `TREATMENTLowDown`.

To prove the safety of this machine we use the method of section 9.2.4, i.e. unfold the top level process until we get an expression which refers recursively to itself, and check that the bad event does not occur in the unfolding. Note that in this example we have a set of mutually recursive equations, therefore each one will have to be unfolded and checked for occurrences of the bad event. This makes the proof longer, and a little more tedious, than before. Note that we have to complete the proof by hand, as we expect that the conjecture above *does not* hold; we merely use PAM to unfold the expression **hide** `events` **in** `UNSAFETEST`

In order to allow certain expressions to be reduced and/or unfolded in a different way, the following new axioms are added to the usual set.

```
x |[Sort(x)]| stop = stop
hide A in hide A in x = hide A in x
(x [] y) |[s]| z = (x |[s]| z) [] (y |[s]| z)
```

where `Sort(x)` calculates the language of the process. These PAM axioms are derived from laws in the LOTOS standard and are mentioned in appendix D.2. The need for these axioms was discovered in an earlier attempt to perform the proof.

The proof proceeds by unfolding all process expressions of the form `TREATMENT`*bs*, where *bs* stands for `LowDown`, `LowUp`, `HighDown` or `HighUp`. In the unfoldings of the proof below, the following shorthands are used, `UNSAFETEST`*bs* for `TREATMENT`*bs* `|[fire_High_Down]| TEST`, `delta_events` for {`d_Low_Down, d_Low_Up, d_High_Down, d_High_Up`}, and, finally, `all_events` for `therac_events union fire_events union delta_events`. As in the examples in chapter 9, the axioms used to transform the expressions are noted on the right hand side.

```
    hide all_events in UNSAFETESTLowDown
```
$\equiv_{wbc}$ *(unfold)*
```
    hide all_events in
       TREATMENTLowDown
       |[delta_events union fire_High_Down]| fire_High_Down; testok; exit
```

```
≡wbc                                                                    (expansion)
      xr;(   UNSAFETESTLowDown
          [] i; (    UNSAFETESTHighDown
                [] i; (   UNSAFETESTHighUp
                      [] fire_High_Up; (    UNSAFETESTHighUp
                                       [] i; (  UNSAFETESTLowUp
                                             []i; UNSAFETESTLowDown)))))
          [] el; (fire_Low_Down; UNSAFETESTLowDown [] UNSAFETESTLowDown)
```

We also need to unfold each of UNSAFETESTLowUp, UNSAFETESTHighDown and UNSAFETESTHighUp.

```
    UNSAFETESTLowUp
≡wbc
    hide hb, hs, lb, ls in TREATMENTLowUp
       |[delta_events union fire_High_Down]| fire_High_Down; testok; exit
≡wbc
       xr; (   UNSAFETESTLowUp
           [] i; (   fire_High_Up; (   UNSAFETESTHighUp
                                   [] i; (UNSAFETESTLowUp [] i; UNSAFETESTLowDown))
                 [] UNSAFETESTHighUp))
    [] el; (err_55; UNSAFETESTLowUp [] UNSAFETESTLowUp)


    UNSAFETESTHighDown
≡wbc
    hide hb, hs, lb, ls in TREATMENTHighDown
       |[delta_events union fire_High_Down]| fire_High_Down; testok; exit
≡wbc
       xr; (   UNSAFETESTHighDown
           [] i; (   UNSAFETESTHighUp
                 [] fire_High_Up; (UNSAFETESTHighUp [] i; UNSAFETESTLowDown))))
    [] el; (err_54; UNSAFETESTHighDown [] UNSAFETESTHighDown)


    UNSAFETESTHighUp
≡wbc
    hide hb, hs, lb, ls in TREATMENTHighUp
       |[delta_events union fire_High_Down]| fire_High_Down; testok; exit
≡wbc
       xr; (   UNSAFETESTHighUp
           [] fire_High_Up; (   UNSAFETESTHighUp
                            [] i; (UNSAFETESTLowUp [] i; UNSAFETESTLowDown))))
    [] el; (fire_High_Up; UNSAFETESTHighUp [] UNSAFETESTHighUp)
```

By examining these unfoldings it can be seen that none have the capability to perform a fire_High_Down action, and that the machine must therefore be safe.

Summary and Discussion

Completion of this example has shown us that, while it may be *possible* to transform a full LOTOS specification into a Basic LOTOS one, the Basic LOTOS specification is so much larger that the proof becomes extremely tedious, especially as the proof is not fully automated. In the original paper, [Tho94], the LOLA tool was able to simulate the full LOTOS process, and identify duplicate states and therefore conclude that the bad test was rejected, i.e. the machine was safe. Certainly in this case, the LOLA simulation/testing tool seems more appropriate than our method of proof using PAM. On the other hand, PAM has a nice graphical interface and is easy to use, despite the process expressions being presented all on one line, with no indentation showing process structure. The interface to LOLA is text based, and although indentation is used to aid readability of the processes, other features of LOLA such as rather terse on-line help, and lots of confusing information produced during the analysis, make LOLA more difficult to use.

We also note here that while performing the transformation some errors in the original description of [Tho94] were found. The `FIRE` process was incorrect in that it never allowed the high beam to be fired, even if the shield was up. This meant that the machine was incapable of delivering an xray treatment, and could only perform electron treatments. This error was due to an oversight in the specification of the guard conditions of the `FIRE` process. A further problem is that although the mechanism is set up to detect errors (either firing at the wrong level or the beam becoming weak), no actions are taken when an error is detected. This means that once an error occurs the machine is livelocked: the operator can select a treatment, but the machine refuses to deliver that treatment, allowing the operator to select another treatment instead. Obviously the original author made a decision to model only certain aspects of the problem, and this was one of the features that were ignored.

We now go on to explore a different approach to transforming a full LOTOS specification into a Basic LOTOS one.

10.3.3 Using Contexts

Obviously each of the abstract interpretation approaches of section 10.3.1 has its problems; what may be useful is to use a combination of both techniques, preserving information of finite data types and throwing away information about infinite data types. This may allow some verification to be carried out. There is another problem however: the second transformation may be suitable where data types are simple, e.g. the naturals, characters, integers, etc., but what happens when more "complex" data types are added, e.g. stacks, queues, i.e. adts with more structure? It quickly becomes impossible to deal with gates which are composed of gate names and the entire contents of a stack, for example. We may also encounter a similar problem for gates at which many values are exchanged, e.g. `g!3 ?x !2(n + 1) ?y`. A further problem of the second transformation is that common data types such as stacks and queues are typically infinite. Obviously, for the first transformation these are not problems, as the data is ignored; however, the first transformation throws away too much data to be useful in general, which means we have to find a solution to the problems of the second transformation.

This section describes a method, presented in [BK91, Bri92], of turning complex data types into simple ones. The method is general, as the transformation is derived from the definition of the data types. Data types are not completely replaced by processes; the aim is to produce processes having a maximum of one data parameter and only simple data types. Below we refer only to [BK91]; this is a technical report which contains more detail than the published version of the work [Bri92].

The method is presented here as it provides a means of deriving a process algebra oriented specification from an adt oriented specification. The two specifications are weak bisimulation congruent (or possibly strong bisimulation equivalent, depending on which derivation rules are used). As with the two approaches above, the main reason for using this derivation is that it is typically easier to reason about Basic LOTOS specifications than full LOTOS ones. We illustrate the use of the method by taking `Stack_1` of section 10.2.2 and deriving a weak bisimulation congruent process algebra stack; the process algebra stack happens to be the `Stack_2` of section 10.2.3. Thus the derivation also supports our earlier proof, using a different technique, that `Stack_1` $\equiv_{wbc}$ `Stack_2`. In the original paper, four examples of common data types are transformed from ACT ONE specifications into process algebra specifications; the bag, the queue, the stack and the set. The version of stack presented in [BK91] is slightly different from our `Stack_1`, therefore presentation of this example is not merely a repeat of the work in [BK91].

We begin by presenting the main ideas and some technical background of the method of [BK91].

The Technicalities

The method is based on *contexts*. A context can be imagined as a LOTOS behaviour expression with a number of holes in it. In the definition of weak bisimulation congruence in section 3.4.3 and in the definition of the **cred** relation in section 3.5.3 contexts with only one hole were used to express the congruence property, i.e. that congruent processes should behave in the same way in all environments. The main theorem of [BK91] is that given an appropriate context equation, where the context is the unknown, it is possible to construct a process algebra description of that context to perform the function of the data type, thus removing the need for the ACT ONE specification of the data type.

A context may be characterised by *transductions*; in the same way that process states are related by transitions, contexts are related by transductions. In general, transductions are written

$$C - [a/b] \rightarrow C'$$

and should be read as "the context C evolves into C' by producing an a action, which may interact with the outside environment, and consuming a b action from the process inside". A transduction $C - [a/b] \rightarrow C'$ corresponds to an SOS inference rule of the form

$$X \xrightarrow{b} X' \ \vdash \ C[X] \xrightarrow{a} C'[X']$$

Contexts may be composed, written $C \circ D[X]$, which means $C[D[X]]$.

The main theorem of [BK91] describes a method for constructing a process corresponding to a context based on the information contained in the transductions of that context. The only transductions considered are those where $b = 0$, which means that no action is produced internally, i.e. C performs a *independently*, and $b = a$, which means that the action a is passed through the context. The process, p_C, which represents the context C, is built as follows

$$\mathtt{p_C} := [\!]\ \{a'; \mathtt{p_{C'}} \mid C - [a/0] \rightarrow C'\}\ [\!]\ \ [\!]\ \{a; \mathtt{p_{C'}} \mid C - [a/a] \rightarrow C'\}$$

and the following equation holds

$$C[X] \sim (\mathtt{p_C}\ \ |[\mathtt{M}]|\ X)[\mathtt{S_M}]$$

where $\mathtt{S_M}$ is a renaming function which turns primed actions back into unprimed actions. The actions are primed because, if the context performs an action independently, then the action must be temporarily renamed to ensure it does not accidentally synchronise with actions from X. On the other hand, if the context is merely passing an action on from the inner process, it performs the action in parallel with the inner process; this has the desired effect because of the multi-way synchronisation of LOTOS.

The equation is phrased in terms of strong bisimulation equivalence, but of course the weaker relations may be used in place of $\sim$.

Stack Example

We start with the full LOTOS stack, `Stack_1`, as given in figure 10.3. In this definition, the processes are parameterised by values of type `Stack_Type`. The aim of the transformation is to reduce these stack parameters to simple data types. A suitable context equation for this example is

$$C_x[\mathtt{Used\_Stack\_1(s)}] \sim \mathtt{Used\_Stack\_1(Push(s,x))}$$

i.e. the context models the last element pushed onto the stack, the *top* element, ignoring the rest of the stack (which is modelled by the process inside the context).

Given this context equation, we define the transductions of the context on which we can then use the method of [BK91] to generate the process modelling that context. The process of defining the transductions is a weak point of the method, as expressing the most appropriate transductions is highly dependent on the skill of the user. It is important to realise that several ways of expressing the transductions might be possible, and that not all transductions allow the process corresponding to the context to be easily generated.

We define two transductions for the initial case, $s = New$:

$$\begin{array}{lll} C & -[\texttt{push?x}/0] \rightarrow & C_x \circ C \\ C & -[\texttt{empty!true}/0] \rightarrow & C \end{array}$$

Since we know that the stack is empty, only the actions **push?x** and **empty!true** are possible. We also define four transductions for the general case, $s \neq New$:

$$\begin{array}{lll} C_x & -[\texttt{push?y}/0] \rightarrow & C_y \circ C_x \\ C_x & -[\texttt{pop}/0] \rightarrow & I \\ C_x & -[\texttt{top!x}/0] \rightarrow & C_x \\ C_x & -[\texttt{empty!false}/0] \rightarrow & C_x \end{array}$$

where I denotes the identity context, with transductions $I - [a/a] \rightarrow I$ for all a. Since the context models the top of the stack we know that x is at the top of the stack and that the stack is not empty, allowing the **top!x** and **empty!false** actions.

In all of these transductions none of the actions are generated by the process inside the context C_x since this process is just the bottom part of the stack and is unaffected by operations to the top of the stack. After **top** or **empty** the context remains unchanged and after **pop** action the context behaves like the identity context, i.e. it allows all actions to pass through unchanged since after x is popped from the top of the stack there is nothing more to do. Lastly, if a new element is pushed onto the stack, the context behaves like the context for the new element composed with the old context, i.e. the old context is kept around until it is needed again.

The problem with transforming these equations according to the theorem is that the last equation gives composed contexts of arbitrary length, implying that each process has arbitrarily complex arguments. To solve this problem in [BK91], a new theorem is introduced which deals specifically with contexts of the form $C - [a/0] \rightarrow C'$ and $I - [a/a] \rightarrow I$. The new theorem, expressed using $\equiv_{wbc}$ rather than $\sim$, allows identity contexts to be interpreted as **exit** and sequential composition to be used in place of parallelism. Applying this theorem gives the following solution for $C_x[X]$:

```
Cx[X] := (  push?y; q(x,y)
         [] pop; exit
         [] top!x; Cx[X]
         [] empty!false; Cx[X]) >> X
```

where $q(x,y) = C_y \circ C_x[X]$.

Of course the criteria for the transformation have still not been met in that, although only simple data types are now used, the process q has *two* arguments. $q(x,y)$ can be further decomposed by further examination of the original transductions for C_x and also of the above process, $C_x[X]$. Two transitions can be applied to $C_x[X]$ according to these sources:

$$C_x[X] \overset{\text{push?y}}{\longrightarrow} C_y \circ C_x[X]$$
$$C_x[X] \overset{\text{push?y}}{\longrightarrow} q(x,y) \gg X$$

(distributing $\gg$ through $[]$). The right hand side states are equivalent by definition since our processes are deterministic, i.e.

$$q(x,y) \gg X \equiv_{wbc} C_y \circ C_x[X]$$

The right hand side of this equation can be transformed by rewriting the composition of contexts, and applying associativity of $\gg$ and the definition of $C_x[X]$.

$q(x,y) \gg X \equiv_{wbc}$
  ((**push?z**; $q(y,z)$ [] **pop**; **exit** [] **top!y**; $C_y \circ C_x[X]$ [] **empty!false**; $C_y \circ C_x[X]$)
   $\gg$ (**push?z**; $q(x,z)$ [] **pop**; **exit** [] **top!x**; $C_x[X]$ [] **empty!false**; $C_x[X]$))
   $\gg X$

Since X can be an arbitrary process, we can take it to be **exit** and use a weak bisimulation congruence $P \gg$ **exit** $\equiv_{wbc} P$ to further transform this expression. This equivalence holds by the following argument.

If P terminates with **exit**, then by law **(d2a)** of the standard the final **exit** of $P \gg$ **exit** becomes **i**; **exit**, which can be further reduced to give **exit** (since there must be actions in P we can apply law **(m1)**), therefore $P \gg$ **exit** reduces to P. In this case we know that $q(x,y)$ will eventually terminate, therefore use of this equivalence is sound. We note that in [BK91] the process **stop** was (incorrectly) used rather than **exit**. By applying the weak bisimulation congruence laws to $P \gg$ **stop** we get the actions of P followed by **exit** $\gg$ **stop** which is the same as **stop**. Effectively we have turned the final **exit** of P into **stop**. Obviously the two process do not have the same behaviour, i.e. $P \gg$ **stop** $\not\equiv_{wbc} P$.

So, by taking X to be **exit** we get

$q(x,y) \equiv_{wbc}$
 ((**push?z**; $q(y,z)$ [] **pop**; **exit** [] **top!y**; $C_y \circ C_x[X]$ [] **empty!false**; $C_y \circ C_x[X]$)
 $\gg$ (**push?z**; $q(x,z)$ [] **pop**; **exit** [] **top!x**; $C_x[X]$ [] **empty!false**; $C_x[X]$))

and, prefixing each side by **push?y**,

```
push?y; $q(x,y)$ $\equiv_{wbc}$
  push?y;
    ((push?z; $q(y,z)$ [] pop; exit [] top!y; $C_y \circ C_x[X]$ [] empty!false; $C_y \circ C_x[X]$)
    $\gg$ (push?z; $q(x,z)$ [] pop; exit [] top!x; $C_x[X]$ [] empty!false; $C_x[X]$))
```

i.e. push?y; $q(x,y)$ is a solution of

```
$X(x)$ $\equiv_{wbc}$
  push?y; (($X(y)$ [] pop; exit [] top!y; $C_y \circ C_x[X]$ [] empty!false; $C_y \circ C_x[X]$)
              $\gg$ ($X(x)$ [] pop; exit [] top!x; $C_x[X]$ [] empty!false; $C_x[X]$))
```

By the uniqueness of solutions to recursive process we know that the above expression represents only one process, therefore we name it Unit (x).

```
Unit(x) $\equiv_{wbc}$
  push?y; ((Unit(y) [] pop; exit [] top!y; $C_y \circ C_x[X]$ [] empty!false; $C_y \circ C_x[X]$)
              $\gg$ (Unit(x) [] pop; exit [] top!x; $C_x[X]$ [] empty!false; $C_x[X]$))
```

One more step gives us the final solution to our transformation. Let Top(x) be the process corresponding to $C_x[X]$ given earlier, then from the solution to push?y; $q(x,y)$ given above by the process Unit we get the following for Top (x):

```
    Top (x) := (  Unit (x)
               [] pop; exit
               [] top!x; Top (x)
               [] empty!false; Top (x))
```

remembering that X = exit. From this we can then express Unit as

```
    Unit (x) := push?y; Top (y) $\gg$ Top (x)
```

By the same method as above, we define a process corresponding to $C[X]$, the context which models the empty stack. Since only two transductions are defined for $C[X]$ we only have two branches in the process.

```
$C[X]$ :=     (push?x; ($C_x \circ C[X]$)
          []  empty!true; $C[X]$) $\gg X$
```

As above, we let X = exit, and call $C[X]$ Stack, giving

```
Stack =     push?x; (Top (x) >> Stack)
        []  empty!true; Stack
```

Note that our derived process, Stack, is weak bisimulation congruent to the original abstract data type oriented stack, Stack_1, by definition of the derivation, but also to Stack_2. With some manipulation, substituting Unit into Top, reversing the process of distributing >> through [] and changing process names, the process Stack above is syntactically identical to the process Stack_2 of figure 10.4.

This example shows that it is possible to derive a process algebra specification from an abstract data type; the method also gives us that the two specifications are equivalent, either by strong bisimulation equivalence, or weak bisimulation congruence, depending on which theorems of [BK91] are used.

10.3.4 Related Work

In [Led87] a different approach to verifying full LOTOS specifications is taken: the idea is to separate the descriptions involving adts from the purely process descriptions. The specification of the system as a whole is obtained by composing the adt and process algebra descriptions using parallelism. The verification of the system then proceeds on the assumption that the correctness of the whole implementation with respect to the specification can be deduced from the correctness of the parts. Correctness here is measured by the **red** preorder of [BSS87]. This relation, rather than weak bisimulation congruence for example, is chosen because of the properties of the transformation used to convert a process dependent on data to one independent of data. In [Led87] a rather peculiar transformation from data dependent to data independent processes is used; only the guards of the data dependent process are considered, the event offers and process parameters being ignored. This excludes two important sources of information about the possible value of a variable; presumably this approach is taken in order to make the transformation simpler. The transformation does not preserve the branching structure of the original process as branches whose guards are false, and thus are not executed if data information is preserved, are retained. The introduction of internal events to model the nondeterminism of choices in which a number of guards are true precludes the possibility of using weak bisimulation congruence. Given the transformation of section 10.3.1, which behaves more as expected, this approach to the data of a full LOTOS specification seems very strange.

Apart from the odd transformation from full LOTOS to Basic LOTOS used, this approach has three main problems: firstly, the equivalence of processes involving data has still to be evaluated, and it is not made clear whether or not this is possible. Secondly, as demonstrated in chapter 7, it is not always possible to deduce anything about the correctness of the whole from the correctness of the parts, or rather, it may not be possible to prove anything about the correctness of the parts. Thirdly, it seems that the approach only works in the special cases where the two concerns can be separated, and therefore relies on the specifier adopting this fairly restrictive approach. In many specifications, the data has an important part to play in the flow of control of the process, and is intrinsic to the process, therefore it cannot easily be moved to a separate process.

10.4 Summary and Discussion

In this chapter the problems of performing verification on full LOTOS specifications have been examined and several possible solutions have been explored. We presented a modified version of the approach used in chapter 9 and attempted to show (using hand proofs) that an abstract data type oriented version of the Stack was equivalent to a process algebra oriented version of the Stack. Although successful for `Stack_1` $\equiv_{wbc}$ `Stack_2`, we could not show `Stack_1` $\equiv_{wbc}$ `Stack_3`.

The proof technique used successfully for `Stack_1` $\equiv_{wbc}$ `Stack_2` seems too complex and the proofs too tedious to be carried out reliably (if at all) on other (larger) examples.

A problem we encountered in attempting to prove `Stack_1` $\equiv_{wbc}$ `Stack_3` was that LOTOS is not a rich enough language to be able to express the properties we wish to show. The next chapter proposes the use of logic to specify system properties as a solution to this problem. Meanwhile, in this chapter we continued by investigating the possibility of transforming the data information in a full LOTOS specification into a process, allowing us to continue using the equational reasoning paradigm. Although we can already see that equational reasoning does not provide an ideal setting for verification of full LOTOS specifications, it is interesting to see exactly how far we can go before the method breaks down completely. The methods investigated rely on first transforming the full LOTOS specification into a Basic LOTOS specification, and carrying out any verification on the Basic LOTOS specification. We presented the approaches of two other authors to this problem. The methods were illustrated by a version of the radiation machine example and the Stack example respectively. We also investigated the properties of the transformations produced by one of the methods.

It can be seen that each approach on its own does not offer a satisfactory solution to the problem of verification of Full LOTOS specifications. In particular, abstract interpretation either throws away too much information, or results in an explosion in the size of the specification, and the contexts approach relies heavily on the ingenuity of the person doing the transformation (although the main data types are covered in [BK91]). However, it may be possible to obtain useful results if the different approaches are used together.

For example, a specification of a telephone system is presented in [FLS90]. This specification uses three types of data: sets, enumerated types, including booleans, and natural numbers. The natural numbers represent the telephone numbers, so we could restrict them to a finite portion of the naturals to allow the verification to be carried out. Booleans can be treated like an enumerated type with two values. Two other enumerated types are used: one to denote the state of the telephone, *Ok* or *Busy*, and the other to denote different phases of the connection process, e.g. *dial*, *ring*, *connect* and so on. In the (constraint-oriented) specification the latter are used mainly to constrain synchronisations. All enumerated types can be replaced by applying the second transformation, which will preserve all data. Finally, the sets can be implemented as processes by using context equations to derive the appropriate behaviour. Having performed these transformations, various properties of the system could be checked, such as it is impossible to call two numbers at once, the number dialled is the number connected to, and so on.

This example demonstrates how the approaches to full LOTOS specifications can be used together to simplify the specification, hopefully retaining enough information that the results proved on the simplified specification can also be applied to the original system.

The next chapter abandons the equational reasoning approach to verification, looking instead at the possibility of using temporal or modal logic to describe the requirements of a system.

Chapter 11

Verification Requirements III: Temporal and Modal Properties

11.1 Introduction

Since chapter 4 we have considered only the approach to verification of comparing two LOTOS specifications using equational reasoning. We saw that this approach was beginning to break down in the case study of chapter 7, in the attempt to axiomatise the **cred** preorder in section 8.4, and again when we tackled full LOTOS in chapter 10. More specifically, in the case study we had to dramatically alter the form of the specification in order to prove that it was equivalent to the implementation; in expressing the **cred** preorder in PAM we had to resort to axiomatising **cred** as an equivalence, which is not generally sound; and in the full LOTOS studies we discovered that the introduction of data leads to an unacceptable increase in the complexity of the proof technique.

A feature common to all these problems is the notion of *partial specification.* The original case study specification, i.e. the protocols, was partial with respect to the system described by the implementation, but neither equivalence relations nor the **cred** preorder could express this relationship; instead we had to strengthen the specification.

Although the **cred** preorder expresses something of the notion of partial specification, namely the reduction of nondeterminism, it was too strong for the case study. In addition, as showed in section 8.4.1, we are unable to implement it soundly in the equational reasoning paradigm without losing much of the power of the original relation.

As for the problems with full LOTOS, we could alleviate the complexity of proofs involving data types by using partial specifications of the properties of the system, thereby simplifying the specification and hopefully also the proofs. As can be seen in the above examples, we are unable to use process algebra and equivalence relations to express the notion of partial specification directly.

In chapter 2 we indicated that we might sometimes want to use a different formalism to describe the requirements of a system; this may also solve the problems of partial specification described above. In this chapter we discuss the advantages and disadvantages of using logic in that role.

