TOOLS TO SUPPORT A FORMAL VERIFICATION METHOD FOR SYSTEMS WITH CONCURRENCY AND NONDETERMINISM

By
Ramesh Bharadwaj

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Doctor of Philosophy

McMaster University

Doctor of Philosophy (Jan 1996) McMaster University
(Electrical Engineering) Hamilton, Ontario

TITLE: Tools to Support a Formal Verification Method for Systems with Concurrency and Nondeterminism

AUTHOR: Ramesh Bharadwaj

SUPERVISOR: Professor David L. Parnas

NUMBER OF PAGES: x, 140

Abstract

With the availability of inexpensive computer hardware, software intensive systems are becoming sophisticated and pervasive, creating a need for software design methods that deliver robust and efficient systems. Unfortunately, most systems today are designed by trial and error, with designers having little insight into their correctness before they are implemented. The hallmark of an engineering method is the use of system models (prototypes) to verify designs by *provably* establishing essential characteristics of products *before they are constructed*. This thesis attempts to bring the craft of software construction closer to an engineering discipline, by transforming theoretical ideas in program verification and concurrency theory into a concrete method for the analysis of software requirements and system specifications. In our opinion, such methods will find wider acceptance if they are adequately supported by a set of tools. This thesis also describes tools that are being developed to support our verification method.

In this thesis, our primary concern is the problem of establishing that systems, when constructed in accordance with a design, will meet their required correctness properties. We describe a verification method, TOP, in which the description of a system (i.e., its design) is expressed as an abstract program in a programming language-like notation called MeLa. We represent essential characteristics of the system (or its *requirements*) as predicates in a formal logic. The problem of establishing required correctness properties for the system is then reduced to the problem of establishing that a set of logical formulae hold. This is known as the *verification problem*. Our verification method, TOP, is mainly applicable to systems with complex control structures and simple data structures, and is tailored to systems that are designed not to terminate, and which often exhibit nondeterministic behaviour i.e., different runs of the system may produce different results, for the same inputs. Examples of such systems are operating systems, computer hardware, communications protocols, telecommunications systems, and control systems.

Our method offers a unified framework in which correctness properties may be verified using a combination of model checking (a fully automatic method), and theorem proving (a partially automated approach). This is desirable, because automated methods are applicable only to a limited class of systems. They are severely limited by the size of the state space generated by the abstract program denoting the system description — current limits are in the region of 10^8 states — roughly the number of states that could be generated by a program with a *single* 32-bit integer variable. Theorem proving approaches, while more general, are tedious and require specialised knowledge (the ability to provide proofs) on the part of verifiers. When carried out manually, proofs also tend to be error-prone. Theorem proving is particularly valuable in cases where the effort invested in carrying out a verification may be amortised over several projects (good examples are communications protocol specifications and distributed algorithms). Theorem proving can benefit tremendously from appropriate mechanical support, which can automate the tedious parts, in addition to playing the part of a relentless skeptic who demands the utmost precision and rigour in proofs. By presenting verifiers with a unified framework, therefore, we hope to extend the range of verification to systems that cannot be verified by either method alone.

One of the common complaints about theorem proving systems is that they are bewildering to beginners and hard to use. In this thesis we present an improved user interface for theorem provers. We also describe the details of system SNAP which has been built with this interface. SNAP has been designed for the purpose of checking proofs, in addition to providing partial automation of proofs. SNAP additionally allows users to carry out proofs at the desired level of abstraction. To support this, SNAP includes proof libraries and simplification algorithms based on conditional term rewriting.

Our method, TOP, has been used to verify problems derived from practice. We present two case studies that involved the use of TOP. In the first study, we provide a formal semantics for a tabular notation for requirements documents in terms of MeLa. We then use model checking and theorem proving to establish certain "safety assertions" for the requirements specification of a system that mediates access to data shared by two processes. In the second study, we analyse liveness violations in a communications protocol standard, and verify that suggested changes to the standard have indeed fixed the problem. Finally, we conclude by describing ongoing work and suggestions for future research.

Acknowledgements

I thank my supervisor Prof. David Parnas for his encouragement and support. Dr. Gerard Holzmann provided additional impetus and motivation for this work. Comments by Prof. Jeffery Zucker and Dr. Paul Clements proved valuable in making the exposition clearer. Prof. Terence Todd provided references for the DQDB protocol. Thanks are also due to Dr. Dennis Ritchie and AT&T Bell Laboratories for partially funding my thesis work. Dr. Frank Stomp collaborated with me on the design and implementation of the theorem prover SNAP. I also thank the Telecommunications Research Institute of Ontario for support.