Logic is a non-constructive specification formalism; this means the ordering of events is specified implicitly, while safety and liveness properties are specified explicitly. We are particularly interested in modal and temporal logics, e.g. [HM85, MP92, Koz83]. Some of these logics have another useful property, already referred to in section 3.4.3, which is that they provide alternative characterisations for the various equivalences on processes. This means that if two processes are equivalent, they satisfy the same modal formulae, and if they are not equivalent, there is at least one formula satisfied by one process and not the other. If the logic has this property we say that it is *adequate* with respect to the equivalence.

The proof technique usually associated with logic is model checking. Model checking allows us to determine whether a given formula holds in the model. In particular, we may use process algebra to express that model. Algorithms exist which automate model checking; the user is not required to intervene. (Therefore model checking does not depend on the skill of the user for its success, unlike the system developed in chapter 8 which is highly reliant on the skills of the user.)

We begin our study of the use of temporal or modal logics with LOTOS by surveying the current state of use of temporal logics for LOTOS; only two approaches are known to the author, [FGL89, FGR90] and [DFGR92]. Neither of these is suitable for our purposes; the reasons are given below. In contrast to this small literature on logic for LOTOS is the large body of work on logics for CCS; in particular, for HML (Hennessy-Milner Logic) and its various extensions, including modal mu-calculus. A very simple proof technique [SW90] can be used to show that a CCS process satisfies a property expressed in one of these logics. Moreover, the logics and proof technique do not rely on the syntax of CCS; they are based on labelled transition systems. For this reason, we consider the use of the modal mu-calculus with Basic LOTOS.

We present HML and the modal mu-calculus in some detail, together with descriptions and examples of the sorts of properties we might use these logics to specify. We continue by illustrating the application of logic to two of the problem examples mentioned above, showing how logic can solve the problems introduced by partial specifications. Finally, we conjecture that it might be possible to extend this logic and proof techinique for use with full LOTOS and give a sketch of this together with some examples. We do not pursue the topic here; it is to be the subject of a SERC-funded research project, "Temporal Aspects of Verification of LOTOS Specifications", at the University of Glasgow.

11.2 Temporal Logics and LOTOS

In [FGL89] a temporal logic for Basic LOTOS is described. The aim of [FGL89] is to give a compositional temporal logic semantics for Basic LOTOS, this allows equality of processes and satisfiability of formulae to be expressed in terms of temporal logic formulae. In this approach each operator of LOTOS corresponds to a logical formula and the meaning of a process is given by its *characteristic formula*, written $\chi(p)$. The ability of a process p to satisfy a particular formula ϕ, written $p \models \phi$, can then be expressed as $\chi(p) \Rightarrow \phi$. The logic defined is *expressive* with respect to trace equivalence over processes, which means that $p \equiv_{trace} q$ is expressed as $q \models \chi(p)$ (or vice versa).

The logic is adequate with respect to trace equivalence, but, as discussed previously, trace equivalence is too weak for most verification purposes because liveness properties are not preserved. The main problem of this approach, therefore, is that the logic is too weak. Other problems are that the proof technique can get very complicated because it involves (possibly nested) fixed points and that the **i** action is not given its special status as an unobservable action.

This work is extended in [FGR90], in which the logic CTL* [CES86] and its relationship to a subset of Basic LOTOS, namely action prefix, choice and recursion, is considered. The main result of [FGR90] is that while it is possible to give a compositional temporal logic semantics which is adequate with respect to strong bisimulation equivalence for this subset of LOTOS, no such logic exists for a larger subset of LOTOS. In order to obtain a logic of this strength for Basic LOTOS, either the compositionality of the semantics must be abandoned or the logic must be strengthened by adding new operators.

The author knows of no other work on the use of a temporal logic with LOTOS (either full or Basic LOTOS), although some related work may be found in [DFGR92, LITE]. In [DFGR92] a proof technique is described which allows formulae expressed in ACTL (action based CTL) to be checked against the model provided by a labelled transition system, expressed in either CCS or MEIJE. The underlying model checker is based on CTL and Kripke structures (state based transition systems), and the method defines two translations, one from labelled transition systems to Kripke structures, and the other from ACTL to CTL. This proof technique is implemented for LOTOS in the LITE toolkit [LITE]. Unfortunately, the translations described by [DFGR92] only work in one direction; this means that if two processes are not equivalent, the system cannot express this information in terms of the labelled transition systems and ACTL. The user must learn to work in the world of CTL and Kripke structures or lose the information supplied in the case of inequivalence of processes. Such information is one of the advantages of using logic-based systems, therefore it seems a shame to lose it.

In direct contrast to the very small literature for logic and LOTOS is the large volume of literature concerned with logic and CCS. This proliferation of work on CCS and logic may in fact be the reason that this topic has not been pursued specifically in relation to LOTOS.

The most commonly used logic for CCS, HML (Hennessy-Milner Logic), is defined over labelled transition systems, therefore there should be no reason why HML cannot be applied to Basic LOTOS. Indeed, HML has already been used indirectly for LOTOS in [EFJ90], where the tool Caesar is used to provide a labelled transition system which is modified (by changing the event names) by hand and then entered into the Concurrency Workbench for model checking. This application of HML to LOTOS shows that the only difficulty with using HML for LOTOS lies in the lack of a suitable proof technique, or rather proof tool. Obviously it is not acceptable to have to modify every example by hand in order to use the Concurrency Workbench.

To show that a formula Φ holds for a particular process p we must show p belongs to the set of processes satisfying Φ. One way of doing this is to construct that set and then test for p being a member. Obviously this method involves a lot of unnecessary work since p will typically not be the only process satisfying Φ. Fortunately, other methods for checking a process satisfies a formula exist. Safety properties over finite state systems can be checked by reachability analysis, but this technique suffers from the state explosion problem. Other methods use proof systems built from inference rules which allow the truth of a statement to be deduced from the truth of previously proved statements; there are several examples of such systems. Methods which make use of the structure of the syntax of the process include [Sti87, AW92]; others, such as [SW90, BS90, Lar90a], rely on the structure of the labelled transition system. Obviously this latter class of proof techniques is more useful to us since it is independent of CCS syntax.

Not all these proof techniques can be easily automated. Those intended for automation include the technique described in [Lar90a], which is automated in TAV [GLZ89], and the tableau method of [SW90], a version of which is automated in [CPS89]. These systems are fully automatic, but do not support infinite transition systems; however, the tableau method of [SW90] is extended to infinite transition systems in [BS90]. This extension is automated in the proof assistant [Bra92].

In the rest of this chapter we consider the use of an extension of HML, the modal mu-calculus, for LOTOS.

11.3 Introducing HML and its Variants

Modal logics are interpreted over labelled transition systems. In addition to the usual propositional logic operators, we also have modalities expressing transitional change. We begin by presenting a simple logic, a very slight extension of HML [HM85].

The syntax of the logic is:

$$\Phi ::= \text{tt} \mid Z \mid \neg\, \Phi \mid \Phi_1 \wedge \Phi_2 \mid [K]\Phi$$

where $K \subseteq Act$. In the original HML only single actions are allowed in the modalities.

Operators of this logic are the usual boolean operators (tt, $\neg$ and $\wedge$), plus $[K]$, the modal operator "always", where $[K]\Phi$ means that Φ holds after every performance of all actions in K.

In addition to the main operators of the language, a number of dual operators can be defined; see figure 11.1. The new modal operator $\langle K\rangle$ is called "possible"; $\langle K\rangle\Phi$ means that Φ holds after *some* performance of any action in K.

$$\begin{array}{lcl} \text{ff} & \overset{def}{=} & \neg\text{tt} \\ \Phi_1 \vee \Phi_2 & \overset{def}{=} & \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\ \langle K\rangle\Phi & \overset{def}{=} & \neg[K]\neg\Phi \end{array}$$

Figure 11.1: Additional Defined Operators for the Modal Mu-Calculus

The modal operators $[K]$ and $\langle K\rangle$ are analogous to the **must** and **may** testing relations of chapter 3. Each operator expresses the necessity or possibility, respectively, of an action being performed.

Abbreviations which are useful in formulae are: $[-]$ to denote $[Act]$ and $[-K]$ to denote $[Act - K]$. Similarly, $\langle -\rangle$ denotes $\langle Act\rangle$ and $\langle -K\rangle$ denotes $\langle Act - K\rangle$.

One problem with the above logic is that it cannot express invariants over an infinite computation. For example, given the process $P = a; P$ we can express the property of P being capable of performing an a action by $P \models \langle a\rangle\text{tt}$. However, we cannot express the property that P is always able to perform the action a; the nearest we can get is an infinite formula of the form $P \models \langle a\rangle\text{tt} \wedge [a](\langle a\rangle\text{tt} \wedge [a](\langle a\rangle\text{tt} \wedge \ldots))$. To fully express this property we need to extend the logic. One method is to introduce infinitary connectives, but these are undesirable if automated verification is planned. An alternative solution is to use modal equations to express properties; the property above then becomes $P \models Z = \langle a\rangle\text{tt} \wedge [a]Z$, i.e. Z says a is possible, and after all a transitions, Z holds. This equation has a number of solutions/fixed points; the smallest being the empty set and the largest being the set containing the process P.

The smallest solution to a modal equation is not always the empty set. Another example of a modal equation is $Z = \langle b\rangle\text{tt} \vee [a]Z$ which says either b is possible, or after all a transitions Z holds. The smallest solution gives us processes which perform a finite number of a actions followed by a b action, while the largest solution can perform an infinite number of a actions and b need never occur. This formula is also known as *until* in some logics because we are expressing the property that a holds until b holds.

As we have seen, the smallest and largest solutions to a modal equation can specify quite different properties, therefore in practice we will find it convenient to be able to indicate which of the two is required. If we continue to use the equational format to express such properties, indicating whether we want the largest or the smallest solution may become cumbersome, especially when the formula involves more than one equation. Rather than introduce equations to the language, we introduce fixed point operators, one for the least solution to the equation, one for the greatest. These two solutions are guaranteed to exist, due to the properties of the operators of the logic and Tarski's theorem.

The resulting logic is called the *modal mu-calculus* and it was first used in conjunction with process algebras in [Lar90b]. Modal mu-calculus can also be described as a propositional branching time temporal logic. It will be formally introduced below and used for all the examples following.

11.3.1 The Modal Mu-Calculus

The syntax of the modal mu-calculus is the same as that of HML given above, with the addition of $\nu Z.\Phi$ where Z ranges over propositional variables, and the removal of tt. We define tt $\stackrel{def}{=} \nu Z.Z$.

The new operator, $\nu Z.\Phi$, denotes the greatest fixed point over Φ (the maximal solution to the equation $Z = \Phi$). This expression has the syntactic restriction that each free occurrence of Z in Φ must lie within the scope of an even number of negations (this constraint allows the application of Tarski's theorem, guaranteeing the existence of least and greatest fixed points).

Just as we defined $\vee$ to be the dual of $\wedge$ and $\langle K\rangle$ to be the dual of $[K]$ we can define a dual of ν which denotes the least fixed point (minimal solution) of an equation. It is called μ, and is defined as follows:

$$\mu Z.\Phi \stackrel{def}{=} \neg\nu Z.\neg\Phi[Z := \neg Z]$$

where $\Phi[Z := \neg Z]$ is the formula obtained by substituting $\neg Z$ for each free occurrence of Z in Φ.

The semantics of the modal operators is given in figure 11.2.

$$\begin{array}{lcll}
||Z||_{\mathcal{V}} & = & \mathcal{V}(Z) & (Z \in Var) \\
||\neg\,\Phi||_{\mathcal{V}} & = & \mathcal{P} - ||\Phi||_{\mathcal{V}} & \\
||\Phi_1 \,\wedge\, \Phi_2||_{\mathcal{V}} & = & ||\Phi_1||_{\mathcal{V}} \,\cap\, ||\Phi_2||_{\mathcal{V}} & \\
||[K]\Phi||_{\mathcal{V}} & = & \{P \in \mathcal{P} : \forall k \in K.\forall P'.P \xrightarrow{k} P' \Rightarrow P' \in ||\Phi||_{\mathcal{V}}\} & \\
||\langle K\rangle\Phi||_{\mathcal{V}} & = & \{P \in \mathcal{P} : \exists k \in K.\exists P'.P \xrightarrow{k} P' \Rightarrow P' \in ||\Phi||_{\mathcal{V}}\} & \\
||\nu Z.\Phi||_{\mathcal{V}} & = & \bigcup\{\mathcal{E} \subseteq \mathcal{P} : \mathcal{E} \,\subseteq\, ||\Phi||_{\mathcal{V}[\mathcal{E}/Z]}\} & \\
||\mu Z.\Phi||_{\mathcal{V}} & = & \bigcap\{\mathcal{E} \subseteq \mathcal{P} : ||\Phi||_{\mathcal{V}[\mathcal{E}/Z]} \,\subseteq\, \mathcal{E}\} &
\end{array}$$

Figure 11.2: Interpretation of Modal Formulae

Formulae of the logic are interpreted over the model $(\mathcal{T},\mathcal{V})$, where $\mathcal{T}$ is a labelled transition system and $\mathcal{V}$ is a valuation assigning a set of processes to each variable, $\mathcal{V}(Z) \subseteq \mathcal{P}$ for each Z. The valuation may be updated, written $\mathcal{V}[\mathcal{E}/Z]$, to give $\mathcal{V}'$, where $\mathcal{V}' = \mathcal{V}$ except at Z, where

$\mathcal{V}'(Z) = \mathcal{E}$. The notation $||\Phi||^{\mathcal{T}}_{\mathcal{V}}$ is used to denote the set of processes satisfying Φ. Usually the $\mathcal{T}$ is dropped, since it will be obvious from the context.

A process p satisfies a property Φ, written $p \models \Phi$, if and only if $p \in ||\Phi||_{\mathcal{V}}$, for some valuation $\mathcal{V}$. Modal mu-calculus is *adequate* with respect to observation/weak bisimulation equivalence.

In the interpretations of figure 11.2, the labelled transition system $\mathcal{T}$ has the transition relation $\longrightarrow$; the relation $\Longrightarrow$ may also be used, allowing silent actions to be ignored. The modal operators are then written $[\![K]\!]$ and $\langle\!\langle K\rangle\!\rangle$, and are defined as follows:

$$\begin{aligned} ||[\![K]\!]\Phi||_{\mathcal{V}} &= \{P \in \mathcal{P} : \forall k \in K.\forall P'.P \overset{k}{\Longrightarrow} P' \Rightarrow P' \in ||\Phi||_{\mathcal{V}}\} \\ ||\langle\!\langle K\rangle\!\rangle\Phi||_{\mathcal{V}} &= \{P \in \mathcal{P} : \exists k \in K.\exists P'.P \overset{k}{\Longrightarrow} P' \Rightarrow P' \in ||\Phi||_{\mathcal{V}}\} \end{aligned}$$

The power of the logic is unchanged by this use of $\Longrightarrow$ instead of $\longrightarrow$, because both $[\![K]\!]$ and $\langle\!\langle K\rangle\!\rangle$ can be defined in terms of $[K]$ and $\langle K\rangle$; however, these forms are often more convenient to use. Note that now the modalities may also include $[\![\epsilon]\!]$ and $\langle\!\langle\epsilon\rangle\!\rangle$, where ϵ is the empty string. These formulae indicate the occurrence of a sequence of internal actions.

Above we claimed that it is possible to use the modal mu-calculus in conjunction with Basic LOTOS because its definition relies on labelled transition systems rather than process algebra syntax. To demonstrate this, we present several examples of the use of mu-calculus for Basic LOTOS in the next two sections. We also describe classifications of properties, giving common examples from each class.

11.4 Classes of Properties

It is useful to be able to classify various properties for two reasons. First, the classifications supply templates for formulae which are used frequently in specifications; second, as mentioned in chapter 2, the classification can be used as a measure of the completeness of our specification, e.g. typically a specification contains representatives of each class. One such classification is known as the *safety/liveness* classification [Lam77]. This classification contains two almost disjoint classes, which can be informally described as:

safety "nothing bad" happens, or *invariance*, i.e. some property on states holds continuously.

liveness "something good" happens, or *response*, i.e. requests are eventually dealt with.

The only formula common to both classes is the atomic formula tt. This classification is nice because of its simple and intuitive definition.

A different classification, put forward in [MP89], gives a hierarchy of classes distinguished by syntactic structure. Each class is also associated with a distinct proof technique. (No such clear cut relations exist between the classes of safety and liveness and syntactic structure and/or proof technique.) However, the classification of [MP89] does not encompass all expressible properties, although the authors claim that all the properties generally required are included.

Below we give a few examples of the sort of properties which may be expressed in the modal mu-calculus in order to illustrate what we might expect to be typical usage of the logic with respect to LOTOS processes; the presentation reflects the division of properties into the classes of safety and liveness, and also the existence of several subclasses. The examples are small, and already familiar from earlier chapters. The specific problem examples we drew attention to in the introduction to this chapter will be dealt with in the next section.

We begin with safety properties.

11.4.1 Safety Properties

Capacity This class is the set of properties which say that a process may perform some action, i.e. the process is capable of performing the action, although it won't necessarily perform it. Formulae of this subclass contain the operator $\langle K \rangle$.

For example, given a buffer

```
Buffer := in; out; Buffer
```

we may say that once the buffer receives an input, then it is capable of an output. This can be expressed using logic.

$$\texttt{Buffer} \models [\mathbf{in}]\langle \mathbf{out} \rangle \mathrm{tt}$$

After all transitions labelled by **in**, the action **out** is possible.

Necessity This class expresses the property that an action not only can occur, but that it must occur. For example, once the buffer receives an input, then that item will be output.

$$\texttt{Buffer} \models [\mathbf{in}](\langle \mathbf{out} \rangle \mathrm{tt} \wedge [-\mathbf{out}]\mathrm{ff})$$

After all transitions **in**, the action **out** is possible and all other actions are impossible (since ff can never be satisfied).

Global Invariants The particular formula for a global invariant will depend on the system under consideration; however, we can say that in general global invariants are expressed using a greatest fixed point. This is because, as a safety property, we want to say that the invariant holds forever, i.e. over a (possibly infinite) sequence of actions.

A particular example of a global invariant is *freedom from deadlock.* We begin by expressing what it means for a process to be deadlocked, i.e. incapable of performing an observable action.

$$Deadlock \overset{def}{=} [-\epsilon]\mathrm{ff}$$

Although deadlock can be good (if viewed as termination), usually we interpret deadlock as *unsuccessful* termination. To say that a process is never deadlocked, it must satisfy:

$$\nu Z.\neg Deadlock \wedge [-]Z$$

Yet another global invariant commonly encountered is the property of *mutual exclusion.*

Mutual Exclusion Given two processes and a shared resource, we may wish to ensure that only one process at a time may access that resource, i.e. the processes must exclude each other from their *critical section* (the part where they use the resource).

This was described using process algebra in the Readers and Writers example of section 9.3, where the mutually exclusive actions were reading and writing (from/to a part of memory). We may specify that reading and writing never occur at the same time using logic:

$$\mathrm{Spec} \overset{def}{=} \nu Z.([\mathtt{rb}][\mathtt{wb}]\mathrm{ff} \wedge [\mathtt{wb}][\mathtt{rb}]\mathrm{ff} \wedge [-]Z)$$

This formula says that the sequences of actions `rb,wb` and `wb,rb` are not permitted at any stage, since no process can satisfy the formula ff. Compare this with the process algebra specification of section 9.3, where it was necessary to specify the actions which *could* occur. Using logic allows us to specify directly that actions may *not* occur.

To say that the processes implementing the system, as given in figure 9.9 of section 9.3, satisfy this formula, we write `Impl` $\models$ Spec.

We also want to look at liveness properties.

11.4.2 Liveness Properties

We mentioned above that it is a useful general principle that safety properties are best described by maximal solutions because these express properties which hold over infinite computations. On the other hand, liveness properties are best described by minimal solutions because these express properties which hold over finite computations. (Remember that liveness properties say something good eventually happens, i.e. within a finite amount of time.)

Response A response property usually has the form "if some event occurs, then eventually we get a response to that event". Using the buffer example above, a response property might be that after an `in` action, there will eventually be an **out** action (although we may first have a finite sequence of some other actions):

$$\texttt{Buffer} \models [\mathbf{in}]\mu Z.(\langle\mathbf{out}\rangle\mathrm{tt} \vee [-]Z)$$

We would also want to say that this formula is true for every occurrence of `in`, so we wrap the above response formula up in a greatest fixed point, giving:

$$\texttt{Buffer} \models \nu Y.([\mathbf{in}]\mu Z.(\langle\mathbf{out}\rangle\mathrm{tt} \vee [-]Z) \wedge [\mathbf{out}]Y)$$

Termination The most obvious liveness property is that the process eventually terminates. This can be expressed by saying the process *converges*, i.e. may only perform a finite number of actions.

$$\mathit{Converges} \stackrel{def}{=} \mu Z.[-]Z$$

The opposite of this is *divergence*, i.e. the process is capable of performing silent actions forever.

$$\mathit{Diverges} \stackrel{def}{=} \nu Z.\langle\mathbf{i}\rangle Z$$

Liveness/Livelock Freedom We may also want to express the absence of divergence by saying that the process is *live*.

$$\mathrm{Live} \stackrel{def}{=} [\![\epsilon]\!]\langle\!\langle -\epsilon\rangle\!\rangle\mathrm{tt}$$

$$\mathrm{Liveness} \stackrel{def}{=} \mu Z.\mathrm{Live} \wedge [\![-]\!]Z$$

In other words, there may be a finite number of internal actions, but eventually an observable action will occur.

Above we have given a very small selection of properties which can be expressed in the modal mu-calculus. In the next section we explore further the greater suitability of logic for expressing certain system properties by considering how logic can solve the partial specification problems mentioned in the introduction.

11.5 Using Logic For Partial Specifications

We began this chapter by describing the problems encountered in the equational reasoning approach to verification when considering partial specifications. Logic now provides us with a means of expressing a partial specification of system properties, and the $\models$ relation allows us to express that an implementation satisfies that specification. We illustrate this by reconsidering the case study example of chapter 7 and the radiation machine of section 9.2.

11.5.1 Login Case Study

Consider the first protocol of the case study and our failure to show that equation 7.5 held with any interpretation of "satisfies"; see page 101. The problem we encountered there was the introduction of nondeterminism by use of the **hide** operator; the specification used deterministic choice, but the implementation (with events hidden) used nondeterministic choice. If we had a way of relating the specification and implementation which automatically ignored events in the implementation not specified by the specification, we wouldn't have to use **hide** and this problem wouldn't arise. Using logic allows this; it also allows us to specify this protocol without specifying deterministic or nondeterministic choice between `p1` and `n1`, something we cannot avoid when using process algebra.

The first protocol, `P1`, is specified as a response property:

$$[\texttt{m1}]\mu Z.(\langle -\texttt{m1},\texttt{p1},\texttt{n1}\rangle Z \ \vee\ \langle \texttt{p1}\rangle \texttt{tt} \ \vee\ \langle \texttt{n1}\rangle \texttt{tt})$$

After all occurrences of `m1` we have a finite sequence of actions which does not include `m1`, `p1` or `n1`; but eventually either `p1` or `n1` will occur.

This formula really applies to the once-only version of the case study in chapter 7. For the recursive version, given in section 9.1, we want to be able to say that this formula holds for every occurrence of `m1`. As with the buffer example given in the previous section, we wrap up the response property above in a greatest fixed point.

$$\nu Y.(([\texttt{m1}]\mu Z.(\langle -\texttt{m1},\texttt{p1},\texttt{n1}\rangle Z \ \vee\ \langle \texttt{p1}\rangle \texttt{tt} \ \vee\ \langle \texttt{n1}\rangle \texttt{tt})) \ \wedge\ [-]Y)$$

To express that the implementation of the system satisfies this property we write:

$$\texttt{Processes} \models \nu Y.(([\texttt{m1}]\mu Z.(\langle -\texttt{m1},\texttt{p1},\texttt{n1}\rangle Z \ \vee\ \langle \texttt{p1}\rangle \texttt{tt} \ \vee\ \langle \texttt{n1}\rangle \texttt{tt})) \ \wedge\ [-]Y)$$

We anticipate that it should be straightforward to use the tableau method of [SW90] to show that this expression holds.

We can express the other protocols similarly, therefore constructing the conjecture and proof of correctness in three parts corresponding to the three protocols as we originally thought possible in section 7.3. Using process algebra and **cred** (or any equivalence relation) this was not possible because we had to explicitly hide events of the implementation which the specification did not consider. Here logic allows us to say some events occur, without specifying *which* events occur, or even how many of them occur.

We can also express some other properties of the case study which were added as constraints to the specification, such as property that the positive response **p1** hinges on the transmission of an **m5** event.

$$[\mathtt{m5}]\mu Z.((\langle -\mathtt{p1},\mathtt{n1}\rangle Z \ \vee\ \langle \mathtt{p1}\rangle \mathrm{tt}) \ \wedge\ \langle \mathtt{n1}\rangle \mathrm{ff})$$

As above, we can wrap this up in a greatest fixed point to express that the formula holds repeatedly, and show that it is satisfied in the model **Processes**.

The case study example also illustrates the feature of constraint-oriented specification that liveness properties are not preserved by parallel composition of processes. When we combined the protocols in chapter 7 using parallelism we suddenly were specifying much more than intended; the liveness properties had changed. The relationship between constraint-oriented specification and safety and liveness was discussed in [Bri89], where he concluded that liveness properties are only preserved by composition if the constraints are consistent, i.e. $P' \models \Phi$ may only be deduced from $P \models \Phi$ if the traces of P' are all included in the traces of P. This was obviously not the case with the protocols, and therefore liveness properties were not preserved.

11.5.2 The Radiation Machine

The safety requirement of the radiation machine example of section 9.2 provides an example of a global invariant. We note that the term *safety* is overloaded; we mean specifically that life is endangered by faulty operation of the machine; however, this property also happens to be expressed as "nothing bad happens".

We wish to express the requirement that a **fire** event can never occur when the shield is low and the beam is high. Since we cannot easily say in process algebra that an action cannot occur, we must instead specify the converse. We did this in section 9.2 by specifying the general form of bad traces of the system, i.e. traces in which the **fire** event occurs when the beam is high and the shield is low. This sort of property can be specified by a process. We can then show using the **cred** relation that the machine either satisfies or does not satisfy that trace, and is therefore unsafe or safe respectively. For an unsafe machine this means showing an expression of the form *(A* **cred** *B) = true* holds; however, in the case of a safe machine we have a problem because this requires showing an expression of the form *(A* **cred** *B) = false* holds. As discussed in section 8.4, this is not possible in our proof system. Since logic can express that an event cannot occur, we solve both of these problems using logic.

The good trace of the radiation machine example can be expressed as follows:

```
((not (hb | hs))*; hb; (not (lb | hs | fire))*; (lb | hs))*
```

Initially, we can have any actions except **hs** and **hb**. As soon as an **hb** occurs we may then have any actions except **lb**, **hs** and **fire**. If either **lb** or **hs** occur, we loop round the expression again. Effectively, the danger zone is when an **hb** event has occurred and before either a **lb** or a **hs** occurs, so we block the bad event, **fire**, during that period.

We can now express the good trace as a logical formula

$$\textit{good-therac} \overset{def}{=} \nu Z.((\langle -\mathtt{hb},\mathtt{hs}\rangle \mathtt{Z} \ \vee \ [\mathtt{hb}]\nu \mathtt{Y}.(((\langle -\mathtt{lb},\mathtt{hs},\mathtt{fire}\rangle \mathtt{Y} \ \vee \ [\mathtt{lb},\mathtt{hs}]\mathtt{Z}) \ \wedge \ [\mathtt{fire}]\mathrm{ff}))$$

The fixed point operators in this formula correspond roughly to occurrences of the $*$ in the trace formula above.

The safety requirement of the radiation machine is fully expressed by

$$\mathtt{Therac} \models [\mathtt{lb}][\mathtt{ls}]\textit{good-therac}$$

the extra [lb][ls] reflecting the initial set up of the machine. This has an equivalent effect to the process algebra expression of section 9.2, i.e.

((testok; exit) cred THERACTEST) = false

An extension of the radiation machine example is to add data types; this was done in section 10.3.2. The next section considers how we might use the modal mu-calculus with *full* LOTOS.

11.6 The Modal Mu-Calculus and Full LOTOS

Above we have claimed that it is straightforward to use the modal mu-calculus with Basic LOTOS because both are based on transition systems. In particular, the same proof techniques can be used as long as we choose one which relies only on the transition system, e.g. [SW90], and not on the syntactic structure of the process, although these could perhaps be used by applying the translation from finite Basic LOTOS to CCS given in [BIN92].

We saw in the chapter on full LOTOS that partial specification could be a way of alleviating the complexity introduced by the addition of data types; we now consider extending the modal mu-calculus for use with full LOTOS. The following discussion merely speculates on the sort of extensions required; this topic is to be taken up as further work.

11.6.1 Extending the Modal Mu-Calculus

The first part of the extension is to adapt the definitions of the operators of the modal mu-calculus and the rules of the proof system to take account of data values in the transitions; it is just a matter of drawing transition labels from the set of structured actions rather than plain gate names.

This is similar to the way in which the equivalence relations of Basic LOTOS are extended for use with full LOTOS; see section 10.1.3.

So, for example, in the semantics of $[K]$,

$$||[K]\Phi||_{\mathcal{V}} = \{P \in \mathcal{P} : \forall k \in K.\forall P'.P \xrightarrow{k} P' \Rightarrow P' \in ||\Phi||_{\mathcal{V}}\}$$

the actions in K can now structured; $k = gw$, where g is a gate name and w is a list of data values. In evaluating formulae we must now consider gate names *and* data values.

We illustrate the use of the new logic by extending the buffer example given earlier; we can now add data to the formulae. The new buffer is described as

```
Buffer := in?x; out!x; Buffer
```

and we can express the capability and necessity to output the same data as was input.

$$\texttt{Buffer} \models \nu Z.([\mathbf{in?x}]\mu Y.(\langle \mathbf{out!x}\rangle \mathrm{tt} \vee [-]Y) \wedge [-]Z)$$

$$\texttt{Buffer} \models \nu Z.([\mathbf{in?x}]\mu Y.(([\mathbf{out!x}]\mathrm{tt} \wedge [-\mathbf{out!x}]\mathrm{ff}) \vee [-]Y) \wedge [-]Z)$$

The occurrence of `?x` is a binding occurrence for `x` in these formulae, therefore if the labelled transition system contains the label `in?3` the value 3 will be substituted for `x` in the remainder of the formula. Since the buffer is defined recursively this happens every time the event `in?x` occurs.

These formulations of the properties of the buffer seem a little clumsy, especially for recursively defined processes, in which case we want to look at just one unfolding of the labelled transition system and deduce from that and looping in the transition system that the formula holds for all `x`. To express this, the logic needs to be strengthened by *quantification*; we want to be able to write

$$\forall \mathtt{x}.(\texttt{Buffer} \models \nu Z.([\mathbf{in?x}]\mu Y.(\langle \mathbf{out!x}\rangle \mathrm{tt} \vee [-]Y) \wedge [-]Z))$$

Introducing quantification seems less straightforward than the introduction of data, and will not be discussed further here.

Assuming the logic has been successfully extended along the lines described above, it is useful only if we have a corresponding proof technique.

11.6.2 Extending the Proof Technique

We claim it is possible to use the proof technique of [SW90] for the modal mu-calculus in conjunction with Basic LOTOS processes and to extend it for use with full LOTOS. Below we give a sketch of the proof technique and possible extensions.

The proof technique is a tableau system comprising several inference rules for building the tableau based on the operators of the *logic*, and conditions that allow us to evaluate whether or not a tableau is successful. A successful tableau indicates that the formula holds for the given labelled transition system.

For example, consider the rule for the $[K]$ modality.

$$\frac{P \vdash_{\Delta} [K]\Phi}{P' \vdash_{\Delta} \Phi} \quad \{P' \mid \forall k \in K.P \xrightarrow{k} P'\}$$

where the subscript Δ stands for a definition list relating variable names to formulae which is used to unroll occurrences of fixed point operators.

Given a tableau ending with $P \vdash_{\Delta} [K]\Phi$, the top line above, the rule tells us how to build the next level in the tableau. In this case we must investigate all possible K transitions of the labelled transition system P, using the next state in the labelled transition system as the model in the next line of the tableau. As might be expected, the rule for $\langle K \rangle$ allows us to choose *one* possible transition from P, and therefore one state P', ignoring all the others. This makes tableau construction nondeterministic; we have a choice as to how to construct the next row in the tableau. The rule for $\vee$ also gives us a choice.

We do not give full details of the proof system here, referring the reader to [SW90] or the introductory [Sti91]. We have successfully carried out preliminary studies using this technique to show the validity of some of the formulae of sections 11.4 and 11.5 with respect to LOTOS processes.

Now consider using this proof technique for full LOTOS specifications. Given the rule for $[K]$ as presented above we have to work from a labelled transition system in which all data variables are instantiated, otherwise we cannot know which transitions are really possible. Computing the labelled transition system is time-consuming, and also is not possible if the process is infinite (in which case we want to rely on properties of recursive equations, as mentioned above). The technique of [SW90] has to be modified to support uninstantiated variables in the labelled transition systems. The strongest way of doing this would be to carry around details of the environment (as in the second transformation from full to Basic LOTOS described in section 10.3.1) so that we can work out the exact value attached to a variable. An alternative might be to work from the predicates and guards of the process, maintaining a list of conditions which must hold for the formula as a whole to hold. For example, rather than having an instantiated variable, we might instead have a predicate which tells us that the value of that variable lies within a particular range, e.g. $n > 0$. Again, this topic will be investigated as further work.

We conclude this section with some examples of the way in which we might use an extension of the modal mu-calculus with full LOTOS. This gives us some incentive for overcoming the problems of proving these properties hold with respect to a structured labelled transition system.

11.6.3 Examples

We can express the safety of the radiation machine presented in section 10.3.2 very simply as the inability to perform the action `fire!high!down`:

$$\nu Z.(\langle \texttt{fire!high!down} \rangle \mathrm{ff} \ \wedge \ [-]Z)$$

Contrast this with the fairly complicated specification of safety given earlier for the Basic LOTOS specification. We believe the proof of the above formula would also compare favourably with the complicated proof of section 10.3.2.

Now consider the third stack of section 10.2.5. Although equivalence proofs turned out to be impossible for the third stack in section 10.2.6, we can describe some aspects of the behaviour of the stack as partial specifications using logic.

For example, `New_Stack_3` can only perform `empty!true`, and not `empty!false`,

$$\texttt{New\_Stack\_3} \ \models \ \langle \texttt{empty!true} \rangle \mathrm{tt} \ \wedge \ [\texttt{empty!false}]\mathrm{ff}$$

and it cannot perform `top` or `pop` actions.

$$\texttt{New\_Stack\_3} \ \models \ [\texttt{top!x}]\mathrm{ff} \ \wedge \ [\texttt{pop}]\mathrm{ff}$$

On the other hand, `Used_Stack_3` is unable to perform `empty!true`.

$$\texttt{Used\_Stack\_3(x,n)} \ \models \ [\texttt{empty!true}]\mathrm{ff}$$

Bearing in mind that full LOTOS allows data parameters to the processes, we might also want to use these in our formulae, e.g.

$$\texttt{Used\_Stack\_3(x,n)} \ \models \ \langle \texttt{top!x} \rangle \mathrm{tt} \ \wedge \ [\texttt{top!z}]\mathrm{ff} \quad \text{where } \texttt{x} \neq \texttt{z}$$

Given that `x` is the value at the top of the stack, `top` may only produce the value `x` and no other. In this case it is the occurrence of `x` as a parameter to `Used_Stack_3` which binds the value for the remainder of the formula.

Looking back at the abstract data type equations of figure 10.2 we might also want to express invariants over the behaviour of the stack, such as

$$\langle \mathbf{push?x} \rangle (\nu Z. \langle \mathbf{pop} \rangle \mathrm{tt} \ \wedge \ [-]Z)$$

After a **push** action it is possible to perform a **pop** action. Again we might want quantification in this formula, to express that the formula holds for all states of **Used_Stack_3**, e.g.

$$\forall \mathtt{x}. \forall \mathtt{n}. (\mathtt{Used\_Stack\_3(x,n)}) \ \models \ \langle \mathbf{push?y} \rangle (\nu Z. \langle \mathbf{pop} \rangle \mathrm{tt} \ \wedge \ [-]Z))$$

The topic of extending the modal mu-calculus for use with full LOTOS is beyond the scope of the present work.

11.7 Summary

In this chapter, due to the inadequacy of process algebra and the equational reasoning framework when considering partial specification, we have considered the use of logic as an alternative means of specifying a system. Verification of the system can then be expressed by checking the LOTOS implementation of the system against the logic specification of the system. We specifically considered using the modal mu-calculus for both Basic and full LOTOS.

In order to demonstrate the advantages of using logic we reconsidered the examples which had motivated our study of logic, namely the login case study of chapter 7 and the use of **cred** in the radiation machine of section 9.2. Using logic we showed that it was possible to express the correctness of the case study by splitting the conjecture up into three parts, each corresponding to a protocol, and we claim that the proof is straightforward. Similarly, it was straightforward to specify the good trace of the radiation machine, something which was not possible using process algebra.

The modal mu-calculus can be used with no alteration for Basic LOTOS; however, some extensions must be made when considering full LOTOS. The proof technique must also be extended to allow model checking of full LOTOS specifications. We considered some examples illustrating the sort of extensions which might be necessary to allow reasoning about models containing data types, and also some full LOTOS examples drawn from chapter 10 which illustrate the ease of specifying system properties using logic rather than process algebra.

Logic can express properties which were difficult to express using process algebra, and the proof technique (model checking) is less reliant on the skills of the user than our equational reasoning approach. Nevertheless, this approach is not perfect. The main problem might be that the method of describing the properties is a skilled task, more so than describing processes. We are not unduly concerned by this, as every approach must have some drawback; the area of logic and LOTOS remains a promising one, and will be further researched in the project "Temporal Aspects of Verification of LOTOS Specifications".

Chapter 12

Conclusions

In this thesis we have introduced the topic of verification of properties of concurrent systems, in particular those described using LOTOS, in a manner suitable for those with no prior knowledge of the subject. We followed this with a thorough, practically-based investigation of verification of properties of LOTOS specifications expressed using comparison of two LOTOS specifications by a behavioural relation and equational reasoning.

We developed a partially automated proof technique based on equational reasoning, and used this, together with hand proofs where necessary, to study verification via particular examples. This allowed us to develop a greater understanding of the verification process and also demonstrated the utility of the proof system developed.

The main outcome of our work on equational reasoning and verification of properties of LOTOS specifications is that equational reasoning is highly suitable for carrying out *equivalence* proofs, but that the method begins to break down when partial specifications are considered; we are forced to write clumsy specifications, and were unable to (soundly) automate the proof process. This implies that a different proof paradigm should be adopted when considering *ordering* of specifications.

We investigated one method of dealing with partial specifications: the use of temporal or modal logic for specifications. We do not abandon LOTOS; a LOTOS expression may be used as the model in which we evaluate the validity of the logical specification. We made a preliminary study of the advantages and disadvantages of this approach, illustrating the use of logic for specification by examples drawn from the earlier part of the thesis. We showed how some of the examples for which the equational approach had been unsatisfactory are better treated using logic.

12.1 Detailed List of Achievements

The achievements of the thesis may be considered in four main groups. We began by introducing and surveying the field. This survey gives the necessary background for the main investigation of verification of properties of LOTOS specifications; the investigation had both theoretical and practical elements. During the practical work we made some contributions to the use of term rewriting for automation of process algebra proofs. We concluded by studying the use of logic with LOTOS.

Following this grouping, we list these achievements in more detail.

- We began by providing an introduction to verification of concurrent systems, process algebra, LOTOS, equational reasoning and logic which may be used as a springboard for other researchers entering the field. This makes the thesis self-contained by providing the background necessary for the main investigation of verification of properties of LOTOS specifications.
  - We surveyed the topic of verification of properties of LOTOS specifications. The introductory work comes in chapter 2, where we discuss possible interpretations of the term "verification", and chapter 4, where one particular approach to verification is described.
  - In chapter 3 we presented aspects of the three process algebras CCS, CSP and Basic LOTOS, including equivalence relations and proof techniques. The work is not new, but the presentation of the three together in a comparative manner is.
  - We presented those aspects of equational reasoning relevant to our work with LOTOS, namely proof by rewriting and Knuth-Bendix completion, in chapter 5.
  - In chapter 11 we presented the logics HML and modal mu-calculus.
  - As part of our survey of verification we also surveyed currently available proof tools which might be used with LOTOS. This is mentioned in chapter 4; the survey is given in more detail in appendix A.
  - The syntax and semantics of LOTOS is presented in appendix B. We found the presentation of the same information in the standard [ISO88] rather complex and poorly organised. Our intention was to provide a clearer presentation for ourselves (and we believe this has been achieved); others may also find our presentation easier to follow.
- The bulk of our work was related to the verification of properties of LOTOS specifications where the verification requirement is expressed by a behavioural equivalence and the proof carried out using equational reasoning.
  - We have surveyed and discussed the topic of verification of properties of LOTOS specifications, including two areas which have been largely ignored in the literature, namely

verification of properties of full LOTOS specifications and also the use of logic in specifying the requirements of a system; see chapters 2, 4, 10 and 11.

- As part of the study of choices the user is faced with in the verification process we identified several possible criteria, given in section 4.2.2, which might help differentiate between the various equivalences/preorders.
- We have made a thorough investigation of one aspect of the verification topic, namely the method of comparing two LOTOS specifications in terms of a behavioural equivalence relation. The theoretical part of the study was carried out first for a portion of Basic LOTOS in chapter 4, extended to the complete language of Basic LOTOS, including recursion, in chapter 8, and finally extended to full LOTOS in chapter 10.
- The above approach to verification of properties of Basic LOTOS specifications has been implemented in a term rewriting framework. The initial implementation described in chapter 6 deals only with a subset of the language, but the system has evolved to include all features of Basic LOTOS. The system finally obtained is described in chapter 8. The ease with which it was possible to adapt and develop the system is a consequence of choosing the equational reasoning paradigm. This development also required changing the underlying equational reasoning tool. Note that the relabelling operator of LOTOS is slightly simplified in our implementation.
- The utility of the above proof system has been demonstrated via a number of examples, presented in chapter 9. We deliberately chose examples which had been presented by other authors using different proof systems as a means of avoiding unintentional bias towards examples suited to our proof system.
- When investigating verification of properties of full LOTOS specifications we considered the approaches of other authors to the problem in addition to considering how the above equational reasoning approach could be modified. In particular we studied the transformations from full LOTOS to Basic LOTOS detailed in [Bol92]. We investigated the properties of these transformations with respect to their use in verification of full LOTOS specifications, concluding that the results of verification carried out on the transformed specifications can only be extrapolated to the original full LOTOS specification for one of the transformations. The other transformation preserves very weak properties only. This was not considered in the original presentation of [Bol92]; our contribution is detailed in section 10.3.1.
- An important part of verification is specification; if the possible approaches to verification are borne in mind when specifying a system, the verification may be easier. We also contributed to research on specification in LOTOS.

The LOTOS language was developed for use in specification of communications and this is the area in which it is usually applied. We successfully used LOTOS for non-communications examples in chapter 7 and in chapter 9, showing that the language is applicable outside the originally conceived area of application.

We reviewed the language LOTOS in section 7.6. Of special relevance are the observations that some features of the LOTOS language make verification more difficult, in particular the disable operator, discussed in section 9.2.5, and also the hide operator, discussed in section 7.4.1.

- During the practical investigation of verification we used equational reasoning and term rewriting for automation; this resulted in the following contributions.

  - Early experiments in using a rewriting tool for proofs of equivalence were centered around an attempt to find a confluent and terminating set of rewrite rules for the LOTOS weak bisimulation congruence relation for a subset of the language; this is described in chapter 6. This experiment was successful in that such a rule set was developed, but unsuccessful in that the rule set did not have sufficient power for any but the simplest proofs.

    A complete rule set corresponding to the equivalence of the semantics is impossible to obtain because weak bisimulation is undecidable, therefore we also discussed the relative merits of different choices of rules and how they might affect the verification process, see section 8.3. In particular, we discussed the effects an incomplete set of rules might have on the verification process, and how that might necessitate the introduction of a strategy in applying the rules, see sections 6.4.2 and 8.3.

  - The above completion work had two side effects. The first was the generation of several diverging sequences of rules (useful for work detailed in [Wat92]), see section 6.4.3. The second was to show that the laws of weak bisimulation congruence given in [ISO88] are not sound (although this is easily corrected), see section 5.5.2.

  - The complete set of rules developed above was not powerful enough for any but the simplest examples, a fact easily ascertained by experiment. Our initial solution was to develop a set of rewrite rules which reduce a term according to the expansion law for parallelism; see section 6.4.1. (Note that the final implementation does not use these rules because this facility is built into PAM).

  - Although obtaining a set of rewrite rules for an equivalence relation is just a matter of orienting the axioms or laws, the process is not so simple for a preorder relation. In section 8.4 we presented two possible rule sets for the **cred** preorder, together with analysis of the effects of using these rewrite rules in proofs.

  - Throughout the study various equational reasoning tools were used. The principle of equational reasoning is a simple and familiar one, which makes proof in this paradigm straightforward. Equational reasoning tools, on the other hand, are hard to use on the whole. This is due more to the status of these tools as research tools rather than a pieces of software engineered for industry; see particularly our remarks about RRL in section 7.6. However, we note that PAM, reviewed in appendix A, is also an equational reasoning tool and yet is easy to use. Perhaps its simple graphical interface shows the way of the future for such tools.

- As a result of the shortcomings of the proof system developed above, we identified a need for an alternative approach to specification and verification. We studied the use of logic in specifying the requirements of a system, with a LOTOS specification being used as the model in which those requirements are evaluated.
  - In chapter 11 we presented HML and the modal mu-calculus and proposed that, although defined for use with CCS, they could also be used in conjunction with LOTOS since both are based on the model of labelled transition systems. We outlined a suitable proof technique and gave several examples of the use of the modal mu-calculus in expressing properties of Basic LOTOS specifications. In particular we considered examples from earlier in the thesis for which the equational approach had been unsuitable.
  - The use of the modal mu-calculus for Basic LOTOS seems straightforward, but the modification of the logic for use with full LOTOS may be more difficult. We discussed possible extensions to the modal mu-calculus and proof system required for use with full LOTOS, illustrating those requirements by means of selected examples; see section 11.6.

Our original aims have been only partially met in that we have only thoroughly researched one particular approach to verification of LOTOS specifications. Specifically, we have not fully considered verification of LOTOS specifications with respect to logical requirements specifications (but see further work below). We believe that with respect to verification of equivalence/ordering of two LOTOS specifications we have achieved our original goals, including the goal to present the work as simply and clearly as possible, making our work easily understood by a newcomer to the subject, although this is of course a rather subjective evaluation.

12.2 Further Work

There are two kinds of work discussed here: work which follows on naturally from the work of the thesis, and work which is indirectly related to the main body of the thesis work.

12.2.1 Work Directly Related to the Thesis

This category contains four main topics: development of the PAM proof system, further case studies, further investigation of verification for full LOTOS, in particular methods of automating proofs, and use of modal or temporal logic for full LOTOS.

Developing the PAM System There are several ways in which the proof system could be further developed. The most obvious one is to add axiomatisations for other relations. This could also mean finding a better axiomatisation for the **cred** preorder, although really we cannot properly express preorders in the equational framework.

We remarked in chapter 10 that a new version of PAM which can handle parameterised processes is under development. The addition of parameters to processes, both gate parameters and data type parameters, is important if our system is to be used for real LOTOS verification, therefore we anticipate modifying our approach to utilise these new features.

A further development, which also depends on development of PAM, is implementation of the LOTOS relabelling operator as described in [ISO88], rather than the current simplified version.

More Examples It is clear that we have only attempted fairly small examples in the studies of chapters 7 and 9; although we note that our system was easy to use, and, for most of these examples, proofs were completed quickly. An important question is: can our method be scaled up to deal with larger examples? The easiest way to answer this question is to attempt verification proofs involving larger, more complex specifications.

Development of Proof Techniques for full LOTOS Verification Although we were able to carry out some verification of properties of full LOTOS specifications, these results were not satisfactory; the (hand) proofs were complex and tedious. While some of the difficulty lies in the lack of automated tools, there has also been very little research on verification techniques for full LOTOS. In particular, the method of constructing a bisimulation in the stack proof of section 10.2.4 is very tedious; it would be useful to find a better way, which could be easily automated, to prove two full LOTOS specifications equivalent.

Developing a Modal/Temporal Logic for full LOTOS While the use of the modal mu-calculus for Basic LOTOS is perfectly valid, since both are based on labelled transition systems, it seems much harder to generalise the modal mu-calculus for use with full LOTOS. As seen in chapter 11, the problems lie more in the development of a proof system; we can already formulate properties which use data. This work will be continued in the SERC-funded project "Temporal Aspects of Verification of LOTOS Specifications".

12.2.2 Work Indirectly Related to the Thesis

The remainder of these topics for further work are ones which were encountered during our investigations but which are somewhat tangential to the main body of the work.

Criteria for Choosing a Relation One of the most difficult parts of verification lies in interpreting the verification requirements; a particular aspect of this is choosing which of the many equivalence and preorder relations defined for process algebras is most appropriate for a given example. In section 4.2.2 we postulated a number of possible criteria which might be used in making this choice; these remain to be more thoroughly investigated.

Specification Styles To what extent does the style of specification affect the verification? It is clear that some specifications exclude certain methods; see, for example, the third stack and equivalence proofs of section 10.2.6. A possible direction for future work lies in determining whether such problems can be classified, identified in advance, and avoided.