Contents

Abstract iii

Acknowledgements vi

1 Introduction 1

1.1 Motivation 1

1.2 Our Approach 2

1.3 A Motivating Example 4

1.3.1 The Mutual Exclusion Problem 4

1.4 MeLa 5

1.4.1 Why another language? 6

1.4.2 The MeLa Notation 7

1.4.3 Semantics of MeLa Programs 9

1.5 Formal Verification 11

1.6 Snap 12

1.6.1 About the System 12

2 MeLa User Manual **13**

2.1 Notation and Vocabulary 13

2.2 Data 15

2.2.1 The type boolean 15

2.2.2 The type integer 16

2.2.3 Expressions 16

2.3 Declarations and Definitions 17

2.3.1 Variable Declarations 17
2.3.2 Channel Declarations 18
2.3.3 Event Declarations 19
2.3.4 Function Declarations 19
2.3.5 `adt` Declarations 20
2.3.6 Process Class Definition 20
2.4 Statements 21
2.4.1 Single-step Statements 21
2.4.2 Primitive Statements 22
2.4.3 Compound Statements 23
2.4.4 The Parallel Composition Statement 26
2.5 A MeLa Example 26

3 Formal Semantics for MeLa 32
3.1 Semantic Model 32
3.1.1 Transition System: 32
3.1.2 The Transition Relation 33
3.1.3 Definitions 34
3.1.4 Behaviours 34
3.2 Semantics of MeLa Programs 34
3.2.1 State Variables 35
3.2.2 Process Classes 35
3.2.3 Initial Condition 36
3.2.4 The Transition Relation 37
3.2.5 Compound Statements 39
3.3 An Example 42
3.3.1 State Variables 42
3.3.2 Initial Condition 42
3.3.3 Transition Relation 42

4 TOP — A Verification Method 44
4.1 Introduction 44

4.1.1 Why Model Checking? 44
4.1.2 Background 46
4.2 Notation 48
4.2.1 Notation for Programs 48
4.2.2 Notation for Requirements 49
4.2.3 Notation for Annotated Programs 51
4.3 The Model Checking Algorithm 52
4.4 Adapting LTL Rules to Model Checking 54
4.4.1 Rules for Invariance 54
4.4.2 Rules for Eventuality 59
4.5 Conclusion and Discussion 63

5 Computer Assistance for Proofs **64**
5.1 Formal Theories 64
5.1.1 Logical Consequence 65
5.1.2 Proofs and Validity 65
5.1.3 Hilbert-style Axiom Systems 66
5.2 Natural Deduction 67
5.2.1 Computers Assistance for Proofs 68
5.2.2 A Survey of Computer Assisted Proof Systems 69
5.2.3 The User Interface 72
5.2.4 Some Examples 73

6 Snap — A Proof Validator **78**
6.1 Why another system? 78
6.2 The Design Objectives of Snap 80
6.3 The User Interface 81
6.3.1 Conventional Wisdom 81
6.3.2 The Snap Interface 82
6.4 The Snap System 83
6.5 Snap's *Simplify* 87
6.6 Application to Dekker's Algorithm 88

7 SNAP Internals **91**

7.1 The Proof Annotation Algorithm 91

7.2 How Simplify Works 96

7.2.1 Rewriting Systems 96

8 Case Studies **98**

8.1 Event Tables 98

8.1.1 A MELA Description of the System 99

8.1.2 A MELA description of the Environment 100

8.2 The DQDB Protocol 105

8.2.1 Preliminaries 107

8.2.2 The DQDB Protocol in MELA 108

8.2.3 Bandwidth Balancing 110

9 Conclusion and Future Work **113**

A Proof of Dekker's Algorithm **116**

Bibliography **140**

Chapter 1

Introduction

1.1 Motivation

Computer networks, and distributed applications that run over them, are becoming sophisticated and pervasive. In order to seamlessly interconnect computer systems over multiple networks, perhaps using equipment from different vendors, and to be able to run (distributed) applications over them reliably and efficiently, we need computer and communication products that are robust and efficient. Unfortunately, current techniques for distributed systems design are woefully inadequate to meet this need. Most systems today are designed by trial and error, with designers having little insight in their correctness before they are implemented. As Gerard Holzmann [Hol93] argues, a good engineering discipline should allow designers to use engineering models (prototypes) to verify designs, by establishing essential characteristics of products *before* they are built.