Nondeterminism In the Login case study example of chapter 7 we originally anticipated that the conjecture expressing correctness and hence the proof could be split into three parts due to the disjoint nature of the protocols. In attempting the subproofs we discovered that the use of the **hide** operator led to extra nondeterminism and the proof could not proceed. We could not prove anything about the correctness of the parts of the conjecture, and therefore nothing could be deduced about the correctness of the system as a whole. However, the technique of divide and conquer as a method of simplifying problems is both commonly used and valuable. Since the use of **hide** causes problems by introducing internal events, the problem may be that we have not as yet found the right method of splitting the conjecture up, or of expressing the correctness of the parts. As we have seen in chapter 11, one approach to this problem involves the use of logic. If we want to remain within process algebra a solution could lie in *relativized bisimulation* [LM92], where bisimulation is measured with respect to an environment which expresses "allowed" actions.

We conclude with a discussion of the possible impact of our work on academics in this field and also on the wider LOTOS community.

12.3 Prospects for this Work

The thesis as a whole may be useful to other researchers getting started in the area of verification of properties of concurrent systems; we provide an introduction to the main topics of this area, a thorough study of the applicability of equational reasoning techniques to such verification, and also preliminary investigations of the use of temporal or modal logic for use with LOTOS. It is important to point out that the area of verification has been largely ignored by the LOTOS community in favour of validation methods such as testing and simulation, therefore few, if any, large scale works on verification of properties of LOTOS specifications, such as our own, exist.

Given that LOTOS is an ISO standard and therefore used by industry, particularly the telecommunications industry, we must also consider the impact of our work on the wider LOTOS community, i.e. outside academia.

Through PAM, our system provides an environment in which to reason about Basic LOTOS which is easy to use and also to extend (only a knowledge of LOTOS is required; the form of the PAM files is straightforward and requires no special coding ability). However, the quality and robustness of tools demanded by industrial practitioners is much higher than we have yet attained; our tool is still in the early stages of development. We require to carry out further tests, particularly on larger examples. In addition, our proof system relies to a large extent on the skill of the user in guiding the proof process, which requires a significant investment in terms of time both in preliminary study and in the proof process.

As long as tool support for LOTOS continues to be concentrated in the areas of simulation, testing and translation, there is little future for verification in the wider LOTOS community. Although verification can give us greater confidence in the correctness of our systems, perhaps, relative to the amount of work required to gain that confidence, the gain is not as great as can be achieved through use of testing and simulation tools which require less effort on the part of the user to obtain results. Nevertheless, we hope that our work will help lead to a greater understanding of verification and the development of better tools and techniques for verification in the future.

Bibliography

[AB84] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:90–131, 1984.

[AB90] G.J. Akkerman and J.C.M. Baeten. Term Rewriting Analysis in Process Algebra. Technical Report P9006, University of Amsterdam, 1990.

[Abr87] S. Abramsky. Observation Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[Aju89] I. Ajubi. Formal Description of the OSI Session Layer: Session Protocol. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 153–210. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Ald89] R. Alderen. COOPER: the compositional construction of a canonical tester. In S. Vuong, editor, *Formal Description Techniques, II*, pages 13–18. Elsevier Science Publishers B.V. (North-Holland), 1989.

[AW92] H.R. Andersen and G. Winskel. Compositional Checking of Satisfaction. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 24–36, 1992.

[BA91] G. Bruns and S. Anderson. The Formalization and Analysis of a Communications Protocol. Technical Report ECS-LFCS-91-137, LFCS, University of Edinburgh, 1991.

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra*. Number 17 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Bai91] J. Baillie. A CCS case study: a safety-critical system. *Software Engineering Journal*, pages 159–167, July 1991.

[BBH$^+$91] J.C. Baeten, J.A. Bergstra, C.A.R. Hoare, R. Milner, J. Parrow, and R. de Simone. The Variety of Process Algebra. Deliverable ESPRIT Basic Research Action 3006, CONCUR (R. Milner and F. Moller, eds.), University of Edinburgh, 1991.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, 1984.

[BIN92] M. Boreale, P. Inverardi, and M. Nesi. Complete sets of axioms for finite basic LOTOS behavioural equivalences. *Information Processing Letters*, 43:155–160, 1992.

[BK84] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1/3):109–137, 1984.

[BK91] E. Brinksma and P. Kars. From Data Structure to Process Structure. Technical Report Memorandum INF-91-38/TIOS-91-11, University of Twente, 1991.

[Bol92] T. Bolognesi, editor. Catalogue of LOTOS Correctness Preserving Transformations. Technical Report Lo/WP1/T1.2/N0045, The LOTOSPHERE Esprit Project, 1992. Task 1.2 deliverable. LOTOSPHERE information disseminated by J. Lagemaat, email `lagemaat@cs.utwente.nl`.

[Boo89] R. Booth. An Evaluation of the LCF Theorem Prover using LOTOS. In S. Vuong, editor, *Formal Description Techniques, II*, pages 83–100. Elsevier Science Publishers B.V. (North-Holland), 1989.

[BR83] S.D. Brookes and W.C. Rounds. Behavioural Equivalence Relations induced by Programming Logics. In *Proceedings of ICALP 83*, LNCS 154, pages 97–108. Springer-Verlag, 1983.

[Bra92] J. Bradfield. A proof assistant for symbolic model checking. Technical Report ECS-LFCS-92-199, University of Edinburgh, 1992.

[Bri88a] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification, VIII*, pages 63–74. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Bri88b] E. Brinksma. *On the Design of Extended LOTOS.* PhD thesis, University of Twente, 1988.

[Bri89] E. Brinksma. Constraint-oriented specification in a constructive formal description technique. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, pages 130–152. Springer-Verlag, 1989. REX School/Workshop, Mook, The Netherlands, May/June 1989.

[Bri92] E. Brinksma. From Data Structure to Process Structure. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 244–254, 1992.

[BS87] T. Bolognesi and S.A. Smolka. Fundamental Results for the Verification of Observational Equivalence: a Survey. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 165–179. Elsevier Science Publishers B.V. (North-Holland), 1987.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In *CONCUR 90*, LNCS 458, pages 115–125, 1990.

[BS94] B. Berthomieu and T. Le Sergent. Programming with Behaviors in an ML framework — The Syntax and Semantics of LCS. In *Proceedings of ESOP*, 1994. To appear in LNCS.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, their Implementations and their Tests. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 349–360. Elsevier Science Publishers B.V. (North-Holland), 1987.

[CCI88] CCITT. *Specification and Description Language (SDL) Recommendations Z.100*, 1988.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. In *ACM TOPLAS*, volume 8, 1986.

[CH90] R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 11–23, 1990.

[CIN91] A. Camilleri, P. Inverardi, and M. Nesi. Combining Interaction and Automation in Process Algebra Verification. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of TAPSOFT 91*, volume II, pages 283–296. Springer-Verlag, 1991.

[Cle89] R. Cleaveland. Tableau-Based Model Checking in the Propositional Mu-Calculus. Technical Report 2/89, University of Sussex, 1989.

[Cle91] R. Cleaveland. On Automatically Explaining Bisimulation Inequivalence. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of CAV 90*, LNCS 531, pages 364–372. Springer-Verlag, 1991.

[CN91] P. Curran and K. Norrie. Specification of an ISO Protocol in LOTOS. Technical report, University of London, 1991.

[CN92] P. Curran and K. J. Norrie. An approach to verifying concurrent systems — a medical information bus (MIB) case study. In *Proceedings of the 5th annual IEEE symposium on computer-based medical systems*, 1992.

[Com91] *The Computer Journal*, 34(1), 1991. Special Issue on Term Rewriting.

[CPS89] R. Cleveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 24–37. Springer-Verlag, 1989.

[CR90] S.J. Colwill and G.H.B. Rafsanjani. Towards Machine-Assisted Formal Validation of LOTOS Specifications. Technical report, British Telecom, 1990.

[De 87] R. De Nicola. Extensional Equivalences for Transition Systems. *Acta Informatica*, 24:211–237, 1987.

[Der82] N. Dershowitz. Orderings for Term Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982.

[DFGR92] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 37–47, 1992.

[DH84] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Dic90] A.J.J. Dick. A Case Study for the ERIL Project. Private communication, 1990.

[Dic91] A.J.J. Dick. An Introduction to Knuth-Bendix Completion. *The Computer Journal*, 34(1):–, 1991. Special Issue on Term Rewriting.

[DIN89] R. De Nicola, P. Inverardi, and M. Nesi. Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Expressions. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 54–67, 1989.

[DIN91] R. De Nicola, P. Inverardi, and M. Nesi. Equational Reasoning about LOTOS Specifications: A Rewriting Approach. In *Proceedings of 6th International Workshop on Software Specification and Design*, pages 148–155. IEEE Press, 1991.

[DMdS90] G. Doumenc, E. Madelaine, and R. de Simone. Proving process calculi translations in ECRINS: The PureLOTOS $\rightarrow$ MEIJE Examples. Technical Report RR 1192, INRIA, 1990.

[DP91] D. Duce and F. Paterno. A Formal Specification of a Graphics Systen in the Framework of the Computer Graphics Reference Model. Technical Report RAL-91-065, Rutherford Appleton Laboratory, September 1991.

[dS85] R. de Simone. Higher-level synchronizing devices in Meije-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[EBB$^+$86] H. Ehrig, J. Buntrok, P. Boehm, F. Nurnberg K-P. Hasler, C. Rieckhoff, and J. de Meer. Towards an Algebraic Semantics of the ISO-Specification Language LOTOS. Technical Report SEDOS/C2/N58, ESPRIT SEDOS Project, 1986.

[EFJ90] P. Ernberg, L. Fredlund, and B. Jonsson. Specification and Validation of a Simple Overtaking Protocol using LOTOS. Technical Report T9006, Swedish Institute of Computer Science, 1990.

[EFP91] P. Ernberg, L. Fredlund, and J. Parrow. An Extended Bibliography of Case Studies. Deliverable ESPRIT Basic Research Action 3006, CONCUR (R. Milner and F. Moller, eds.), University of Edinburgh, 1991.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[Ern91] P. Ernberg. CCS as a method of specification and verification: Analysis of a case study. Technical Report T91:05, Swedish Institute of Computer Science, 1991.

[FGL89] A. Fantechi, S. Gnesi, and C. Laneve. An Expressive Temporal Logic for LOTOS. In S. Vuong, editor, *Formal Description Techniques, II*, pages 261–276. Elsevier Science Publishers B.V. (North-Holland), 1989.

[FGR90] A. Fantechi, S. Gnesi, and G. Ristori. Compositional Logic Semantics and LOTOS. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 365–378. Elsevier Science Publishers B.V. (North-Holland), 1990.

[Fid93] C. Fidge. Comparison of CCS, CSP and LOTOS. Notes from a seminar given at FORTE 93, but not published as part of the proceedings, 1993.

[Fle87] M. Fletcher. The Boyer-Moore Theorem Prover and LOTOS. Research & Technology Memorandum RT62/014/87, British Telecom, Ipswich, 1987.

[FLS90] M. Faci, L. Logrippo, and B. Stépien. Formal Specifications of Telephone Systems in LOTOS. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 25–36. Elsevier Science Publishers B.V. (North-Holland), 1990.

[FO91] L. Fredlund and F. Orava. Modelling Dynamic Communication Structures in LOTOS. In *Formal Description Techniques, IV*, 1991.

[GH91] J.F. Groote and H. Hüttel. Undecidable Equivalences for Basic Process Algebra. Technical Report ECS-LFCS-91-169, LFCS, University of Edinburgh, 1991.

[GL91] S. Gallouzi and L. Logrippo. A Hoare-style Proof System for LOTOS. In J. Quemada, J. Mañas, and E. Vásquez, editors, *Formal Description Techniques, III*, pages 49–62. Elsevier Science Publishers B.V. (North-Holland), 1991.

[GLO91] S. Gallouzi, L. Logrippo, and A. Obaid. An Expressive Trace Theory for LOTOS. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing, and Verification, XI*, pages 159–175. Elsevier Science Publishers B.V. (North-Holland), 1991.

[GLZ89] J.C. Godskesen, K.G. Larsen, and M. Zeeberg. TAV (Tools for Automatic Verification): Users Manual. Technical report, Aalborg University, 1989.

[GM92] J.F. Groote and F. Moller. Verification of Parallel Systems via Decomposition. In W.R. Cleaveland, editor, *CONCUR'92*, LNCS 630, pages 62–76. Springer-Verlag, 1992. Third International Conference on Concurrency Theory.

[Gor88] M.J.C. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Got87] R. Gotzhein. Specifying Abstract Data Types with LOTOS. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 15–26. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Gro87] R. Groenwald. Verification of a sliding window protocol by means of process algebra. Technical Report P8701, University of Amsterdam, 1987.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HJOP89] H. Hansson, B. Jonsson, F. Orava, and B. Pehrson. Specification for Verification. In S. Vuong, editor, *Formal Description Techniques, II*, pages 227–244. Elsevier Science Publishers B.V. (North-Holland), 1989.

[HKK91] M. Hermann, C. Kirchner, and H. Kirchner. Implementations of Term Rewriting Systems. *The Computer Journal*, 34(1):20–33, 1991. Special Issue on Term Rewriting.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985.

[HO82] G. Huet and D.C. Oppen. Equations and Rewrite Rules - A Survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems.* Academic Press, 1982.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Hüt91] H. Hüttel. Silence is Golden: Branching Bisimilarity is Decidable for Context-Free Processes. Technical Report ECS-LFCS-91-173, LFCS, University of Edinburgh, 1991.

[IN90] P. Inverardi and M. Nesi. A Rewriting Strategy to Verify Observational Congruence. *Information Processing Letters*, 35:191–199, 1990.

[ISO74] International Organisation for Standardisation. *The Reference Model for Open Systems Interconnection*, 1974.

[ISO88] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[ISO90] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — Estelle — Formal Description Technique Based on an Extended State Transition Model*, 1990.

[IYK90] H. Ichikawa, K. Yamanaka, and J. Kato. Incremental Specification in LOTOS. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 183–196. Elsevier Science Publishers B.V. (North-Holland), 1990.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[Kir91] C. Kirkwood. An Experiment using Term Rewriting Techniques for Concurrency. In S.L. Peyton-Jones, G. Hutton, and C.K. Holst, editors, *Functional Programming, Glasgow 1990*, pages 196–200. Springer-Verlag, 1991. Extended abstract.

[Kir92] C. Kirkwood. A Case Study for the ERIL Project. Technical Report 1992/R4, University of Glasgow, 1992.

[Kir93] C. Kirkwood. Automating (Specification $\equiv$ Implementation) using Equational Reasoning and LOTOS. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development*, LNCS 668, pages 544–558, 1993.

[KN90] C. Kirkwood and K. Norrie. Some Experiments using Term Rewriting Techniques for Concurrency. Technical Report CSD-TR-623, Royal Holloway and Bedford New College, 1990.

[KN91] C. Kirkwood and K. Norrie. Some Experiments using Term Rewriting Techniques for Concurrency. In J. Quemada, J. Mañas, and E. Vásquez, editors, *Formal Description Techniques, III*, pages 527–530. Elsevier Science Publishers B.V. (North-Holland), 1991. Extended Abstract.

[Koz83] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KS83] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.

[KS90] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[KZ87] D. Kapur and H. Zhang. *RRL : Rewrite Rule Laboratory User's Manual*, 1987. Revised May 1989. Available by anonymous ftp from `herky.cs.uiowa.edu`.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lan89] St. Lange. Towards a Set of Inference Rules for Solving Divergence in Knuth-Bendix Completion. In K.P. Jantke, editor, *Proceedings of Analogical and Inductive Inference 89*, LNCS 397, pages 304–316. Springer-Verlag, 1989.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 87–98. Elsevier Science Publishers B.V. (North-Holland), 1990.

[Lan92] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, 1992.

[Lar86] K.G. Larsen. *Context-Dependent Bisimulation between Processes*. PhD thesis, University of Edinburgh, 1986.

[Lar90a] K.G. Larsen. Modal Specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 232–246, 1990.

[Lar90b] K.G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theoretical Computer Science*, 72:256–288, 1990.

[Led87] G.J. Leduc. The Intertwining of Data Types and Processes in LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 123–136. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Lin91] H. Lin. PAM User Manual (Version 0.6). Technical Report 9/91, University of Sussex, 1991.

[Lin92] H. Lin. PAM : A Process Algebra Manipulator. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 136–146, 1992.

[LITE] M. Caneve and E. Salvatori, editors. LITE User Manual. Technical Report Lo/WP2/N0034/V08, The LOTOSPHERE Esprit Project, 1992. LOTOSPHERE information disseminated by J. Lagemaat, email `lagemaat@cs.utwente.nl`.

[LM92] K.G. Larsen and R. Milner. A Compositional Protocol Verification Using Relativized Bisimulation. *Information and Computation*, 99:80–108, 1992.

[LT91] K.G. Larsen and B. Thomsen. Partial specifications and compositional verification. *Theoretical Computer Science*, 88:15–32, 1991.

[LX90] K.G. Larsen and L. Xinxin. Compositionality Through an Operational Semantics of Contexts. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP 90)*, LNCS 443, pages 526–539, 1990.

[Mad92] E. Madelaine. Verification Tools from the CONCUR Project. *EATCS Bulletin*, 47, 1992.

[MFV89] C. Miguel, A. Fernández, and L. Vidaller. LOTOS Extended with Probabilistic Behaviours. *Formal Aspects of Computing*, 5(3):253–281, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[Mil85] G. Milne. Circal and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7:270–298, 1985.

[Mil89a] R. Milner. A Complete Axiomatisation for Observation Congruence of Finite-state Behaviours. *Information and Control*, 81(2):227–247, 1989.

[Mil89b] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[MM92] W. Mao and G.J. Milne. An Automated Proof Technique for Finite-State Machine Equivalence. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 233–243, 1992.

[Mol90] F. Moller. The importance of the left merge operator in process algebras. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP 90)*, LNCS 443, pages 752–764, 1990.

[Mol91] F. Moller. The Edinburgh Concurrency Workbench (Version 6.0). Technical Report LFCS-TN-34, LFCS, University of Edinburgh, 1991.

[MP89] Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 201–284. Springer-Verlag, 1989. REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1988.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Volume 1: Specification*. Springer-Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[MT90] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proceedings of CONCUR 90*, LNCS 458, pages 401–415, 1990.

[MV89] E. Madelaine and D. Vergamini. Auto: A verification tool for distributed systems using reduction of finite automata networks. In S. Vuong, editor, *Formal Description Techniques, II*, pages 61–66. Elsevier Science Publishers B.V. (North-Holland), 1989.

[MV91a] E. Madelaine and D. Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of CAV 90*, LNCS 531, pages 353–363, 1991.

[MV91b] E. Madelaine and D. Vergamini. Specification and Verification of a Sliding Window Protocol in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, volume C-2 of *IFIP Transactions*. Elsevier Science Publishers B.V. (North-Holland), 1991.

[Naj87] E. Najm. A Verification Oriented Specification in LOTOS of the Transport Protocol. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 181–203. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Nes92] M. Nesi. A Formalization of the Process Algebra CCS in Higher Order Logic. Technical Report 278, University of Cambridge Computer Laboratory, 1992.

[New42] M.H.A. Newman. On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[OP91] F. Orava and J. Parrow. An Algebraic Verification of a Mobile Network. Technical Report R9102, Swedish Institute of Computer Science, 1991. To appear in FACS.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science, 5th GI Conference*, LNCS 104, pages 167–183, 1981.

[Par88] J. Parrow. Verifying a CSMA/CD-protocol with CCS. In S. Aggerwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification, VIII*, pages 373–384. Springer-Verlag, 1988.

[Ple87] U. Pletat. Algebraic Specifications of Abstract Data Types with CCS: An Operational Junction. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 361–372. Elsevier Science Publishers B.V. (North-Holland), 1987.

[PT87] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[QAF90] J. Quemada, A. Azcorra, and D. Frutos. A Timed Calculus for LOTOS. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*. Elsevier Science Publishers B.V. (North-Holland), 1990.

[QFA89] J. Quemada, D. Frutos, and A. Azcorra. TIC: A TImed Calculus. *Formal Aspects of Computing*, 5(3):224–252, 1989.

[Raf92] G.H.B. Rafsanjani. A Data Type Specification for the Process Part of Basic LOTOS — An Axiomatic Semantics. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*, pages 321–332. Oxford University Press, 1992.

[RS91] S. Ramanathan and G. Sivakumar. Rewrite Systems for Protocol Specification and Verification. In J. Quemada, J. Mañas, and E. Vásquez, editors, *Formal Description Techniques, III*, pages 79–94. Elsevier Science Publishers B.V. (North-Holland), 1991.

[RvB91] N. Rico and G. v. Bochmann. Performance description and analysis for distributed systems using a variant of LOTOS. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing, and Verification, XI*, pages 199–213. Elsevier Science Publishers B.V. (North-Holland), 1991.

[Sco89] G. Scollo. Formal Description of the OSI Session Layer: Transport Service. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 97–116. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Sti87] C. Stirling. Modal Logics for Communicating Systems. *Theoretical Computer Science*, 49:311–347, 1987.

[Sti91] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In A. Yonezawa, editor, *Concurrency: Theory, Language, and Architecture*, LNCS 491, pages 2–20. Springer-Verlag, 1991. UK/Japan Workshop, Oxford, UK, September 1989.

[SW90] C. Stirling and D. Walker. CCS, liveness, and local model checking in the linear time mu-calculus. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 166–178, 1990.

[Tho93] M. Thomas. A Translator for ASN.1 into LOTOS. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 37–52. Elsevier Science Publishers B.V. (North-Holland), 1993.

[Tho94] M. Thomas. The Story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1(1):3–15, 1994.

[TJ89] M. Thomas and K.P. Jantke. Inductive Inference for Solving Divergence in Knuth-Bendix Completion. In *Proceedings of Analogical and Inductive Inference 89*, LNCS 397. Springer-Verlag, 1989.

[Tur92] K. Turner. Constraint-Oriented Specification in LOTOS — The Compositional Specification of a File Handler. Lecture given at University of Glasgow, 1992.

[Tur93] K.J. Turner, editor. *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL.* John Wiley and Sons, 1993.

[TW93] M. Thomas and P. Watson. Solving Divergence in Knuth-Bendix Completion by Enriching Signatures. *Theoretical Computer Science*, 112:145–185, 1993.

[vE89] P. van Eijk. Tools for LOTOS Specification Style Transformation. In S. Vuong, editor, *Formal Description Techniques, II*, pages 43–52. Elsevier Science Publishers B.V. (North-Holland), 1989.

[vEKvS90] P. van Eijk, H. Kremer, and M. van Sinderen. On the use of specification styles for automated protocol implementation from LOTOS to C. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 157–168. Elsevier Science Publishers B.V. (North-Holland), 1990.

[vEVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V. (North-Holland), 1989.

[vG86] R.J. van Glabbeek. Notes on the Methodology of CCS and CSP. Technical Report CS-R8624, Centrum voor Wiskunde en Informatica, Amsterdam, 1986.

[vG90] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[Vis90] C. Vissers. FDTs for Open Distributed Systems, A Retrospective and a Prospective View. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 341–362. Elsevier Science Publishers B.V. (North-Holland), 1990. Invited paper.

[vS89] M. van Sinderen. Formal Description of the OSI Session Layer: Session Service. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 117–152. Elsevier Science Publishers B.V. (North-Holland), 1989.

[vSPV92] M. van Sinderen, L. Pires, and C.A. Vissers. Protocol Design and Implementation using Formal Methods. Technical Report Memoranda Informatica 92-19, TIOS 92-19, Universiteit Twente, 1992. To appear in The Computer Journal.

[VSvSB91] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[VTL93] Verification Techniques for LOTOS specifications. Final project report. Project information available from M. Thomas, email `muffy@dcs.gla.ac.uk`, 1993.

[Wal89] D.J. Walker. Automated Analysis of Mutual Exclusion Algorithms using CCS. *Formal Aspects of Computing*, 1(3):273–292, 1989.

[Wat92] P. Watson. The expressive power of recurrence terms. Technical Report FM-92-6, Department of Computing Science, University of Glasgow, 1992. Also submitted for publication.

[Wez90] C. Wezeman. The Co-op Method for Compositional Derivation of Conformance Testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 145–158. Elsevier Science Publishers B.V. (North-Holland), 1990.

[CONCUR] W.R. Cleaveland, editor. *CONCUR'92*, LNCS 630. Springer-Verlag, 1992. Proceedings of the Third International Conference on Concurrency Theory.

[FORTE] M. Diaz and R. Groz, editors. *Formal Description Techniques, V.* Elsevier Science Publishers B.V. (North-Holland), 1993.

[PSTV] B. Jonsson, J. Parrow, and B. Pehrson, editors. *Protocol Specification, Testing, and Verification, XI.* Elsevier Science Publishers B.V. (North-Holland), 1991.

Appendix A

A Survey of Proof Tools for LOTOS and Related Formalisms

A.1 Introduction

This appendix looks at tools which automate the methods discussed in chapters 4 and 11, and briefly at tools for LOTOS which perform functions other than verification. Although our main investigation is limited to LOTOS, the scope is widened here mainly because there are very few tools which can be used for verification of properties of LOTOS specifications. The tools presented here which are not specifically for LOTOS were chosen because they are all, or can be, proof tools for CCS [Mil89b], one of the languages from which LOTOS is derived. This also implies more concern with Basic LOTOS than with full LOTOS. A similar survey covers the process algebra verification tools which were developed during the CONCUR project [Mad92].

The tools discussed fall into two main categories: behaviourally based tools, which use the semantics of the language to build graphs representing systems and then perform various manipulations on that graph, and algebraic tools, which rely on symbolic manipulation of the terms of the language. In the first group we have, for example, the Concurrency Workbench [CPS89], TAV [GLZ89] and AUTO [MV89]. The second group includes the Pisa tool [DIN89], further development of this tool [Nes92] using HOL [Gor88], and also PAM (Process Algebra Manipulator) [Lin92].

Each tool mentioned above is discussed in a separate section; the relative merits of these tools are discussed in the final section, which also includes comment on the relation of these tools to our work on the verification requirements of LOTOS.

A.2 Behaviourally Based Tools/Semantic Reasoning

A.2.1 The Concurrency Workbench

The Concurrency Workbench [CPS89, Mol91] is an automated tool which manipulates and analyses concurrent systems, called agents, expressed in Timed CCS (TCCS) [MT90]. One of the aims of the CONCUR project was to develop a common format, called FC2, for representing systems, and to build front-ends for the existing tools which would allow them to take input from any language. The aim of this research was to strengthen tool support in general, by making tools applicable to more than just one language. As far as we are aware, this front-end to the CWB has not yet been completed.

The CWB supports the following analyses:

- Analysis of behaviours expressed in TCCS, including proof of various relations between two agents, and analysis of the state space of the transition systems of those agents. Several of the standard equivalences and preorders over concurrent systems are supported, as are some of the less well-known relations. At present, relations supported are: strong bisimulation equivalence, observational equivalence and congruence, the preorders associated with these relations (i.e. simulation relations), observational equivalence which respects divergence, may, must and testing preorders and equivalences, 2/3 preorder and equivalence, branching bisimulation equivalence, and contraction preorder. An important feature of the implementation is the ease with which it may be adapted to deal with other relations. This is discussed in more detail below.

- Determining whether a given specification satisfies a given modal logic expression by checking the model. The logic used is the propositional μ-calculus [Koz83].

- Interactive simulation of the behaviour of an agent.

- Derivation of the missing part of an incomplete specification, given an equivalent specification.

- Verification by decomposition, as described in chapter 4.

The power of the Workbench comes from splitting its implementation into three distinct parts: *interface*, which deals with Workbench-User interaction, *semantics*, which consists of various graph transformation procedures, and *analysis*, which consists of the algorithms for equivalence checking, preorder checking and satisfaction of logical propositions. The Workbench proceeds by building a graph of the transition system of the agent, and then analysing it. Each node of the graph has an information field, whose contents may be varied by applying different manipulations in the semantics layer, which may be used in the equivalence checking procedure. Flexibility is gained by using the same equivalence checking algorithm for many different equivalences by varying the contents of the information field.

Equivalence checking is carried out by a graph partition algorithm. This works by splitting the nodes of the graph into blocks, ending when either the root nodes of the agents are in different blocks (in which case the agents are not equivalent) or the relation induced by the partition is a bisimulation. The particular algorithm used is derived from that presented in [KS83].

Preorder checking is similar. To check if two graphs are related by a preorder each node of the combined graph is annotated by the nodes which are bigger than it in the preorder. The algorithm stops when the root node of one is no longer annotated by the root node of the other (they are not related), or when the annotations determine a prebisimulation (a bisimulation with special clauses to say which nodes of the transition system must be matched).

Checking that an agent satisfies a modal propositions is carried out by a tableau based procedure, documented in [Cle89], which attempts to build a top-down proof of the proposition. Again the graph transformation procedures of the semantics layer are utilised to allow different notions of satisfaction.

The Workbench is probably the most widely used of the tools considered in this report. See [EFP91] for a list of case studies, compiled for the CONCUR project, some of which use the Workbench. A few examples of use of the CWB are: verification of communications protocols [Par88, BA91], and comparison of the properties of various mutual exclusion algorithms [Wal89].

A.2.2 TAV

The main aim behind the development of TAV (Tools for Automatic Verification) [GLZ89] was to respond to a need for a process algebra tool which could not only check equivalence between agents and answer 'yes' or 'no' (like the CWB), but which could also give a reason for inequivalence, i.e. if two agents are not equivalent, then TAV provides a modal formula distinguishing those agents. This is possible because the characterisation theorem of [HM85] says that agents are bisimulation equivalent exactly when they satisfy the same modal formulae, and vice versa. Therefore, if two agents are not equivalent there must be some formulae which one satisfies but the other does not.

Like the CWB, TAV provides checking of strong and weak bisimulation equivalence, various state analyses of the transition system of the agents, and checking of modal formulas. The logic used is HML [HM85] extended by recursion (both minimum and maximum fixed points). The input language of TAV is CCS.

The algorithm used to determine equivalence between two agents in TAV is different from that used by the CWB. Given two equivalent agents, TAV uses the method of [Lar86] to construct a minimal bisimulation, whereas the partition algorithm of the CWB constructs the maximal bisimulation. This difference is analogous to the difference between the original definition of observation equivalence, in terms of refining the universal relation, and the definition of observation equivalence by bisimulation relations.

Recent work [Cle91] has shown how to refine the partition algorithm to give an explanation for bisimulation inequivalence. This is to be added to the CWB, thus removing the main advantage TAV has over the CWB. The refined algorithm is claimed to be more efficient than that employed by TAV.

A further feature of TAV is the use of modal transition systems [Lar90a]. These allow transitions to be labelled as 'allowed' or 'required', so making the specification less rigid. TAV provides checking and construction of strong and weak refinement relations and equation solving in these modal transition systems using the method of [LX90]. This allows the user to check, for example, if an implementation satisfies a specification, or to derive an implementation given a specification.

A.2.3 AUTO

AUTO [MV89] is an equivalence checking tool which constructs finite state machines corresponding to the input agents, which are finite closed terms of the MEIJE process algebra [AB84]. AUTO computes strong, weak and branching bisimulations, reduces the input machines according to the axioms of the language (which helps control the state space of the agents) and can simulate behaviours of agents. The interface can be textual, through a command language, or window based.

MEIJE has been shown to be a *universal* process algebra [dS85]. Taking advantage of this fact, a generalisation of AUTO, MAUTO, has been developed which can be compiled with the structural operational semantics of a process algebra to give an equivalence checker for that formalism. An instantiation of MAUTO for Basic LOTOS is part of the LITE toolkit (mentioned later in appendix A.4). The translation from LOTOS to MEIJE is documented in [DMdS90].

A problem suffered by all of the systems considered so far is that of state explosion, since the systems construct graphs of the agents. This also limits their application to finite state processes only. We now move on to look at algebraic proof systems.

A.3 Algebraically Based Tools/Syntactic Reasoning

A.3.1 A Rewriting Strategy

[DIN89] presents an interactive rewriting-based system for proving properties of CCS specifications; in particular, the system attempts to prove observational equivalence between two CCS specifications. This system is different those above in that it has no special internal representation for the CCS specification; all manipulation of the terms is symbolic. In this way, the developers of the system avoid some of the problems of state explosion. The system also allows a high degree of user control and, like TAV, gives meaningful responses to queries.

The power of the system comes from the three ways in which the user may manipulate the input:

- Operational semantics of CCS. This section implements the transition rules of CCS. This allows simulation of a CCS process.
- Operationally defined equivalences. A term rewriting system corresponding to the chosen equivalence relation.
- Axioms for Equivalences. Here the axioms of the equivalence relation can be applied one at a time. This process is user controlled.

The implementation is modular and may be extended easily to encompass different verification strategies and equivalences. Currently modules dealing with observational equivalence between two (possibly recursive) agents have been implemented.

The main problem of this approach is that only some of the defined behavioural equivalences defined in the process algebra literature actually have complete term rewriting systems associated with them. In particular, the rewriting system associated with observational equivalence causes divergence of the completion algorithm. This is a problem because unique normal forms are not guaranteed unless the term rewriting system is complete.

To combat this problem, an ad hoc rewriting strategy is used which derives the normal form of a term without using the completion algorithm. The strategy is based on the normal forms of observational congruence, obs-normal forms, which have been used by Hennessy and Milner to prove the completeness of the axiomatisation of observational congruence for a subset of CCS (no recursion) [HM85]. The proof of the correctness and completeness of this strategy appears in [IN90].

Informally, the divergence of the completion algorithm can be attributed to the interaction of the τ laws. The problem is that many proofs will require these laws to be applied first in one direction, then in the other (unfolding and folding), but of course rewrite rules can only be applied in one direction. The strategy works by applying these unfolding operations until it is possible to eliminate all derivatives of the term which are semantically contained in other parts of the term. When this is done the term is folded again. In other words, instead of trying to create an infinite number of rules to do the right number of unfolds and folds (which is what happens in completion), the original rules are applied a sufficient number of times to "simulate" the necessary rule. The most difficult part of the implementation of this strategy lies in identifying when sufficient unfoldings have been completed.

From CCS to LOTOS

A further development of this work [DIN91] is concerned with adding a translation module to the system to allow it to deal with a subcalculus of LOTOS (without value passing or recursion) rather than CCS (LOTOS being more widely used in the industrial community). Some of the laws from appendix B.2.2 of the LOTOS standard [ISO88] are used; a slightly different set from the ones we use in RRL in chapter 6.

In [DIN91] two examples are described: the readers and writers problem, and a rather bizarre candy/change machine. This is the source of the examples of sections 9.3 and 9.4.

Using HOL

The system described above is implemented in PROLOG. Yet another development of the work above [CIN91, Nes92] uses a different system as a base on which to build the CCS tools, namely HOL. The use of HOL provides a meta language which will enable the user to define their own verification strategies and tactics. The current system supports verification of observational congruence and expansion of parallel terms (using a lazy evaluator of the expansion laws), and also satisfaction with respect to the propositional μ-calculus.

A.3.2 PAM

The Process Algebra Manipulator [Lin91, Lin92] differs from the other systems discussed so far in that it is a proof assistant, and not designed to be fully automatic, and also in that, like MAUTO, it is not tied to a particular process algebra. The main features of PAM are described in section 8.2. Not mentioned there is the ability to link to the CWB. Care must be taken that the agents supplied to the CWB are in the correct format and also finite state. Examples carried out in PAM include the Alternating Bit Protocol in CCS and ACP [BK84], the Scheduler from [Mil89b] and the Two Bit Buffer in CCS. Other languages described for PAM are EPL [Hen88], CSP [Hoa85], and, of course, LOTOS, as described in chapter 8.

The main advantages of PAM over other systems are its generality, its ability to reason over open terms, and its ability to cope with infinite state processes. PAM also has a relatively nice mouse-driven graphical interface, with buttons for each rule, rather than the usual text-based interface.

This concludes the (brief) survey of verification tools for LOTOS and related process algebras. The next section looks at other tools specifically for LOTOS.

A.4 Tools for LOTOS

One of the products of the LOTOSPHERE project [vSPV92] was the LITE (LOTOS Integrated Tool Environment) toolkit [LITE]. LITE provides a set of tools which are built round a common representation of LOTOS behaviours. The following functionalities are provided:

- Editing/syntax checking (including a graphical editor for GLOTOS).
- Analysis of cross references and static dependencies.
- Completion of abstract data types.
- Checking for various properties of the abstract data types, e.g. termination, consistency and completeness.
- Simulation of LOTOS behaviours by building an extended finite state machine and a simulation tree obtained by specifying values for the variables in the description. These values are obtained by input from the user, or automatically generated by a narrowing tool.
- Compilation of an annotated LOTOS specification into C code. The annotations provide information to the compiler about the implementation, e.g. which C object implements a LOTOS sort or operation.
- Translations of a LOTOS specification into its labelled transition system, and abstract data types into their equivalent rewrite system.
- Testing of behaviours by composition with user-defined test processes. Tests can be accepted or rejected, i.e. the process may/must pass the test, or the process will never pass the test.
- Transformation of a LOTOS specification into an equivalent one. Two transformations are available: regrouping parallel processes according to a pattern specified by the user and preserving strong bisimulation equivalence, and bipartition of functionality, splitting the specification with respect to event sets preserving weak bisimulation equivalence.
- Proof of behavioural equivalences and preorders over Basic LOTOS processes (using AUTO or by writing the specification in a FC2 format which can be understood by the Concurrency Workbench, see section A.2.1).
- Temporal logic model checking (Basic LOTOS only). The logic used is ATCL [CES86].
- Generation of the canonical tester of a Basic LOTOS specification (via Cooper [Ald89]).

All components can be applied to full LOTOS except those flagged as for Basic LOTOS processes only. Basic LOTOS is obtained from the full LOTOS specification by forgetting the data types.

This transformation only preserves reachability properties of the original specification, as discussed in section 10.3.1.

Other tools for LOTOS include the University of Ottawa toolkit (interpreter for LOTOS, various tools for Graphical LOTOS) and Squiggles (checking of strong, observational and testing equivalences on finite Basic LOTOS processes using Paige and Tarjan's partition algorithm [PT87]).

A.5 Summary and Discussion

Our aim is to investigate the verification of LOTOS specifications, and to do so using tools. Although we are really interested in equational reasoning as a proof technique, we should also consider other proof techniques which might be suitable and which are already implemented, perhaps for CCS, if not for LOTOS. Above we have presented summaries of the abilities of several proof tools; in this section we consider how *we* might use those proof tools.

To summarize the relative advantages and disadvantages of the tools discussed above: the most versatile tool in terms of proof facilities seems to be the CWB. The CWB will be even more versatile when the front end accepting FC2 is completed, making the CWB independent of input language, in which case the CWB can be used for LOTOS. The main drawback of the CWB is the problem of state explosion, and its inability to handle infinite state processes. TAV seems to be largely superseded by the Workbench, especially if the Workbench is extended to provide reasons for inequivalence. The final tool in the behaviourally based category, AUTO, is already implemented for LOTOS as part of LITE. However, this tool can only accept Basic LOTOS input.

The Pisa tool of Inverardi and Nesi offers a high degree of interaction and control over the proof to the user; however, only observation equivalence over LOTOS processes is evaluated, and extensions to the system requires knowledge of Prolog or HOL. One of the benefits of the HOL system is that it is built from the very basics of CCS, i.e. the operational semantics, with all the axioms of the observation equivalence being derived from that and the definition of bisimulation. Also, the tactic language of HOL is very powerful.

As we discovered in the case study of chapter 7, if a rewriting strategy is to be used we require a high degree of interaction and control over the development of the proof. PAM offers this, as well as flexibility in the input language and the equivalence relation. The price that has to be paid is that defining the language and relation takes time (and skill). However, this only has to be done once, and we have defined the most commonly used relations. A further benefit of using PAM is that it is the only tool, apart from AUTO, which has a more sophisticated interface, making it easy to use.

It is clear that none of the tools presented in this appendix have all the features we desire of a verification tool for LOTOS. By adopting PAM we have the opportunity to build a system to our liking, but without having to do too much implementation work (which, given the number of tools available, would be rather a waste of time).

At the moment, our proof system is applicable only to Basic LOTOS specifications, but we feel that we have laid the ground work for an extension of the proof system which could provide a unified framework for verification of full LOTOS specifications.

Appendix B

LOTOS Inference Rules

In this appendix we present the syntax and semantics of full LOTOS. Although these may be found in the LOTOS standard [ISO88], the presentation there is rather cluttered and unclear. We hope our presentation of LOTOS is simpler and easier to understand than that of the standard.

B.1 LOTOS Syntax

The LOTOS syntax is given in the form of a BNF grammar. To make the grammar more readable we adopt the convention that reserved words of LOTOS will be shown in bold, other symbols of LOTOS will be shown in typewriter style (following the format used for the examples presented in the main body of the thesis), while the non-terminals of the BNF are shown in italics. Note that in the LOTOS language, no special meaning is attributed to the font in which a character is written. Optional parts of a production rule (0 or 1 occurrences) are enclosed by square brackets, optionally repeated parts (0 or more occurrences) are enclosed by braces.

The grammar is split into three parts below: first we give the basic elements of the language; identifiers, special symbols, etc., second we give the syntax for the process algebra part of the language, and finally we give the ACT ONE related syntax.

B.1.1 Basic Definitions

The syntax of the basic elements such as identifiers is as follows:

letter	$\stackrel{def}{=}$	`a` | `b` | `c` | ... | `z`.
digit	$\stackrel{def}{=}$	`0` | `1` | ... | `9`.
normal-character	$\stackrel{def}{=}$	*letter* | *digit*.
special-character	$\stackrel{def}{=}$	`#` | `%` | `&` | `*` | `+` | `-` | `.` | `/` | `<` | `=` | `>` | `@` | `\` | `^` | `~` | `{` | `}`.

reserved-words $\stackrel{def}{=}$ **specification** | **endspec** | **behaviour** | **type** | **endtype** | **is**
| **actualizedby** | **using** | **library** | **endlib** | **renamedby**
| **formalsorts** | **formalopns** | **formaleqns** | **sorts** | **opns**
| **ofsort** | **forall** | **eqns** | **sortnames** | **opnnames** | **for**
| **process** | **endproc** | **where** | **stop** | **exit** | **noexit** | **any**
| **i** | **let** | **in** | **par** | **accept** | **choice** | **hide** | **of**.

special-symbols $\stackrel{def}{=}$ `=` | `=>` | `:=` | `>>` | `[>` | `||` | `|||` | `|[` | `]|`
| `[]` | `->` | `;` | `,` | `:` | `[` | `]` | `(` | `)` | `?` | `!` | `_`.

identifier $\stackrel{def}{=}$ *letter* [{*normal-character* | _ } *normal-character*].

special-identifier $\stackrel{def}{=}$ *special-character* {*special-character*}
| *digit* [{*normal-character* | _ } *normal-character*].

comment $\stackrel{def}{=}$ (* "any string of text" *)

The following observations may be helpful in understanding the rest of the LOTOS syntax.

- Since brackets and bar appear both as LOTOS syntax, `[]` and `|`, and as BNF syntax, [ ] and |, care must be taken not to confuse them in the following rules. In the syntax rules we have tried to distinguish them by using a different typeface. Additionally, it may help to know that the LOTOS symbols `[]` appears in the production rules for the following non-terminals: *sum-expression, par-op, choice-expression, guarded-expression, selection-predicate, process-instantiation* and *relabelling*, and the `|` symbol appears only in the rule for *par-op*. All other occurrences of these symbols are therefore BNF syntax.

- The reserved word **behaviour** may be substituted by **behavior** if desired.

- A special identifier is used for user-defined operators composed of symbols. All **-identifiers* are *identifiers* except *operation-identifier*, which can either be an *identifier* or a *special-identifier*. No identifier or special identifier is allowed to have the same spelling as a reserved word or special symbol.

- Comments, when included, should be enclosed by `(* ... *)`. Comments are not part of the formal text of a LOTOS specification. A comment may include text (possibly foreign language text) and/or pictures.

- Any **-identifier-list* not otherwise defined is a list, separated by commas, of **-identifiers*, where * stands for different kinds of identifier, e.g. specification, process, gate, value, ...

- Association of binary operators is to the right, unless parentheses indicate otherwise. The operators `[]`, `>>`, `[>`, `||`, `|||` and `|[A]|` are all associative, but note that mixed use of the different parallel operators is not associative.

B.1.2 Process Algebra Syntax

The following production rules apply to specifications and to the process algebra part of a specification. Some reference is made in these rules to the ACT ONE syntax of the next section.

specification	$\overset{def}{=}$	**specification** *specification-identifier formal-parameter-list*
		data-type-definitions **behaviour** *definition-block* **endspec**.
process-definition	$\overset{def}{=}$	**process** *process-identifier formal-parameter-list* :=
		definition-block **endproc**.
formal-parameter-list	$\overset{def}{=}$	[[*gate-identifier-list*]][(*identifier-declarations*)] :**exit** [(*sort-list*)]
		| [[*gate-identifier-list*]][(*identifier-declarations*)] :**noexit**.
identifier-declarations	$\overset{def}{=}$	*identifier-declaration* {, *identifier-declaration*}.
identifier-declaration	$\overset{def}{=}$	*value-identifier-list* : *sort-identifier*.
sort-list	$\overset{def}{=}$	*sort-identifier* {, *sort-identifier*}.
definition-block	$\overset{def}{=}$	*behaviour-expression* [*local-definitions*].
local-definitions	$\overset{def}{=}$	**where** *local-definition* {*local-definition*}.
local-definition	$\overset{def}{=}$	*data-type-definition* | *process-definition*.
behaviour-expression	$\overset{def}{=}$	*local-definition-expression*
		| *sum-expression*
		| *par-expression*
		| *hiding-expression*
		| *enable-expression*.
local-definition-expression	$\overset{def}{=}$	**let** *identifier-equations* **in** *behaviour-expression*.
identifier-equations	$\overset{def}{=}$	*identifier-equation* {, *identifier-equation*}.
identifier-equation	$\overset{def}{=}$	*identifier-declaration* = *value-expression*.
sum-expression	$\overset{def}{=}$	**choice** *sum-domain-expression* [] *behaviour-expression*.
sum-domain-expression	$\overset{def}{=}$	*identifier-declarations* | *gate-declarations*.
par-expression	$\overset{def}{=}$	**par** *gate-declarations par-op behaviour-expression*.
par-op	$\overset{def}{=}$	|| | ||| | |[*gate-identifier-list*]|.
hiding-expression	$\overset{def}{=}$	**hide** *gate-identifier-list* **in** *behaviour-expression*.
enable-expression	$\overset{def}{=}$	*disable-expression* [*enable-op enable-expression*].
enable-op	$\overset{def}{=}$	>> [**accept** *identifier-declarations* **in**].
disable-expression	$\overset{def}{=}$	*parallel-expression* [[> *disable-expression*].
parallel-expression	$\overset{def}{=}$	*choice-expression* [*par-op parallel-expression*].

choice-expression	$\stackrel{def}{=}$	*guarded-expression* [[] *choice-expression*].
guarded-expression	$\stackrel{def}{=}$	[*premiss*] -> *guarded-expression* | *action-prefix-expression*.
action-prefix-expression	$\stackrel{def}{=}$	*action-denotation* ; *action-prefix-expression*
		| *atomic-expression*.
action-denotation	$\stackrel{def}{=}$	*gate-identifier*
		[*experiment-offer* {*experiment-offer*}[*selection-predicate*]]
		| **i**.
experiment-offer	$\stackrel{def}{=}$	?*identifier-declaration* | !*value-expression*.
selection-predicate	$\stackrel{def}{=}$	[*premiss*].
atomic-expression	$\stackrel{def}{=}$	**stop**
		| **exit** [*exit-parameter-list*]
		| *process-instantiation*
		| (*behaviour-expression*)
		| *relabelling-expression*.
exit-parameter-list	$\stackrel{def}{=}$	(*exit-parameter* {, *exit-parameter*}).
exit-parameter	$\stackrel{def}{=}$	*value-expression* | **any** *sort-identifier*.
process-instantiation	$\stackrel{def}{=}$	*process-identifier* [[*gate-identifier-list*]][*value-expression-list*].
relabelling-expression	$\stackrel{def}{=}$	(*behaviour-expression*) *relabelling*.
relabelling	$\stackrel{def}{=}$	[*replacements*].
replacements	$\stackrel{def}{=}$	*gate-name/gate-name* [, *replacements*].
value-expression	$\stackrel{def}{=}$	[*value-expression operation-identifier*] *simple-expression*.
simple-expression	$\stackrel{def}{=}$	*term-expression* [**of** *sort-identifier*].
term-expression	$\stackrel{def}{=}$	*value-identifier*
		| *operation-identifier* [*value-expression-list*]
		| (*value-expression*).
value-expression-list	$\stackrel{def}{=}$	(*value-expression* {, *value-expression*}).
premiss	$\stackrel{def}{=}$	*simple-equation* | *value-expression*.
simple-equation	$\stackrel{def}{=}$	*value-expression* = *value-expression*.
gate-declarations	$\stackrel{def}{=}$	*gate-declaration* {, *gate-declaration*}.
gate-declaration	$\stackrel{def}{=}$	*gate-identifier-list* **in** [*gate-identifier-list*].