To address this problem, we advocate the use of *Formal Engineering Methods* for the design and development of distributed systems. For example, consider communications protocols, the fundamental building blocks of distributed systems. Protocols are notorious for hidden and subtle errors, which often stem from *nondeterministic behaviour* caused by parallelism — parallelism allows several possible executions, resulting in different outputs, for the same inputs. It is difficult to design a reliable and robust communications protocol. Protocol design has remained a black art. In addition, protocol implementors find it difficult to interpret protocol standards because most standards documents are imprecise and incomplete. There is clearly a need for introducing engineering methods and tools in their design.

1.2 Our Approach

In this thesis, we investigate a method, supported by tools, to verify system designs. We use the term *system* in a broad sense; it may be an algorithm, embedded system, distributed application, communications protocol, or even computer hardware. We shall, however, restrict ourselves to a system's *logical* properties such as absence of deadlocks, or error-free data transfer, and not address performance issues such as a protocol's desired throughput or its delay characteristics. We do this by abstracting away from real time. This approach has the advantage of being able to establish properties of systems that are independent of timing constraints. The disadvantage is our inability to express (and verify) certain time dependent properties. We shall also not concern ourselves with methods to derive an implementation from a design, or prove that an implementation conforms to a design. In our opinion, the tools and methods proposed in this thesis are only applicable to systems with complex control structures and simple data structures, such as communications protocols or embedded control systems, and not for data-intensive applications such as systems for information retrieval or data base management.

We foresee practitioners applying our method and tools in one of two ways:

1. To validate a system's requirements specification. This is done by establishing that a given set of user-specified logical properties hold for the specification.

2. To model and analyse certain critical components of a design. We can do this by showing that a design, expressed in terms of an abstract program, has the required safety properties.

The problem of establishing logical properties of a system design is known as the "correctness problem". To start with, the design has to be expressed in an unambiguous notation, with a well-defined formal semantics. We propose a programming language-like notation, called MeLa, for this purpose. Why not use a popular programming language such as C for this? The answer is simple — the semantics of languages such as C are not formally defined. In addition, such languages lack concurrency constructs, and include features that make static reasoning about them very difficult. The question may still be asked as to why a language such as UNITY [CM88] was not used for this purpose. Our answer is that such languages only have academic appeal, and are too "low level" to be used in practice. For example, the UNITY notation abstracts away from control flow; therefore, to model a sequential process, one has to explicitly deal with the program counter, a process that reduces readability and is error prone. Such languages would require considerable enhancement before they can be used by practitioners. As we shall see in chapter 3, the language MeLa we introduce in this thesis may be viewed as a specification for a preprocessor that translates a higher-level language to a transition system notation such as UNITY.

We expect the result of a design process to be a *formal system description* expressed in MeLa. Further, we need a notation for expressing logical properties. In our method, logical properties, or *requirements criteria*, are written as formulae in a formal logic. We therefore reduce the correctness problem to that of establishing that all the requirements criteria hold for the system description. To analyse a formal system description and show that it satisfies all its requirements, we propose a *verification method*, TOP, which is supported by a set of tools that provide automatic as well as semi-automatic (or user guided) verification procedures.

Systems such as computer protocols and distributed algorithms exhibit *concurrent behaviour*, which stems from their inherent parallelism. Our notation for system descriptions, MeLa, therefore includes constructs for *concurrent programming*, such as *processes*, communication over *channels*, and *parallel composition*. Concurrent programming [BA82], and methods to reason about concurrent programs [CM88, MP83, MP91a, MP91b], have been an active area of research. The work reported here draws extensively from the body of literature that has emerged from research in concurrent program verification. Our method and tools also usefully combine two approaches in program verification — model-checking [CES86, Hol91, Kur89, McM93, MS91] and theorem-proving [Dow93, Fel93, GM93, GMW79, McC90] — by capitalising on the strengths of each approach, in order to compensate for the weaknesses of the other (for related work, see for example [KL93, RSS95, MN95]).

1.3 A Motivating Example

Concurrent programming is the name given to programming notations that allow parallelism, and techniques to solve the resulting synchronisation and communication problems. We assume that a *concurrent program* consists of several sequential "processes", whose executions are interleaved. The processes are not independent — they communicate with each other, to "synchronise" or to exchange data. The following problem, drawn from early concurrent programming literature, is meant to introduce the techniques and tools we develop in this thesis.