B.1.3 LOTOS Data Type Syntax

The syntax of this section is only required for full LOTOS.

data-type-definitions	$\stackrel{def}{=}$	{*data-type-definition*}.

data-type-definition $\overset{def}{=}$ **type** *type-identifier* **is** *p-expression* **endtype**

| **library** *type-identifier* {, *type-identifier*} **endlib**.

p-expression $\overset{def}{=}$ [*type-union*] *p-specification*

| *type-identifier* **actualizedby** *type-union* **using** *replacement*

| *type-identifier* **renamedby** *replacement*.

type-union $\overset{def}{=}$ *type-identifier* [, *type-union*].

p-specification $\overset{def}{=}$ [**formalsorts** *sort-list*]

[**formalopns** *operation-list*]

[**formaleqns** *equation-lists*]

[**sorts** *sort-list*] [**opns** *operation-list*] [**eqns** *equation-lists*].

replacement $\overset{def}{=}$ [**sortnames** *sort-pair-list*] [**opnnames** *operation-pair-list*].

sort-pair-list $\overset{def}{=}$ *sort-pair* [*sort-pair-list*].

operation-pair-list $\overset{def}{=}$ *operation-pair* [*operation-pair-list*].

sort-pair $\overset{def}{=}$ *sort-identifier* **for** *sort-identifier*.

operation-pair $\overset{def}{=}$ *operation-identifier* **for** *operation-identifier*.

sort-list $\overset{def}{=}$ *sort-identifier* {, *sort-identifier*}.

operation-list $\overset{def}{=}$ *operation* {*operation*}.

operation $\overset{def}{=}$ *operation-descriptor* {, *operation-descriptor*} :

[*sort-list*] -> *sort-identifier*.

operation-descriptor $\overset{def}{=}$ *operation-identifier*

| _ *operation-identifier* _

equation-lists $\overset{def}{=}$ [**forall** *identifier-declarations*] *equation-list* {*equation-list*}.

equation-list $\overset{def}{=}$ **ofsort** *sort-identifier* [**forall** *identifier-declarations*]

equation {; *equation*} [;].

equation $\overset{def}{=}$ [*premisses* =>] *simple-equation*.

premisses $\overset{def}{=}$ *premiss* {, *premiss*}.

Having formalised the syntax of the language, the next section defines the semantics.

B.2 LOTOS Semantics

The semantics of LOTOS processes is given in two parts:

- structured labelled transition systems, see section B.2.1 for the definition, and the inference rules which generate those transition systems, see section B.2.2,

- equivalence relations over process behaviours/structured labelled transition systems, see section B.2.3 onwards for laws, and also sections 3.4.3 and 3.5.3 for motivation and definitions.

Each process is seen as a set of states, with arcs representing transitions connecting those states. Each transition is labelled by its action; for Basic LOTOS labels are gate names, while for full LOTOS labels are pairs consisting of a gate name and a string of data values.

In order to turn a LOTOS specification into a labelled transition system, the specification is first "flattened". The flattening function essentially ensures that the specification adheres to the LOTOS syntax, but also removes all hierarchical structure, ensures uniqueness of variable names, and that all names and types used are previously defined. The resulting object is called a *canonical LOTOS specification.* The inference rules of transition may then be applied to the canonical LOTOS specification to give a structured labelled transition system (actually the inference rules give a *class* of labelled transition systems, each relating to different instantiations of the formal parameters of the specification).

In the next section we give the standard definitions relating to labelled transition systems and algebras which are required for the definition of the inference rules in section B.2.2.

B.2.1 Algebras and Transition Systems

Most of the following definitions relate to algebras and full LOTOS semantics; they can be ignored for Basic LOTOS.

- A flattened canonical LOTOS specification, *CLS*, is given by a pair $\langle AS,\ BS\rangle$. AS is an algebraic specification $\langle S,\ OP,\ E\rangle$, where $\langle S,\ OP\rangle$ is a signature and E is a set of conditional equations. The semantics of AS is given by the many sorted algebra $Q(AS)$ which is the quotient term algebra of AS. BS is a behaviour specification $\langle PDEFS,\ pdef_0\rangle$ where $PDEFS$ is the set of all the process definition in CLS, and $pdef_0$ is the top level process of the specification. Each element of $PDEFS$ is a pair $\langle p,\ B_p\rangle$ of a process name and the corresponding behaviour expression. All sort names and operations in BS are defined in AS (the flattening function ensures this).

- The algebraic specification generates a derivation system D, which allows us to deduce if two terms are congruent, i.e. $D \vdash t_1 \equiv_{AS} t_2$. The congruence class of a term t, written $[t]$, is defined as $[t] = \{t' \mid t \equiv_{AS} t'\}$.

- A *many-sorted algebra* A is a pair $\langle D, O\rangle$ where

  1. D is a set of sets, where each set is called a *data carrier* of A. Elements of the data carrier are called *data values.*

2. O is a set of total functions over the data carriers, $D_1 \times \ldots \times D_n \dashrightarrow D$ where $D, D_1, \ldots D_n$ are data carriers and $n \geq 0$.

- A *quotient term algebra* $Q(A) = \langle D_Q,\ O_Q \rangle$ where

  1. D_Q is the set $\{Q(s) \mid s \in S\}$, where $Q(s) = \{[t] \mid t \text{ is a ground term of sort } s\}$ for each $s \in S$,
  2. O_Q is the set of functions $\{Q(op) \mid op \in OP\}$, where the $Q(op)$ are defined by

$$Q(op)([t_1], \ldots, [t_n]) = [op(t_1, \ldots, t_n)].$$

- A *labelled transition system Sys* is a 4-tuple $\langle S,\ Act, T, s_0 \rangle$ as defined in section 3.4.3, with $Act = \ G \ \cup \ \{\mathbf{i}\}$.

- A *structured labelled transition system Struc* is a 5-tuple $\langle S, Act, A, T, s \rangle$ where $A = \langle D, O \rangle$ is a many-sorted algebra such that $\langle S, Act, T, s \rangle$ is a labelled transition system, for $Act \overset{def}{=} \{\mathbf{i}\} \cup \{gv \mid g \in G, v \in (\bigcup D)^*\}$. This is also referred to as a labelled transition system over A.

  In other words, a structured labelled transition system is just like a labelled transition system, except that each label g is decorated by a string of values from D.

We also define the following auxiliary functions to help in the definition of the inference rules:

fg : *process-names* $\dashrightarrow$ $(gate\text{-}names)^*$

which yields the formal gate parameter list of a process p

fv : *process-names* $\dashrightarrow$ $(variables)^*$

which yields the formal value parameter list of a process p

sort : *variables* $\dashrightarrow$ *sort-names*

which yields the sort of a value.

name : *transition-label* $\dashrightarrow G \ \cup \ \{\mathbf{i}\}$

given an experiment offer, yields the gate name of the transition, which could be δ, or $\mathbf{i}$.

$\mathcal{L}$: *behaviour-expression* $\dashrightarrow G^*$

yields the language of a process.

The following notational conventions help us to give the inference rules of transition with less clutter:

- B (B', B_i, ...) refer to process behaviours. B, B_i is defined for each rule, but $B', B'', \ldots$ are *behaviour-expression* instances, unless stated otherwise.

- $g, g_1, \ldots$ are *gate-names*.

- $d_1, \ldots$ are *experiment-offer* instances, i.e. of the form $!x$ or $?x$.
- $x, x_1, \ldots$ are *variable* instances.
- $t_1, \ldots$ are *term-expression* instances.
- $x_1 = t_1, \ldots$ are therefore *identifier-equation* instances.
- p is a variable over *process-names.*
- SP denotes a *selection-predicate* instance.
- a ranges over transition labels, *Act.*

We also use the following notation for substitution: $[t_1/x_1, \ldots, t_n/x_n]B$ meaning the simultaneous replacement of the x_i in B by the appropriate t_i.

B.2.2 LOTOS Inference Rules

These are split into two distinct groups: the axioms of transition, and inference rules of transition. The numbering given here coincides with that in the standard [ISO88]. Note that one of the rules appears to be repeated (B_1 is a simplification of B); this is because one inference rule applies to sum-expressions while the other applies to par-expressions.

For Basic LOTOS the same axioms and rules apply but assume the data parts of the interactions are empty. The rules relating to guards, selection predicates, etc. may be ignored.

The presentation here puts prerequisites of the axiom or inference rule on the left and the inference rule or axiom itself in a box on the right. Any conditions on the validity of the rule or auxiliary definitions required come below the rule.

Axioms of Transition

(a) $B = \mathbf{i}; B'$

$$\boxed{B \xrightarrow{\mathbf{i}} B'}$$

where B' is an *action-prefix-expression.*
where B' is an *action-prefix-expression.*

(b1) $B = g d_1 \ldots d_n; B'$

$$\boxed{B \xrightarrow{g v_1 \ldots v_n} [ty_1/y_1, \ldots ty_m/y_m]B'}$$

iff

$$\begin{array}{ll} v_i = [t_i] & \text{if } d_i = !t_i (1 \le i \le n) \text{ and } t_i \text{ is a ground term,} \\ v_i \in Q(s_i) & \text{if } d_i = ?x_i (1 \le i \le n) \text{ with } sort(x_i) = s_i, \\ ty_1, \ldots, ty_m \text{ are term instances with } v_i = [ty_i] & \text{if } d_i = ?y_j (1 \le i \le n, 1 \le j \le m) \\ & \text{and } \{y_1, \ldots, y_m\} = \{x_i \mid d_i = ?x_i, 1 \le i \le n\}. \end{array}$$

where B' is an *action-prefix-expression.*

(b2) $B = gd_1 \ldots d_n[SP]; B'$ $\boxed{B \xrightarrow{gv_1 \ldots v_n} [ty_1/y_1, \ldots ty_m/y_m]B'}$

iff

v_i and ty_i defined as above, and providing $D \vdash SP'$ where SP' denotes the ground equation obtained by simultaneous replacement in SP of all x_i in SP that also occur contained in a d_i variable offer, i.e. $d_i = ?x_i (1 \leq i \leq n)$, by a term $t \in v_i$.

(c1) $B = \mathbf{exit}(E_1, \ldots, E_n)$ $\boxed{B \xrightarrow{\delta v_1 \ldots v_n} \mathbf{stop}}$

iff

$v_i = [E_i]$ if E_i is a ground term $(1 \leq i \leq n)$

$v_i \in Q(s_i)$ if $E_i = \mathbf{any}\ s_i (1 \leq i \leq n)$

where $E_1, \ldots, E_n$ are *exit-parameter* instances.

(c2) $B = \mathbf{exit}$ $\boxed{B \xrightarrow{\delta} \mathbf{stop}}$

Inference Rules of Transition

(a) $B = \mathbf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathbf{in}\ B'$ $\boxed{\dfrac{[t_1/x_1, \ldots, t_n/x_n]B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$

(b1) $B = \mathbf{choice}\ g\ \mathbf{in}\ [g_1, \ldots, g_n]\ [\,]\ B'$ $\boxed{\dfrac{(B')[g_i/g] \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$

for each $g_i \in \{g_1, \ldots, g_n\}$.

(b2) $B = \mathbf{choice}\ x\ [\,]\ B'$ $\boxed{\dfrac{[t/x]B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$

iff t is a ground term with $[t] \in Q(s)$, where x is a variable with $sort(x) = s$.

(b3) B_1 **is a simplification of** B **in normal form** $\boxed{\dfrac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1'}}$

(c1) $B = \mathbf{par}\ g\ \mathbf{in}\ [g_1, \ldots, g_n]\ \textit{par-op}\ B'$ $\boxed{\dfrac{(B')[g_1/g]\ \textit{par-op} \ldots \textit{par-op}\ (B')[g_n/g] \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$

(c2) B_1 **is a simplification of** B **in normal form** $\boxed{\dfrac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1'}}$

(d) $B = \textbf{hide } g_1, \ldots, g_n \textbf{ in } B'$

$$\boxed{\begin{array}{c} B' \xrightarrow{a} B'' \\ \hline B \xrightarrow{a} \textbf{hide } g_1, \ldots, g_n \textbf{ in } B'' \\ \text{if name } (a) \notin \{g_1, \ldots, g_n\} \end{array}}$$

$$\boxed{\begin{array}{c} B' \xrightarrow{a} B'' \\ \hline B \xrightarrow{\mathbf{i}} \textbf{hide } g_1, \ldots, g_n \textbf{ in } B'' \\ \text{if name } (a) \in \{g_1, \ldots, g_n\} \end{array}}$$

(e) $B = B_1 \gg \textbf{ accept } x_1, \ldots, x_n \textbf{ in } B_2$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{a} B_1{}' \\ \hline B \xrightarrow{a} B_1{}' \gg \textbf{ accept } x_1, \ldots, x_n \textbf{ in } B_2 \\ \text{name } (a) \neq \delta \end{array}}$$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{\delta v_1 \ldots v_n} B_1{}' \\ \hline B \xrightarrow{\mathbf{i}} [t_1/x_1, \ldots, t_n/x_n] B_2 \end{array}}$$

where $t_i, \ldots, t_n$ are ground terms with $[t_1] = v_1, \ldots, [t_n] = v_n$.

(f) $B = B_1 \,[\!>\; B_2$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{a} B_1{}' \\ \hline B \xrightarrow{a} B_1{}' \,[\!>\; B_2 \\ \text{name } (a) \neq \delta \end{array}}$$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{\delta v_1 \ldots v_n} B_1{}' \\ \hline B \xrightarrow{\delta v_1 \ldots v_n} B_1{}' \end{array}}$$

$$\boxed{\begin{array}{c} B_2 \xrightarrow{a} B_2' \\ \hline B \xrightarrow{a} B_2' \end{array}}$$

(g1) $B = B_1 | [g_1, \ldots, g_n] | B_2$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{a} B_1{}' \\ \hline B \xrightarrow{a} B_1{}' \,| [g_1, \ldots, g_n] | \, B_2 \\ \text{name } (a) \notin \{g_1, \ldots, g_n, \delta\} \end{array}}$$

$$\boxed{\begin{array}{c} B_2 \xrightarrow{a} B_2' \\ \hline B \xrightarrow{a} B_1 \,| [g_1, \ldots, g_n] | \, B_2' \\ \text{name } (a) \notin \{g_1, \ldots, g_n, \delta\} \end{array}}$$

$$\boxed{\begin{array}{c} B_1 \xrightarrow{a} B_1{}', B_2 \xrightarrow{a} B_2' \\ \hline B \xrightarrow{a} B_1{}' \,| [g_1, \ldots, g_n] | \, B_2' \\ \text{name } (a) \in \{g_1, \ldots, g_n, \delta\} \end{array}}$$

where $g_1, \ldots, g_n$ is a (possibly empty) list of *gate-name* instances.

(g2) $B = B_1 ||| B_2$

$$\boxed{\dfrac{B_1 |[\,]| B_2 \xrightarrow{a} B'}{B \xrightarrow{a} B'}}$$

(g3) $B = B_1 || B_2$

$$\boxed{\dfrac{B_1 |[g_1, \ldots, g_n]| B_2 \xrightarrow{a} B'}{B \xrightarrow{a} B'}}$$

where $\{g_1, \ldots, g_n\} = G$.

(h) $B = B_1 \mathbin{[]} B_2$

$$\boxed{\dfrac{B_1 \xrightarrow{a} {B_1}'}{B \xrightarrow{a} {B_1}'}}$$

$$\boxed{\dfrac{B_2 \xrightarrow{a} B_2'}{B \xrightarrow{a} B_2'}}$$

(j) $B = [SP] \mathrel{->} B'$

$$\boxed{\dfrac{B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$$

iff SP is a ground equation and $D \vdash SP$.

(k1) $B =$ **stop**

$$\boxed{\text{no inference rules are generated}}$$

(k2) $B = p\ [g_1, \ldots, g_n](t_1, \ldots, t_m)$

$$\boxed{\dfrac{([t_1/x_1, \ldots, t_m/x_m]B_p)[g_1/h_1, \ldots, g_n/h_n] \xrightarrow{a} B'}{B \xrightarrow{a} B'}}$$

iff $\langle p, B_p \rangle \in$ BS.PDEFS
where *fg(p)* $= \langle h_1, \ldots, h_n \rangle$, and *fv(p)* $= \langle x_1, \ldots, x_m \rangle$.

(k3) $B = (B')$

$$\boxed{\dfrac{B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}}$$

(m) $B = (B')\ [g_1/h_1, \ldots, g_n/h_n]$

$$\boxed{\dfrac{B' \xrightarrow{a} B''}{B \xrightarrow{a'} (B'')[g_1/h_1, \ldots, g_n/h_n]}}$$

where

$h_1, \ldots, h_n$ are *gate-names*,

$a \ = g v_1 \ldots v_m$,

$a' \ = g v_1 \ldots v_m$ if $g \notin \{h_1, \ldots, h_n\}$

$a' \ = g_i v_1 \ldots v_m$ if $g = h_i (1 \leq i \leq n)$

These rules and axioms completely define the structured labelled transition system of a canonical LOTOS specification.

The equivalence over labelled transition systems, tree equivalence, is too strong for verification purposes, so we usually define other equivalence relations over labelled transition systems which allow more identifications. In the following sections we give the laws relating to weak bisimulation congruence and equivalence, testing congruence and equivalence, and the preorders **red** and **cred** as given in [ISO88]. The laws given do not completely characterise the relations, and therefore some proofs may require additional proof techniques. The definitions of these relations may be found in sections 3.4.3 and 3.5.3, assuming the appropriate modification of *Act* to include structured labels. The laws below are the source for the rewrite rules of chapter 6 and the PAM axioms of chapter 8.

B.2.3 Weak Bisimulation Congruence Laws

Appendix B.2.2 to the LOTOS standard [ISO88] contains a set of weak bisimulation congruence laws. We know these laws are sound with respect to the model, but they are not complete. This does not concern us as it is always possible to add specific laws to cover the case we want.

The laws for weak bisimulation congruence, written $B \equiv_{wbc} C$, are as follows:

a) Action-prefix Let $g \ldots ?x : t \ldots$ be an action-denotation with an experiment offer $?x : t$, and let z be a value identifier that does not occur in $g \ldots ?x : t \ldots [E];\ B$.

1. $g \ldots ?x : t \ldots [E];\ B \equiv_{wbc}\ g \ldots ?z : t \ldots [z/x][E];\ [z/x]B$
2. $g \ldots ?x : t \ldots;\ B \equiv_{wbc}$ **choice** $x : t\ []\ g \ldots !x \ldots;\ B$
3. $g!E_1 \ldots !E_n[E];\ B \equiv_{wbc}\ [E]$ -> $g!E_1 \ldots !E_n;\ B$

b) Choice

1. $B_1\ []\ B_2\ \equiv_{wbc}\ B_2\ []\ B_1$
2. $B_1\ []\ (B_2\ []\ B_3)\ \equiv_{wbc}\ (B_1\ []\ B_2)\ []\ B_3$
3. $B\ []$ **stop** $\equiv_{wbc}\ B$
4. $B\ []\ B\ \equiv_{wbc}\ B$
5. $[E/x]B\ []$ (**choice** $x : t\ []\ B) \equiv_{wbc}$ **choice** $x : t\ []\ B$ if [E] $\in$ Q(t)
6. **choice** $x : t\ []\ B\ \equiv_{wbc}\ B$ if x is not free in B
7. **choice** $x : t\ []$ **exit**$(\ldots,\ x, \ldots) \equiv_{wbc}$ **exit**$(\ldots,$ **any** $t, \ldots)$

c) **Parallel** If a law holds for all of the parallel operators '|' is used to denote any of them (using the same instance throughout the law). Let A, A' be lists of gate-identifiers.

1. $B_1 \mid B_2 \equiv_{wbc} B_2 \mid B_1$
2. $B_1 \mid (B_2 \mid B_3) \equiv_{wbc} (B_1 \mid B_2) \mid B_3$
3. (a) $\mathbf{exit}(E_1,\ldots,E_n) \mid \mathbf{exit}(E'_1,\ldots,E'_m) \equiv_{wbc} \mathbf{exit}(E_1,\ldots,E_n)$ if $n = m$ and $\forall 1 \leq i \leq n.([E_i] = [E'_i]$ or $(E'_i = \mathbf{any}\ t$ and $\mathrm{sort}(E_i) = t))$
   $\equiv_{wbc}$ **stop** otherwise
   (b) $\mathbf{exit}(\ldots) \mid \mathbf{stop} \equiv_{wbc} \mathbf{stop}$
4. $B_1 |[A]| B_2 \equiv_{wbc} B_1 |[A']| B_2$ where A' is any list containing the same elements as A
5. $B_1 |[A]| B_2 \equiv_{wbc} B_1 |[A']| B_2$ where $A' = A \cap (\mathcal{L}(B_1) \cup \mathcal{L}(B_2))$
6. $B_1 |[A]| B_2 \equiv_{wbc} B_1 || B_2$ if $(\mathcal{L}(B_1) \cup \mathcal{L}(B_2)) \subseteq A$
7. $B_1 |[\,]| B_2 \equiv_{wbc} B_1 ||| B_2$

d) **Enabling** Let $\gg^*$ denote any instance of the enable operator, i.e. $\gg$ or $\gg$ **accept** ...**in** .

1. $\mathbf{stop} \gg^* B \equiv_{wbc} \mathbf{stop}$
2. (a) $\mathbf{exit} \gg B \equiv_{wbc} \mathbf{i};\ B$
   (b) $\mathbf{exit}(E_1,\ldots,E_n) \gg \mathbf{accept}\ x_1 : t_1,\ldots,x_n : t_n\ \mathbf{in}\ B \equiv_{wbc} \mathbf{i};\ [E_1/x_1,\ldots,E_n/x_n]B$
3. $(B_1 \gg B_2) \gg B_3 \equiv_{wbc} B_1 \gg (B_2 \gg B_3)$
4. $B \gg^* \mathbf{stop} \equiv_{wbc} B ||| \mathbf{stop}$ if $B \not\equiv_{wbc} \mathbf{exit}(\ldots)$
   Note that this side condition has been added here as a result of our experiments, see section 5.5.2; it is not in the standard.

e) **Disabling**

1. $B_1 [> (B_2 [> B_3) \equiv_{wbc} (B_1 [> B_2) [> B_3$
2. $B [> \mathbf{stop} \equiv_{wbc} B$
3. $(B_1 [> B_2)\ []\ B_2 \equiv_{wbc} B_1 [> B_2$
4. $\mathbf{stop} [> B \equiv_{wbc} B$
5. $\mathbf{exit}(\ldots) [> B \equiv_{wbc} \mathbf{exit}(\ldots)\ []\ B$

f) **Hiding**

1. $\mathbf{hide}\ A\ \mathbf{in}\ B \equiv_{wbc} \mathbf{hide}\ A'\ \mathbf{in}\ B$ where A' is any list containing the same elements as A.
2. $\mathbf{hide}\ A\ \mathbf{in}\ B \equiv_{wbc} \mathbf{hide}\ A'\ \mathbf{in}\ B$ where $A' = A \cap \mathcal{L}(B)$
3. $\mathbf{hide}\ A\ \mathbf{in}\ \mathbf{hide}\ A'\ \mathbf{in}\ B \equiv_{wbc} \mathbf{hide}\ A''\ \mathbf{in}\ B$ where $A'' = A \cup A'$
4. $\mathbf{hide}\ A\ \mathbf{in}\ B \equiv_{wbc} B$ if $A \cap \mathcal{L}(B) = \emptyset$
5. (a) $\mathbf{hide}\ A\ \mathbf{in}\ a!E_1\ldots!E_n;\ B \equiv_{wbc} \mathbf{i};\ (\mathbf{hide}\ A\ \mathbf{in}\ B)$ if $a \in A$
   (b) $\mathbf{hide}\ A\ \mathbf{in}\ g;\ B \equiv_{wbc} g;\ (\mathbf{hide}\ A\ \mathbf{in}\ B)$ if name $(g) \notin A$

6. **hide** A **in** B_1 $[]$ $B_2 \equiv_{wbc}$ (**hide** A **in** B_1) $[]$ (**hide** A **in** B_2)

7. **hide** A **in** (B_1 $|[A']|$ B_2) $\equiv_{wbc}$ (**hide** A **in** B_1) $|[A']|$ (**hide** A **in** B_2) if $A \cap A' = \emptyset$

8. **hide** A **in** ($B_1 \gg^* B_2$) $\equiv_{wbc}$ (**hide** A **in** B_1) $\gg^*$ (**hide** A **in** B_2)

9. **hide** A **in** (B_1 [> B_2) $\equiv_{wbc}$ (**hide** A **in** B_1) [> (**hide** A **in** B_2)

10. **hide** A **in** $[E]$ -> $B \equiv_{wbc}$ $[E]$ -> (**hide** A **in** B)

g) Guarding

1. (a) $[L = R] \rightarrow B \equiv_{wbc} B$ if $L = R$
   $[L = R] \rightarrow B \equiv_{wbc}$ **stop** otherwise
   (b) $[BE] \rightarrow B \equiv_{wbc} [BE = \mathit{true}] \rightarrow B$ if BE is a value-expression

h) Instantiation

$p[a_1, \ldots, a_n](E_1, \ldots, E_m) \equiv_{wbc} ([E_1/x_1, \ldots, E_m/x_m]B_p)[a_1/g_1, \ldots, a_n/g_n]$

if **process** $p[g_1, \ldots, g_n](x_1, \ldots, x_m) : f := B_p$ **endproc** is the format of the corresponding process abstraction for the process-identifier p.

j) Local Definition

let $x_1 : t_1 = E_1, \ldots, x_n : t_n = E_n$ **in** $B \equiv_{wbc} [E_1/x_1, \ldots, E_n/x_n]B$

k) Relabelling Let $[S]$ be any instance $[a_1/g_1, \ldots, a_n/g_n]$ of the post-fix relabelling operator. We associate with $[S]$ the function S on gate-identifiers defined by

$$\begin{aligned} S(g_i) &= a_i \ (1 \leq i \leq n) \\ S(g) &= g \ \text{ if } g \neq g_i (1 \leq i \leq n) \end{aligned}$$

and S maps the internal action **i** to itself.
We extend S to lists, sets, strings etc. in the obvious way.

1. **stop**$[S] \equiv_{wbc}$ **stop**

2. **exit**(...)$[S] \equiv_{wbc}$ **exit**(...)

3. $(a;\ B)[S] \equiv_{wbc} S(a);\ B[S]$

4. $(B_1\ []\ B_2)[S] \equiv_{wbc} B_1[S]\ []\ B_2[S]$

5. $(B_1\ |[A]|\ B_2)[S] \equiv_{wbc} B_1[S]\ |[S(A)]|\ B_2[S]$
   if S is injective on $\mathcal{L}(B_1) \cup \mathcal{L}(B_2) \cup A$

6. $(B_1 \gg^* B_2)[S] \equiv_{wbc} B_1[S] \gg^* B_2[S]$

7. $(B_1$ [> $B_2)[S] \equiv_{wbc} B_1[S]$ [> $B_2[S]$

8. (**hide** A' **in** B)$[S] \equiv_{wbc}$ **hide** A **in** $B[S]$
   if S is injective on $\mathcal{L}(B) \cup A'$, and $S(A') \equiv_{wbc} A$.

9. $B[S] \equiv_{wbc} B$ if S is the identity on $\mathcal{L}(B)$

10. $B[S_1] \equiv_{wbc} B[S_2]$ if $\forall a \in \mathcal{L}(B).S_1(a) = S_2(a)$

11. $B[S_1][S_2] \equiv_{wbc} B[S_2 \circ S_1]$

m) Internal Action

1. $a;\ \mathbf{i};\ B \equiv_{wbc}\ a;\ B$
2. $B\ [\,]\ \mathbf{i};\ B \equiv_{wbc}\ \mathbf{i};\ B$
3. $a;\ (B_1\ [\,]\ \mathbf{i};\ B_2)\ [\,]\ (a;\ B_2) \equiv_{wbc}\ a;\ (B_1\ [\,]\ \mathbf{i};\ B_2)$
4. $[E/x]B\ [\,]\ (\textbf{choice}\ x : t\ [\,]\ \mathbf{i};\ B) \equiv_{wbc}\ \textbf{choice}\ x : t\ [\,]\ \mathbf{i};\ B$ if $[E] \in Q(t)$

n) Expansion Theorems Let $B = [\,]\ \{b_i;\ B_i \mid i \in I\}$, $C = [\,]\ \{c_j; C_j \mid j \in J\}$

1. $$\begin{aligned} B\ |[A]|\, C \equiv_{wbc}\ & [\,]\ \{b_i;\ (B_i|[A]|C) \mid \text{name}\ (b_i) \notin A,\ i \in I\} \\ & [\,]\ [\,]\ \{c_j;\ (B|[A]|C_j) \mid \text{name}\ (c_j) \notin A,\ j \in J\} \\ & [\,]\ [\,]\ \{a;\ (B_i|[A]|C_j) \mid a = b_i = c_j, \text{name}\ (a) \in A,\ i \in I,\ j \in J\} \end{aligned}$$
   if all $b_i,\ c_j$ are of the form $g!E_1, \ldots !E_n$
2. $$\begin{aligned} B\,[> C \equiv_{wbc}\ & C \\ & [\,]\ [\,]\ \{b_i;\ (B_i\,[>\ C) \mid i \in I\} \end{aligned}$$
3. $$\begin{aligned} \textbf{hide}\ A\ \textbf{in}\ B \equiv_{wbc}\ & [\,]\ \{b_i;\ \textbf{hide}\ A\ \textbf{in}\ B_i \mid \text{name}\ (b_i) \notin A,\ i \in I\} \\ & [\,]\ [\,]\ \{\mathbf{i}; \textbf{hide}\ A\ \textbf{in}\ B_i \mid \text{name}\ (b_i) \in A,\ i \in I\} \end{aligned}$$
   if all b_i are of the form $g!E_1 \ldots !E_n$
4. $B[S] \equiv_{wbc}\ [\,]\ \{S(b_i);\ B[S] \mid i \in I\}$

B.2.4 Weak Bisimulation Equivalence Laws

The following rules may be added to the above rules for weak bisimulation congruence when dealing with weak bisimulation equivalence. We denote weak bisimulation equivalence by $\approx_{wbe}$.

1. $B \approx_{wbe} \mathbf{i}; B$
2. Let $C[\,]$ be a LOTOS context of the following forms:
   (a) $g; [\,]$
   (b) $[\,]\ |[A]|\ B$, or $B\ |[A]|\,[\,]$
   (c) $[\,] \gg^* B$, or $B \gg^* [\,]$
   (d) $[\,]\,[> B$
   (e) $\textbf{hide}\ A\ \textbf{in}\ [\,]$
   (f) $[E]$ -> $[\,]$
   (g) $[\,][S]$
   (h) $\textbf{let} \ldots \textbf{in}\ [\,]$

   then $B_1 \approx_{wbe} B_2 \ \Rightarrow\ C[B_1] \approx_{wbe} C[B_2]$,
3. $B \approx_{wbe} C \ \Rightarrow\ a; B \equiv_{wbc} a; C$, for all action-denotations a,
4. $B \equiv_{wbc}\ C \ \Rightarrow\ B \approx_{wbc} C$

B.2.5 Testing Congruence Laws

The laws for testing congruence and equivalence are expressed in the pre-orders **cred** and **red** that generate them. B_1 **red** B_2 can be interpreted as "B_1 implements B_2". **cred** is the largest subrelation of **red** that has the substitution property with respect to LOTOS contexts.

1. $\vdash B_1 \equiv_{wbc} B_2 \Longrightarrow B_1$ **cred** B_2

   Note : this means that **cred** inherits all the laws for weak bisimulation congruence.
2. B **cred** **i**; B
3. $g;(B_1 \mathbin{[]} B_2)$ **cred** $g;B_1 \mathbin{[]} g;B_2$
4. $g;B_1$ **cred** $g;B_1 \mathbin{[]} g;B_2$
5. B_1 **cred** B_2 & B_2 **cred** $B_3 \Rightarrow B_1$ **cred** B_3
6. B_1 **cred** B_3 & B_2 **cred** $B_3 \Rightarrow (B_1 \mathbin{[]} B_2)$ **cred** B_3

B.2.6 Testing Equivalence Laws

In addition to the above laws for **cred** and testing congruence, we also have the following when dealing with **red** or testing equivalence.

1. $\vdash B_1 \approx_{wbe} B_2 \Longrightarrow B_1$ **red** B_2
2. Let $C[]$ be a LOTOS context of the following forms:
   (a) $g;[]$
   (b) $[] \mid[A]\mid B$, or $B \mid[A]\mid []$
   (c) $[] \gg^* B$, or $B \gg^* []$
   (d) $[] [> B$
   (e) $[E] \rightarrow []$
   (f) $[][S]$
   (g) **let** ... **in** $[]$

   then B_1 **red** $B_2 \Rightarrow C[B_1]$ **red** $C[B_2]$,
3. B_1 **red** B_2 & B_2 **red** $B_3 \Rightarrow B_1$ **red** B_3
4. B_1 **red** B_3 & B_2 **red** $B_3 \Rightarrow (B_1 \mathbin{[]} B_2)$ **red** B_3

Appendix C

RRL Rules

C.1 Introduction

In this appendix we give the actual RRL input files that were used for the experiments detailed in chapters 6 and 7. Since the presentation of the rules there was LOTOS like, rather than the form required by RRL, we begin by giving the mapping between the usual LOTOS syntax and the syntax required by RRL. Comments in the RRL input files are preceded by *;;*.

The RRL Algebra

The sorts used in implementing LOTOS are: *process, event, gatelist, rlist* (relabelling list) and the universal type, *univ*, which is built into RRL. In addition to the algebra definition, we also include some subtyping information: all list types are a subtype of *list.* For the arities of the functions see the function declarations in the input files. The way these functions are used to implement the corresponding LOTOS ones is given in the next section.

Representation of LOTOS constructs

Operators represented are given in table C.1; only Basic LOTOS has been considered.

RRL requires alphabetic function names rather than operator symbols. We also adopt prefix style for function application since RRL does not support mixfix parsing. Note that all variables in RRL must start with u, v, w, x, y or z. We try to be consistent in using x, y and z for processes, v for lists and u for events.

We now give the RRL input files which were used for the experiments in chapters 6 and 7.

Operator Name	LOTOS	RRL
emptylist	[ ]	nl
inaction	**stop**	stop
termination	**exit**	exit
internal action	**i**	i
action prefix	a; B	seq (u, x)
choice	$B_1 [] B_2$	ch (x, y)
general parallelism	$B_1 \mid[A]\mid B_2$	par (x, y, v)
interleaving	$B_1 \mid\mid\mid B_2$	par (x, y, nl)
enabling	$B_1 \gg B_2$	en (x, y)
disabling	$B_1 [> B_2$	dis (x, y)
hiding	**hide** A **in** B	hide (v, x)
relabelling	$B[S]$	relabel (x, v)

Table C.1: List of LOTOS expressions and corresponding RRL expressions

C.2 Algebra

This is the algebra for the LOTOS operators used in the completion experiments, i.e. in conjunction with the rule set *core*.

seq	: *event, process* $\rightarrow$ *process*
ch	: *process, process* $\rightarrow$ *process*
par	: *process, process, gatelist* $\rightarrow$ *process*
en	: *process, process* $\rightarrow$ *process*
dis	: *process, process* $\rightarrow$ *process*
hide	: *gatelist, process* $\rightarrow$ *process*
relabel	: *process, rlist* $\rightarrow$ *process*
exit	: *process*
stop	: *process*
i	: *event*
nl	: *list*
cons	: *univ, list* $\rightarrow$ *list*

list < *rlist*
list < *gatelist*

The use of < allows us to define, for example, *list*, generally, with all supertypes of *list* inheriting the properties and functions of *list*.

In addition to these declarations, *ch* is declared to be commutative. Function precedences are as described in chapter 6.

C.3 Core Rules

This is the set of rules, referred to as *core* in chapter 6, derived from the weak bisimulation laws of LOTOS which will give a confluent and terminating rule set when run through the completion algorithm.

```
;; choice is declared as commutative
ch(x, stop) == x
ch(x, x) == x

par (exit, exit, v) == exit
par(exit, stop, nl) == stop

en(stop, x) == stop
en(exit, x) == seq(i, x)
en(x, en(y, z)) == en(en(x, y), z)

dis(x, stop) == x
dis(stop, x) == x
ch(dis(x, y), y) == dis(x, y)
dis (exit, x) == ch (exit, x)

relabel(v, stop) == stop
relabel(v, exit) == exit

seq(u, seq(i, x)) == seq(u, x)
ch(x, seq(i, x)) == seq(i, x)
ch (seq (u, ch (x, seq (i, y))), seq (u, y)) == seq (u, ch (x, seq (i, y)))
```

Other attempts to extend the rule set (sometimes giving a complete set, but mainly giving divergent rule sets) use the following rules:

```
hide (v, en(x,y)) == en(hide(v,x), hide(v,y))
hide(v,stop) == stop
hide(v,exit) == exit

dis(x, dis(y,z)) == dis(dis(x,y),z)
```

We also added associativity for choice, but this was done by declaring *ch* to be associative, rather than by giving the explicit rule.

For the case study, these rules are not sufficient. The remaining sections detail the new rules implemented for the case study proofs. These rules relate mostly to the introduction of the expansion laws for parallelism and hide. The resulting rule set is not complete.

We begin with the auxiliary data structures required, sets and lists.

C.4 Sets and Lists

Only a few set and list operations are required by the expansion rules.

```
{} : set
s : univ → set
++ : set,set → set

x ++ x == x
x ++ {} == x
```

The set constructor ++ is declared to be a-c.

member : univ, list → bool

member(u, nl) == false
member(u, cons(u, xs)) == true
member(u, cons(v, xs)) == member(u, xs) if not(u = v)

;;assume no duplicates
equal : list,list → bool
equal (x cons xs, ys) == equal (xs, delete (x, ys)) if member (x, ys)
equal (x cons xs, ys) == false if not (member (x, ys))
equal (nl, nl) == true

delete : univ, list → list
delete (x, x cons ys) == delete (x, ys)
delete (x, y cons ys) == y cons delete (x, ys) if not (x = y)
delete (x, nl) == nl

C.5 Generalised Choice and Parallelism

To implement the expansion law more easily, two new constructs are introduced: generalised choice, i.e. choice over a set of processes, and an auxiliary parallel operator.

Generalised choice notation is used to express the expansion laws. This operator is only a shorthand for many occurrences of binary choice and is not a part of the LOTOS syntax. Note that since the ++ operator is associative and commutative, we also get associativity and commutativity for generalised choice. In the following *procset* denotes a set of processes.

gch : procset → process
set < procset

ch (x, y) == gch (s(x) ++ s(y))

;; if a choice in a set is stop, then we can delete it, law b3.
gch(s(stop) ++ xs) == gch(xs)

;; we can eliminate nested gch
gch(ys ++ s(gch(xs))) == gch(xs ++ ys)

;; gch of a single element is just the element
gch(s(x)) == x

;;gch of an empty list is stop
gch({}) == stop

The last three rules ensure an uncluttered normal form for *gch*.

The parallel operator causes considerable problems in the RRL framework, because it is a ternary operator between two processes and a list of gate names that acts like an associative and commutative binary operator between two processes. In order to alleviate this problem, we introduce the auxiliary operator *pp*, which forms a process pair from two processes. The *par* operator is then redefined to take a process pair and a gate list. By declaring *pp* to be commutative we model the commutativity of *par*. Unfortunately we cannot do the same for associativity as the occurrences of *pp* are not adjacent, i.e. we have *par(pp(par(pp(x, y), v), z), w)*. However, associativity of the parallel operators only applies when both synchronisation lists are the same, therefore, we may not actually use associativity very often.

The new definitions are as follows:

```
par   : procpair, gatelist → process
pp    : process, process → procpair
```

C.6 Case Study Constants

Each constant in the Login case study of chapter 7 must be defined as different from all the others; this is achieved by mapping events into the naturals.

```
m1 : event          p1 : event          n1 : event          i       : event
m2 : event          p2 : event                              delta   : event
m3 : event          p3 : event          n3 : event          timeout : event
m4 : event          p4 : event          n4 : event          tcancel : event
m5 : event          p5 : event                              set     : event
m6 : event          p6 : event
m7 : event          p7 : event
```

```
0 : nat
f : nat → nat
c : nat → event
```

```
(f(x) = f(y)) == (x = y)
(f(x) = 0) == false
(c(x) = c(y)) == (x = y)
```

```
i == c(0)
delta == c(f(0))
m1 == c(f(f(0)))
m2 == c(f(f(f(0))))
m3 == c(f(f(f(f(0)))))
m4 == c(f(f(f(f(f(0))))))
m5 == c(f(f(f(f(f(f(0)))))))
m6 == c(f(f(f(f(f(f(f(0))))))))
m7 == c(f(f(f(f(f(f(f(f(0)))))))))
p1 == c(f(f(f(f(f(f(f(f(f(0))))))))))
p2 == c(f(f(f(f(f(f(f(f(f(f(0)))))))))))
p3 == c(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))
p4 == c(f(f(f(f(f(f(f(f(f(f(f(f(0)))))))))))))
p5 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))))
p6 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0)))))))))))))))
```

p7 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))))))
n1 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0)))))))))))))))))
n3 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))))))))
n4 == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0)))))))))))))))))))
set == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))))))))))
tcancel == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0)))))))))))))))))))))
timeout == c(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(0))))))))))))))))))))))

When added to RRL the following precedences must also be set: timeout > tcancel > set > m1 > m2 > m3 > m4 > m5 > m6 > m7 > p1 > p2 > p3 > p4 > p5 > p6 > p7 > i > delta > c > f > 0

C.7 Hide Expansion Rules

These rules implement the hide expansion law and also describe how **hide** interacts with *gch*. These rules are derived from laws n3, f4 and f6.

hide (v, gch(s(x) ++ xs)) == gch (s(hide(v, x)) ++ s(hide(v, gch(xs))))

hide (v, exit(nl)) == exit(nl)
hide (v, seq (u, x)) == seq (u, hide (v, x)) if not (member (u, v))
hide (v, seq (u, x)) == seq (i, hide (v, x)) if (member (u, v))

C.8 Parallel Expansion Rules

The expansion law for parallelism is implemented by the following functions:

expanda : procset, procset, gatelist → procset
expandb : procset, procset, gatelist → procset
expandc : process, procset, gatelist → procset

An overview of how these functions operate is given in section 6.4.1. The actual RRL rules are given below.

The general form of the expansion rules is:

par (pp(gch(x),gch(y)), v) == gch(expanda(x,y,v) ++ expanda(y,x,v) ++ expandb(x,y,v))

We must also define cases for each function for the situation in which one process is a simple process with either *seq*, *stop* or *exit* at the top level, and the other process has *gch* at the top level.

```
par(pp(gch(xs), seq(u,x)), v) == gch(expanda(xs, s(seq(u,x)), v) ++
                                    expanda(s(seq(u, x)), xs, v) ++
                                    expandb(xs, s(seq(u,x)), v))
par(pp(gch(xs), exit(v1)), v2) == gch(expanda(xs, s(exit(v1)), v2) ++
                                    expanda(s(exit(v1)), xs, v2) ++
                                    expandb(xs, s(exit(v1)), v2))
par(pp(gch(xs), stop), v) == gch(expanda(xs, s(stop), v) ++
                                    expanda(s(stop), xs, v) ++
                                    expandb(xs, s(stop), v))
```

Finally we have to cater for the situation in which both sides of the parallel operator are simple processes.

```
par(pp(seq(u1, x1),seq(u2, x2)), v) == gch(expanda (s(seq(u1, x1)), s(seq(u2, x2)), v) ++
                                           expanda (s(seq(u2, x2)), s(seq(u1, x1)), v) ++
                                           expandb (s(seq(u1, x1)), s(seq(u2, x2)), v))
par(pp(seq(u, x), exit(v1)), v2) == gch(   expanda (s(seq(u, x)), s(exit(v1)), v2) ++
                                           expanda (s(exit(v1)), s(seq(u, x)), v2) ++
                                           expandb (s(seq(u, x)), s(exit(v1)), v2))
par(pp(seq(u, x), stop), v) == gch(        expanda (s(seq(u, x)), s(stop), v) ++
                                           expanda (s(stop), s(seq(u, x)), v) ++
                                           expandb (s(seq(u, x)), s(stop), v))
par(pp(exit(v1), exit(v1)), v2) == exit(v1)
par(pp(exit(v1), exit(v2)), v3) == exit(v1) if equal(v1, v2)
par(pp(exit(v1), exit(v2)), v3) == stop if not (equal(v1, v2))
par(pp(exit(v1), stop), v2) == stop
par(pp(stop, stop), v) == stop
```

We now define the functions *expanda*, *expandb* and *expandc*.

```
;;Rules for expanda
expanda (s(seq(u, x)) ++ xs, ys,v) == s(seq (u, par (pp(x, gch (ys)), v))) ++
                                      expanda (xs,ys,v) if not(member(u,v))
expanda (s(seq(u, x)) ++ xs, ys,v) == expanda(xs,ys,v) if member(u,v)
expanda (s(seq(u, x)), ys, v) == s(seq(u, par(pp(gch(s(x)), gch(ys)), v))) if not (member (u, v))
expanda (s(seq(u, x)), ys, v) == {} if member(u, v)
;; exit must synchronise with exit - it may not proceed on its own
expanda (s(exit(v1)) ++ xs, ys, v2) == expanda (xs, ys, v2)
expanda (s(exit(v1)), ys, v2) == {}
;; stop may not synchronise with anything or proceed on its own
expanda (s(stop) ++ xs, ys, v) == expanda (xs, ys, v)
expanda (s(stop), ys, v) == {}
expanda({},xs,v) == {}

;;Rules for expandb
;; apply expandc to a choice from the first set, together with the second set.
;; apply expandb down the rest of the first set.
expandb (s(x) ++ xs, ys, v) == expandc(x,ys,v) ++ expandb(xs,ys,v)
expandb (s(x), ys, v) == expandc(x, ys, v)
expandb({},ys,v) == {}

;;Rules for expandc
expandc (seq(u1,x1), (s(seq(u1,x2)) ++ ys), v) == s(seq(u1, par(pp(x1, x2),v))) ++
                                                  expandc(seq(u1, x1), ys,v) if member(u1, v)
expandc (seq(u1,x1), s(seq(u1,x2)) ++ ys, v) == {} if not member(u1, v)
expandc (seq(u1,x1), s(seq(u2,x2)) ++ ys, v) == expandc (seq(u1,x1), ys,v) if not (u1 = u2)
expandc(seq(u1, x1), s(seq(u1, x2)), v) == s(seq(u1, par(pp(x1, x2), v))) if member (u1, v)
expandc(seq(u1, x1), s(seq(u1, x2)), v) == {} if not (member(u1, v))
expandc(seq(u1, x1), s(seq(u2, x2)), v) == {} if not (u1 = u2)
expandc (seq(u1, x1), s(exit(v1)) ++ ys, v3) == expandc (seq(u1,x1), ys, v3)
expandc (seq(u1, x1), s(exit(v1)), v3) == {}
expandc (seq(u1, x1), s(stop) ++ ys, v3) == expandc (seq(u1, x1), ys, v3)
expandc (seq(u1, x1), s(stop), v3) == {}
expandc (exit(v1), s(seq(u1, x1)) ++ ys, v2) == expandc (exit(v1), ys, v2)
expandc (exit(v1), s(exit(v3)) ++ ys, v2) == s(par(pp(exit(v1), exit(v3)), v2))
expandc (exit(v1), s(stop) ++ ys, v3) == expandc (exit(v1), ys, v3)
expandc(exit(v1), s(seq(u, x)), v2) == {}
expandc(exit(v1), s(exit(v2)), v3) == s(par(pp(exit(v1), exit(v2)), v3))
expandc (exit(v1), s(stop), v3) == {}
expandc (stop, ys, v) == {}
expandc (x, {}, v) == {}
```

Appendix D

PAM Input Files

This appendix contains the PAM input files used for the examples in chapter 9. Comments in the input files are preceded by --. Note that in some cases, only a portion of the input file is given, this is to avoid excessive repetition; the accompanying text details the parts omitted.