1.3.1 The Mutual Exclusion Problem

The mutual exclusion problem is an *abstraction* of a class of synchronisation problems encountered in concurrent programming. Consider two processes P_1 and P_2, which perform some set of actions A_1 and A_2 respectively. A_1 and A_2 are said to *exclude each other* if and only if executions of A_1 do not overlap (interleave) executions of A_2. If processes P_1 and P_2 both attempt to execute their respective actions, we must ensure that only one of them succeeds. The losing process must not proceed until the winning process completes execution of its actions. Dijkstra [Dij68] cast this problem as the *critical section problem*. Assume that each process executes the following code:

```
begin:
  remainder;
  pre-protocol;
  critical section;
  post-protocol;
  goto begin
```

Here, "*remainder*" is assumed to represent some processing. After the completion of *remainder*, each process enters a short *critical section*. It executes certain instructions, called the *pre and post protocol*, before and after the critical section. The problem is to devise a protocol which will ensure that each process will in fact execute its critical section, and that two processes will not execute their critical sections at the same time.

Several solutions to this problem are possible [Dij68]. The one without special language features, known as *Dekker's Algorithm*, is documented in the literature [BA82]. Here's a solution (in an Algol-60 like notation) that does ensure mutual exclusion, but fails to satisfy other requirements criteria of the critical section problem (for details, see [Dij68]).

```
Initially, x = 1

start:
  if x = 1 then x := 0;
  critsect:
  x := 1;
  goto start
```

The label *critsect* is a placeholder for the code that is executed in the critical section. For this program, we may state the critical-section requirement as follows: for any reachable state of the program, both processes must not be at the label *critsect*.

1.4 MeLa

In this thesis, we develop a programming notation, MeLa, using which we can express designs such as the solution for the problem above, and formally reason about them. MeLa has constructs for declaring variables and processes, programming constructs such as assignment, conditional and goto statements, and constructs for creating several processes, each executing similar code. We also provide a formal semantics for MeLa using the idea of "single-step relations", to serve as a specification for a language processor to transform MeLa programs into equivalent "transition systems".

A formal system description written in MeLa may then be verified using techniques we shall discuss in this thesis.

1.4.1 Why another language?

As we mentioned before, we need a precise and unambiguous notation for expressing system descriptions, in order to formally reason about them. The notation should include concurrent programming constructs, should have a well-defined and simple semantics, and yet allow descriptions of diverse systems including distributed algorithms, communications protocols, and asynchronous hardware, to be expressed in it. We did not choose to use concurrent programming languages for the reasons we mentioned before — they lack a formal semantics; proof systems to statically reason about them tend to be very complex and cumbersome to use.

Of the alternatives we considered, the protocol description language PROMELA [Hol91] seemed to meet our needs. However, PROMELA also has deficiencies: its design does not facilitate static reasoning (it has been designed exclusively for model checking); it lacks a formal semantic definition (its semantics is defined solely in terms of an interpreter, SPIN). We list below the problem areas of PROMELA:

1. Aliasing of identifiers (channel names) are allowed, which makes (static) reasoning about programs (needlessly) difficult.

2. The language permits system descriptions that are not finite state — an unbounded (implementation dependent) number of process instantiations are expressible in the language.

3. Some of the constructs (notably the `atomic` construct and the `run` command) have semantics which are extremely hard to formalise and reason about.

4. The language does not cleanly separate a system description from its correctness properties. This poses a version control problem during practical verifications, where several correctness properties have to be established for a system description.

We therefore decided to work with a subset of the language, in an attempt to work around these problems. However, we realised that certain fundamental modifications had to be made to the underlying model, in order to make theorem-proving methods easier to work with [Bha94]. These changes affected the basic semantics of the language. We therefore decided to give our language a new name, MeLa. Once we made this decision, MeLa quickly diverged from PROMELA, due to hard-to-resist changes to its concrete syntax, as well as several fundamental semantic ones:

- We did away with the **atomic** construct, and included a general way to group sets of primitive actions into atomic actions, using the comma "," statement separator.
- We introduced a parallel composition construct `par` instead of the `run` command of PROMELA.
- We made synchronisation channels to be typeless, and introduced typing in the form of *event* declarations. This smoothed out a common source of confusion in PROMELA, when messages of different types are sent over the same channel.

We may characterise MeLa as a language which was inspired by PROMELA, but with a new syntax and new semantics. It should, however, be easy to implement a language processor that translates MeLa programs to PROMELA (but not the other way around).

1.4.2 The MeLa Notation

The following treatment is meant to be an informal introduction to the MeLa language. To some readers, this section may raise more questions than it answers! We refer such readers to chapter 2 for a formal treatment of the language.

We list the following characteristics of MeLa:

1. Variable declarations are mandatory.
2. Certain keywords such as `init` and `par` are reserved and may not be used as identifiers.

3. The standard data types are those of whole numbers (`int`) and the logical values (`bool`). These may be augmented by (user defined) abstract data types (`adt`).