D.1 Main LOTOS "Axioms"

This is the input file usually used in conjunction with LOTOS. The main operators are declared here, together with the PAM axioms, derived from the laws given in [BIN92], which allow the derived operators to be converted into choice and sequencing. Also included are the so-called "τ-laws" for weak bisimulation congruence, branching bisimulation, testing congruence and trace equivalence.

```
-- Basic LOTOS
-- Axioms for rewriting LOTOS terms based on BIN:92

signature
type Gate Action Process
with Gate < Action
operator
    i           ::  -> Action
    delta       ::  -> Action
    stop        ::  -> Process
    exit        ::  -> Process
    _ . _       ::  Action Process -> Process 150 RIGHT
    _ [] _      ::  Process Process -> Process 130 AC LEFT
    _ >> _      ::  Process Process -> Process 120 LEFT
    _ [> _      ::  Process Process -> Process 120 LEFT
    hide _ on _ ::  Gate set Process -> Process 30 RIGHT
    _ |[ _ ]| _ ::  Process Gate set Process -> Process 100 AC LEFT
    _ ||| _     ::  Process Process -> Process 100 AC LEFT
```

```
axiom
-- CH1 and CH2 AC axioms, covered by operator declaration
CH3     x [] stop = x
CH4     x [] x = x

-- these first three are covered by the expansion law if DELTA
-- is used, but it may be more convenient to use these forms.
P1      exit ||| exit = exit
P2      exit |[s]| exit = exit
P3      exit |[s]| stop = stop
P4      x ||| y = x |[{}]| y
P5      stop |[s]| stop = stop

H1      hide A on stop = stop
H2      hide A on (x [] y) = (hide A on x) [] (hide A on y)
H3A     hide A on a.x = a.(hide A on x)                      if not(a in A)
H3B     hide A on a.x = i.(hide A on x)                        if (a in A)

E1      stop >> x = stop
E2      (x [] y) >> z = (x >> z) [] (y >> z)
E3A     a.x >> y = a.(x >> y)                             if not (a eq delta)
E3B     a.x >> y = i.y                                        if (a eq delta)

D1      stop [> x = x
D2      (x [] y) [> z = (x [> z) [] (y [> z)
D3A     a.x [> y = y [] a.(x [> y)                        if not (a eq delta)
D3B     a.x [> y = a.x [] y                                   if (a eq delta)

DELTA   exit = delta.stop

-- add these for observation congruence
OBS1    a.i.x = a.x
OBS2    x [] i.x = i.x
OBS3    a.(x [] i.y) [] a.y = a.(x [] i.y)

-- add these for branching bisimulation congruence
BB1     a.i.x = a.x
BB2     a.(i.(x [] y) [] x) = a.(x [] y)

-- add these for testing congruence
TEST1   i.(x [] y) [] i.y = i.y [] x
TEST2   a.(i.x [] i.y) = a.x [] a.y
TEST3   i.(a.x [] a.y [] z) = a.x [] i.(a.y [] z)

-- add these for trace congruence
TR1     i.x = x
TR2     a.(x [] y) = a.x [] a.y

expansion law
EXP let x = a1.x1 [] ...  [] an.xn     y = b1.y1 [] ...  [] bm.ym
then
    (x |[A]| y) = stop     if (sync_move(x,y) eq nil) and (async_move(x,y) eq nil)
    (x |[A]| y) = Sum([],async_move(x,y))                 if sync_move(x,y) eq nil
    (x |[A]| y) = Sum([],sync_move(x,y))                 if async_move(x,y) eq nil
    (x |[A]| y) = Sum([],async_move(x,y)) [] Sum([],sync_move(x,y))     otherwise
with communication function
broadcast
    sync(a, b) = a     if (a eq b) and not(a eq i) and ((a in A) or (a eq delta))
    async(a) = true                             if not((a in A) or (a eq delta))

sort computation
    Sort(stop) = {}
    Sort(i.P) = Sort(P)
    Sort(a.P) = {a} union Sort(P)
    Sort(P [] Q) = Sort(P) union Sort(Q)
    Sort(P ||| Q) = Sort(P) union Sort(Q)
    Sort(P |[A]| Q) = Sort(P) union Sort(Q)
    Sort(hide A on P) = Sort(P) diff A
end
```

D.2 Extra LOTOS "Axioms"

Occasionally the above laws are not enough to complete a proof, c.f. the scheduler example in section 9.5. Below is a selection of the PAM axioms, derived from the laws of the LOTOS standard [ISO88], which may be useful for particular proofs; these axioms should be used in conjunction with the declarations and axioms of the previous section, therefore this section cannot be used on its own as a PAM input file. Also included in this selection is the declaration for the relabelling operator, and the associated laws.

Each group of axioms/laws is denoted by the labels used in the standard, prefixed by an S, otherwise some confusion with the axioms of the previous section may arise. We discuss each law individually, noting ones which are either not Basic LOTOS, in which case they are ignored, or which already occur in the previous set of PAM axioms.

```
-- PAM axioms for rewriting Basic LOTOS terms based on the laws of
-- the LOTOS standard.

-- Relabelling is added to the usual definitions and declarations.
    _ [ _ / _ ] ::  Process Gate Gate -> Process 300 -- relabelling

-- Axioms
-- First, axioms which are a straight translation of the laws in the LOTOS standard.

-- a (action-prefix)
-- not Basic LOTOS.

-- b (choice)
-- laws 1 and 2 covered by standard declaration:  choice AC.
-- laws 3 and 4 covered by standard laws CH3 and CH4.
-- stop as a zero and choice idempotent.
-- laws 5 - 7 not Basic LOTOS.
-- c (parallelism)
-- laws 1 and 2 covered by standard laws:  parallelism AC.
-- law 3a covered by standard laws P1 and P2.
-- law 3b covered by standard law P3.
-- law 4 unnecessary as gate sets are used rather than gate lists.
-- law 6 unnecessary as full synchronisation is not used.
-- law 7 covered by P4.

SC5     x |[s]| y = x |[t]| y           if t eq (s inter (Sort(x) union Sort(y)))

-- d (enable)
-- law 1 covered by standard law E1.
-- law 3 not Basic LOTOS.

SD2     exit >> x = i.x
SD4     (x >> y) >> z = x >> (y >> z)
SD5     x >> stop = x ||| stop                               if not (x eq exit)
-- the side condition must be added as otherwise this law leads
to a contradiction, see section 5.5.2.

-- e (disable)
-- law 4 covered by standard law D1.

SE1     x [> (y [> z) = (x [> y) [> z
SE2     x [> stop = x
SE3     (x [> y) [] y = x [> y
SE5     exit [> x = exit [] x
-- this axiom is covered by the laws D2 and D3B, but this form is more convenient.

-- f (hiding)
-- law 1 unnecessary as sets are used rather than lists.
-- laws 5a, 5b and 6 covered by axioms H3A, H3B and H2.
-- law 10 not Basic LOTOS.

SF2     hide A on x = hide (A inter Sort(x)) on x
SF3     hide A on hide A' on x = hide (A union A') on x
SF4     hide A on x = x                              if ({} eq (A inter Sort(x)))
-- standard law H1 is an instance of this.
SF7     hide A on (x |[s]| y) = (hide A on x) |[s]| (hide A on y)
                                                           if ({} eq (A inter s))
SF8     hide A on (x >> y) = (hide A on x) >> (hide A on y)
SF9     hide A on (x [> y) = (hide A on x) [> (hide A on y)

-- g (guards)
-- not Basic LOTOS.

-- h (instantiation)
-- Since parameters are ignored, this is implemented by the PAM
-- unfold function.

-- j
-- not Basic LOTOS.
-- k (relabelling)
-- Note that these are simplified as relabelling cannot be implemented
-- as a function in PAM, instead we use a pair of gate names.
-- laws 10 and 11 are redundant under this implementation of
-- relabelling.

SK1     stop [a/b] = stop
SK2     exit [a/b] = exit
SK3A    (c.x)[a/b] = c.(x[a/b])                              if not (c eq b)
SK3B    (b.x)[a/b] = a.(x[a/b])
SK4     (x [] y)[a/b] = x[a/b] [] y[a/b]
SK5     (x |[s]| y)[a/b] = x[a/b] |[((s diff {a}) union {b})]| y[a/b]
-- if more than one relabelling pair is present, the user must ensure
-- that the relabelling function as a whole is injective.
SK6     (x >> y)[a/b] = x[a/b] >> y[a/b]
SK7     (x [> y)[a/b] = x[a/b] [> y[a/b]
SK8     (hide A on x)[a/b] = hide ((A diff {a}) union {b}) on x[a/b]
-- as with SK5, the relabelling function as a whole must be injective.
SK9     x[a/a] = x
SK11    x[a/b] [c/d] = x[c/d] [a/b]

-- m (internal action)
-- laws 1 - 3 covered by the axioms OBS1, OBS2 and OBS3.
-- laws 4 not Basic LOTOS.

-- n (expansion laws)
-- law 1 implemented by PAM expansion law
-- law 2 implemented by group of standard axioms D.
-- law 3 implemented by group of standard axioms H.
-- law 4 implemented by group of laws above SK1 - SK4.

-- Some other axioms, also derived from the laws of the LOTOS standard,
-- which were added for particular proofs.

H4      hide A on hide A on x = hide A on x
H5      hide A on hide A' on x = hide A' on hide A on x

ST      x |[Sort(x)]| stop = stop
NEWEXP  (x [] y) |[s]| z = (x |[s]| z) [] (y |[s]| z)
-- this is added because the PAM expansion law insists on all processes
-- being unfolded before expansion can take place, but sometimes we don't
-- want every branch of a process to be expanded.
```

D.3 "Axioms" for the LOTOS cred Preorder

In some relations, rather than viewing the implementation as *equivalent* to the specification, we wish to show that the implementation *approximates* the behaviour of the specification. This can be modelled by the use of the **cred** preorder. As described in section 8.4.2, **cred** is implemented as a predicate, but axiomatised in the style of an equivalence. No restraints are placed on the application of **cred** axioms by PAM; it is up to the user to apply the rules in the manner advised in section 8.4.2. As with the axioms of the previous section, the following axioms are designed to be added to the axioms and declarations of appendix D.1; they cannot be used as a PAM input file on their own. These PAM axioms were used in the Theracl proof, section 9.2.3.

```
-- in addition to the usual declarations:
    _ cred _ ::  Process Process -> Bool 20

-- the cred axioms
-- for these rules we assume the conjecture is of the form (A cred B) = true

-- base cases
CBASE2   A cred i.A = true
CBASE3   a.(B1 [] B2) cred a.B1 [] a.B2 = true
CBASE4   a.B1 cred a.B1 [] a.B2 = true
CBASE7   B cred B = true

-- cred rules which can be applied as axioms,
-- but only to B in (A cred B) = true
-- and only right to left.

CRED2    A = i.A
CRED3    a.(B [] C) = a.B [] a.C
CRED4    a.C = a.B [] a.C

-- axiom CRED1 comes by allowing terms to be reduced by OBS rules.
-- axiom CRED5, transitivity, is implicit.
-- axiom CRED6 allowing implementations to be combined
-- cannot be implemented in this setting.
```

D.4 The Login Case Study

```
-- Login Case Study (recursive version) --

-- Protocols describing interfaces between entities, giving the specification.
-- Processes decribing entities, giving the implementation.
-- Constraints describing some extra information essential to the specification.

conjecture
    PROCESSES = PROTOCOLS
where

PROCESSES = ((B |[first]| A) |[second]| C) |[third]| D

A = m1.((n1.A) [] (p1.A))
B = m1.m3.(   (n3.n1.B)
           [] (p3.m4.(   (n4.m6.p6.n1.B)
                      [] (p4.set.(   (timeout.m6.p6.m7.p7.n1.B)
                                  [] (m5.tcancel.p5.m6.p6.m7.p7.p1.B))))))
C = m3.(n3.C [] p3.m6.p6.C)
D = m4.(n4.D [] p4.(m5.p5.m7.p7.D [] m7.p7.D))
```

```
-- protocols
P1 = m1.(n1.P1 [] p1.P1)
P2 = m3.(n3.P2 [] p3.(P2 [] m6.p6.P2))
P3 = m4.(n4.P3 [] p4.(P3 [] m7.p7.P3 [] m5.p5.(P3 [] m7.p7.P3)))

-- constraints
TIMER     = p4.set.(m5.tcancel.TIMER [] timeout.TIMER)
DEALLOC_C = m3.(p3.m6.p6.DEALLOC_C [] n3.DEALLOC_C)
DEALLOC_D = m4.(p4.m7.p7.DEALLOC_D [] n4.DEALLOC_D)

SYSTEM     = m1.(m5.p1.SYSTEM [] n3.n1.SYSTEM [] n4.n1.SYSTEM [] timeout.n1.SYSTEM)

ORDERT1     = tcancel.p1.ORDERT1
ORDER3467   = m3.( p3.m4.( p4.m6.m7.p7.ORDER3467
                          [] n4.m6.ORDER3467)
                 [] n3.ORDER3467)
ORDER13467  = m1.m3.( n3.n1.ORDER13467
                    [] p3.m4.( n4.m6.p6.n1.ORDER13467
                             [] p4.m6.p6.m7.p7.( n1.ORDER13467
                                              [] p1.ORDER13467)))
ORDER4567   = m4.( p4.( m6.p6.m7.p7.ORDER4567
                      [] m5.p5.m6.p6.m7.p7.ORDER4567)
                 [] n4.m6.p6.ORDER4567)
ORDER1345   = m1.m3.( n3.n1.ORDER1345
                      [] p3.m4.( n4.n1.ORDER1345
                               [] p4.(m5.p1.ORDER1345
ORDER167    = m1.( m6.p6.( m7.p7.( n1.ORDER167 [] p1.ORDER167)
                             [] n1.ORDER167)
                       [] n1.ORDER167)
ORDER134    = m1.m3.( p3.m4.(n4.n1.ORDER134 [] p4.(n1.ORDER134 [] p1.ORDER134))
                      [] n3.n1.ORDER134)
ORDERTIME   = n4.m6.ORDERTIME [] p4.set.( tcancel.p5.m6.ORDERTIME
                                         [] timeout.m6.ORDERTIME)

PROTOCOLS = ((((((P2 |[second]| DEALLOC_C) |[{m3, n3, p3, m6}]| ORDER3467)
            |[{m4, n4, p4, m6, p6, m7, p7}]|
            ((P3 |[{m4, p4, n4, m7, p7}]| DEALLOC_D) |[third]| ORDER4567))
            |[{m3, n3, p3, m4, n4, p4, m6, p6, m7, p7}]| ORDER13467)
            |[{m1, n1, p1, m3, n3, p3, m4, n4, p4, m5}]|
            ((((P1 |[first]| SYSTEM) |[{m1, n1, p1, n3, n4, m5}]| ORDER1345)
            |[{m5, p4, timeout}]| TIMER) |[{p1, tcancel}]| ORDERT1))
            |[{m4, n4, p4, m5, p5, m6, p6, m7, p7}]| ORDER4567)
            |[{p4, n4, set, tcancel, timeout, p5, m6}]| ORDERTIME

macro
    first = {m1, p1, n1}
    second = {m3, n3, p3, m6, p6}
    third = {m4, n4, p4, m5, p5, m7, p7}
end
```

D.5 The Simple Radiation Machine

Several variants of the radiation machine example were presented in section 9.2; the input files for the first variant, Therac1, is given in its entirety here, as is the input file for Therac2. For all the other variants, only the **Therac** part of the definition is given; the complete input file is formed by taking the conjecture, test process, macros and rules from **Therac1**. For all of the radiation machine examples, the alphabet is: `lb` — low beam, `hb` — high beam, `ls` — low shield, `hs` — high shield, `el` — electron beam treatment, `xr` — xray beam treatment, and `fire` — fire!

D.5.1 Therac1

```
-- Therac-25 case study, due to M. Thomas.
-- Therac1

conjecture
    testok.exit = hide therac_events on UNSAFETEST
where

-- definition of the therac machine
Therac1 = STARTUP

STARTUP = SETUPL >> TREATMENT

SETUPL = (lb.exit) ||| (ls.exit)
SETUPH = (hb.exit) ||| (hs.exit)

TREATMENT = xr.XRAY [] el.ELECTRON [] exit

ELECTRON = (fire.TREATMENT) [> TREATMENT

XRAY = (SETUPH >> (fire.SETUPL) >> TREATMENT) [> TREATMENT

-- definition of the test processes
UNSAFETEST = STARTUP |[therac_events]| ((lb.ls.TEST) >> testok.exit)

TEST = Nothbhs >> (hb.Notlbhs) >> (fire.exit)

Nothbhs =     fire.Nothbhs
           [] lb.Nothbhs
           [] ls.Nothbhs
           [] xr.Nothbhs
           [] el.Nothbhs
           [] exit

Notlbhs =     ls.Notlbhs
           [] hb.Notlbhs
           [] xr.Notlbhs
           [] el.Notlbhs
           [] exit

macro
          therac_events = {lb, hb, ls, hs, fire, xr, el}

rule DIS  = *DELTA; *{D2 D3A D3B}; *{DELTA<}
rule EN   = *DELTA; *{E2 E3A E3B}; *{DELTA<}
rule HIDE = *DELTA; *{H1 H2 H3A H3B}; *{DELTA<}
rule INT  = *DELTA; *P4; *EXP; *P5; DELTA<
end
```

D.5.2 Simple Therac

```
-- Therac-25 case study, due to M. Thomas.
-- simplified version in which the interrupts are totally removed.

-- definition of the therac machine
Theracl = STARTUP

STARTUP = SETUPL >> TREATMENT

SETUPL = (lb.exit) ||| (ls.exit)
SETUPH = (hb.exit) ||| (hs.exit)

TREATMENT = xr.XRAY [] el.ELECTRON [] exit

ELECTRON = fire.TREATMENT
-- old version ELECTRON = (fire.TREATMENT) [> TREATMENT

XRAY = (SETUPH >> (fire.SETUPL) >> TREATMENT)
-- old version XRAY = (SETUPH >> (fire.SETUPL) >> TREATMENT) [> TREATMENT

-- definition of the test processes as in section D.5.1.
```

D.5.3 Modified Theracl — Version A

```
-- Therac-25 case study, due to M. Thomas.
-- Theracl with new version of SETUP without interleaving.
-- SETUPH has events in order hb, hs; therefore Theracla is unsafe.

-- definition of the therac machine
Theracl = STARTUP

STARTUP = SETUPL >> TREATMENT

SETUPL = lb.ls.exit
SETUPH = hb.hs.exit

TREATMENT = xr.XRAY [] el.ELECTRON [] exit

ELECTRON = (fire.TREATMENT) [> TREATMENT

XRAY = (SETUPH >> (fire.SETUPL) >> TREATMENT) [> TREATMENT

-- definition of the test processes as in section D.5.1.
```

D.5.4 Modified Theracl — Version B

```
-- Therac-25 case study, due to M. Thomas.
-- Theracl with new version of SETUP without interleaving.
-- SETUPH has events in order hs, hb; therefore Theraclb is safe.

-- definition of the therac machine
Therac1 = STARTUP

STARTUP = SETUPL >> TREATMENT

SETUPL = lb.ls.exit
SETUPH = hs.hb.exit

TREATMENT = xr.XRAY [] el.ELECTRON [] exit

ELECTRON = (fire.TREATMENT) [> TREATMENT

XRAY = (SETUPH >> (fire.SETUPL) >> TREATMENT) [> TREATMENT

-- definition of the test processes as in section D.5.1.
```

D.5.5 Theracld

```
-- Therac-25 case study, due to M. Thomas.
-- alterations from Therac1:  new version of SETUP with no interleaving,
-- choice and sequencing substituted for disable.

-- definition of the therac machine
Therac1 = STARTUP

STARTUP = SETUPL >> TREATMENT

SETUPL = lb.ls.exit
SETUPH = hs.hb.exit

TREATMENT = xr.XRAY [] el.ELECTRON [] exit

ELECTRON = fire.TREATMENT [] TREATMENT

XRAY =    TREATMENT
       [] hs.(   TREATMENT
              [] hb.(   TREATMENT
                     [] fire.(   TREATMENT
                              [] ls.(TREATMENT [] lb.TREATMENT))))

-- definition of the test processes as in section D.5.1.
```

D.5.6 Therac2

```
-- Therac-25 case study, due to M. Thomas.
-- This is the translated version of therac2 (i.e.  plus data types)
-- Note that because of the use of .  instead of ; here, the translation
-- is slightly different from that in section 10.3.2.
-- Gates and values are represented g_u_v_w rather than g.u.v.w.
```

```
conjecture
    testok.exit =
    hide (therac_events union fire_events union delta_events) on UNSAFETEST
where

-- definition of the therac machine
TREATMENTLowDown  = xr.XRAYLowDown [] el.ELECTRONLowDown [] d_Low_Down.stop
TREATMENTLowUp    = xr.XRAYLowUp [] el.ELECTRONLowUp [] d_Low_Up.stop
TREATMENTHighDown = xr.XRAYHighDown [] el.ELECTRONHighDown [] d_High_Down.stop
TREATMENTHighUp   = xr.XRAYHighUp [] el.ELECTRONHighUp [] d_High_Up.stop

-- electron treatments
ELECTRONLowDown  = hide {lb, hb, ls, hs} on
                       (FIRELowDown >> TREATMENTLowDown) [] TREATMENTLowDown
ELECTRONLowUp    = hide {lb, hb, ls, hs} on
                       (FIRELowUp >> TREATMENTLowUp) [] TREATMENTLowUp
ELECTRONHighDown = hide {lb, hb, ls, hs} on
                       (FIREHighDown >> TREATMENTHighDown) [] TREATMENTHighUp
ELECTRONHighUp   = hide {lb, hb, ls, hs} on
                       (FIREHighUp >> TREATMENTHighDown) [] TREATMENTHighDown

-- xray treatments
XRAYLowDown    = hide {lb, hb, ls, hs} on
    (   TREATMENTLowDown
     [] hb.(   TREATMENTHighDown
            [] hs.(   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb.(   TREATMENTLowUp
                                             [] ls.TREATMENTLowDown))))))
XRAYLowUp      = hide {lb, hb, ls, hs} on
    (   TREATMENTLowUp
     [] hb.(   TREATMENTHighUp
            [] hs.(   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb.(   TREATMENTLowUp
                                             [] ls.TREATMENTLowDown))))))
XRAYHighDown   = hide {lb, hb, ls, hs} on
    (   TREATMENTHighDown
     [] hb.(   TREATMENTHighDown
            [] hs.(   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb.(   TREATMENTLowUp
                                             [] ls.TREATMENTLowDown))))))
XRAYHighUp     = hide {lb, hb, ls, hs} on
    (   TREATMENTHighUp
     [] hb.(   TREATMENTHighUp
            [] hs.(   TREATMENTHighUp
                   [] (FIREHighUp >> (   TREATMENTHighUp
                                      [] lb.(   TREATMENTLowUp
                                             [] ls.TREATMENTLowDown))))))

FIRELowDown    = ZAPLowDown
FIRELowUp      = ERROR55
FIREHighDown   = ERROR54
FIREHighUp     = ZAPHighUp
```

```
ZAPLowDown     = fire_Low_Down.exit
ZAPLowUp       = fire_Low_Up.exit
ZAPHighDown    = fire_High_Down.exit
ZAPHighUp      = fire_High_Up.exit

ERROR54 = err_54.exit
ERROR55 = err_55.exit

-- definition of the test processes
UNSAFETEST = TREATMENTLowDown |[{fire_High_Down} union delta_events]| OVERDOSE

OVERDOSE = fire_High_Down.testok.exit

macro
    fire_events    = {fire_Low_Down, fire_Low_Up, fire_High_Down, fire_High_Up}
    therac_events  = {lb, hb, ls, hs, xr, el, err}
    delta_events   = {d_Low_Down, d_Low_Up, d_High_Down, d_High_Up}

need sort computation
end
```

D.6 The Readers and Writers Example

```
-- Readers and Writers Problem, due to De Nicola, Inverardi and Nesi.
-- Gate parameters implemented by relabelling.

-- alphabet
-- rb, re denote reader begin, reader end
-- wb, we denote writer begin, writer end
-- p, v denote semaphore signals

conjecture
    Spec = Impl
where

-- specification
Spec = i.rb.re.Spec [] i.wb.we.Spec

-- implementation
Impl = hide {p,v} on S |[{p, v}]| (Proc [rb/b] [re/e] ||| Proc [wb/b] [we/e])
S    = p.v.S
Proc = p.b.e.v.Proc
end
```

D.7 The Candy Machine Example

```
-- Candy Machine Problem, due to De Nicola, Inverardi and Nesi.
-- Gate parameters modelled by relabelling.

-- alphabet
-- in10p denotes input ten pence
-- in25p denotes input twenty-five pence
-- out10p denotes output 10p
-- out25p denotes output 25p
-- t10p denotes through 10p
-- t25p denotes through 25p
-- candy denotes output candy
-- message denotes a piece of paper saying "try again"

conjecture
    Spec = Candy
where

-- specification
Spec = in10p.(i.message.Spec [] i.out10p.Spec [] i.candy.Spec)

-- Candy machine
Candy = hide {t25p, t10p} on
        (((Slot [t10p / out10p] [t25p / out25p]) |[{t25p, t10p}]|
        (Fair [t10p / in10p] [t25p / in25p])) |[candy_events]| Turn)

Slot = in10p.((i.message.Slot) [] (i.out10p.Slot) [] (i.out25p.Slot))
Fair = (in25p.candy.Fair) [] (in10p.out10p.Fair)

-- Turn is an addition from the original.  It allows a more natural
-- specification.
Turn = in10p.(out10p.Turn [] candy.Turn [] message.Turn)

macro
    candy_events = {in10p, out10p, message, candy}
end
```

D.8 The Scheduler Example

```
-- Scheduler Problem, due to Milner, and De Nicola, Inverardi and Nesi.
-- Translated from CCS.
-- The implementation is altered to give two versions of C1,
-- one which can start, and one which has to wait for the g1 action.
-- The original had a firing process which started C1, but this
-- implementation is inappropriate for LOTOS because of the
-- multi way synchronisation.

-- alphabet
-- a1, a2 denote the ``start'' actions of C1, C2
-- b1, b2 denote the ``stop'' actions of C1, C2
-- g1, g2 synchronise the implementation C1, C2

conjecture
    Spec = hide {b1, b2} on Sch
where

-- specification
Spec = a1.a2.Spec
```

```
-- implementation
Sch  = hide {g1, g2} on (C1 |[{g1, g2}]| C2)

C1   = a1.((b1.g2.C1') [] (g2.b1.C1'))
C1'  = g1.a1.((b1.g2.C1') [] (g2.b1.C1'))

C2   = g2.a2.((b2.g1.C2) [] (g1.b2.C2))
end
```