4. The basic program unit is a `process`, which denotes a single thread of control. The `init` process is similar to the function *main* of *C* [KR88], in that it serves as the entry point for the program.

5. Processes are non-recursive and are instances of process classes. A class of processes is defined in a `process class` definition, which may optionally include arguments with call-by-name (i.e., textual substitution) semantics.

6. Processes are created (and executed) by the parallel composition statement (`par`). A process is created by specifying its process class identifier, followed by the process identifier.

7. Process classes are built up from *single-step statements*, which are atomically executed. Each single-step statement is built up from *primitive statements* separated by commas ("`,`"). The semantics of a single-step statement is characterised by an executability criterion, a boolean expression, and a set of assignments. Operationally, its execution is a multiple assignment, conditional upon the boolean expression being *true*.

8. Primitive statements include the assignment statement, interprocess communication statements as in [Hoa78], and the unconditional branch (`goto`) statement.

9. Arrays of process instances may be created, and array variables of standard data types may be declared. Arrays may be of arbitrary dimensions, with constant bounds.

10. There are constructs for sequencing (";"), conditional execution (`choice`), iteration (`loop`), and branching (`goto`). *Labels* are identifiers, and are not (explicitly) declared.

11. User-defined functions may be declared either globally or within a `process class` declaration, and called within a process.

Let us cast the problem outlined in section 1.3.1 in MeLa.

```
int x = 1;
process crit()
{
start: (x = 1), x := 0;
critsect: x := 1, goto start
}
init
{ par
  { : crit cs[2] }
}
```

1.4.3 Semantics of MeLa Programs

In order to be able to formally reason about a MeLa program, it is imperative that we give it a well defined *meaning*, or semantics. We denote the semantics of a MeLa program as a set of sequences of states. The first state of each sequence is called its *initial state*. We view the program's execution as a series of *steps*, each step leading from a state to the next state, in a sequence. We denote terminating computations by finite sequences, and non-terminating computations by infinite sequences. A set of sequences may be defined in terms of a predicate, or *initialisation condition*, and a relation, known as the *single-step transition relation*. The initialisation condition characterises the set of initial states for the program. For a given state, the single-step transition relation specifies the possible next states. We represent the transition relation as a predicate on unprimed (denoting previous state), and primed (denoting the next state) versions of program variables.

The MeLa program above has *three* variables: the integer variable x, and two additional variables denoting the program counters of the two processes instantiated by the `par` statement. We denote these program counter variables by identifiers `cs[0]` and `cs[1]`. We may specify the semantics of the above MeLa program as follows:

Initialisation condition

$$((x = 1) \wedge (cs[0] = 0) \wedge (cs[1] = 0))$$

Transition Relation

$$\begin{array}{ll} & ((x=1)\wedge(cs[0]=0)\wedge(x'=0)\wedge(cs[0]'=1)\wedge(cs[1]'=cs[1]) \\ \vee & (cs[0]=1)\wedge(x'=1)\wedge(cs[0]'=0)\wedge(cs[1]'=cs[1]) \\ \vee & (x=1)\wedge(cs[1]=0)\wedge(x'=0)\wedge(cs[1]'=1)\wedge(cs[0]'=cs[0]) \\ \vee & (cs[1]=1)\wedge(x'=1)\wedge(cs[1]'=0)\wedge(cs[0]'=cs[0])) \end{array}$$

As we can see, the representation above for the transition relation is not very readable. Notice that it is written as a disjunction of smaller predicates. Each disjunct characterises a function, known as the *transition function*. We choose an alternate representation for the transition relation — we write it as a set of *transitions*, each one denoting a transition function. We call this representation the *transition system semantics*. Each transition consists of a boolean expression b, the symbol "$\Leftrightarrow\!\!\rightarrow$", followed by a finite sequence of assignments (separated by commas). If the boolean expression b of a transition evaluates to "true" in a state s, the transition is said to be *enabled* in that state. An enabled transition may be executed in a state by executing its assignments as a single multiple assignment, producing the next state. For the example program above, we may denote its transition relation as follows:

$$\begin{array}{rcl} (cs[0]=0)\wedge(x=1) & \Leftrightarrow\!\!\rightarrow & x:=0, cs[0]:=1; \\ (cs[0]=1) & \Leftrightarrow\!\!\rightarrow & x:=1, cs[0]:=0; \\ (cs[1]=0)\wedge(x=1) & \Leftrightarrow\!\!\rightarrow & x:=0, cs[1]:=1; \\ (cs[1]=1) & \Leftrightarrow\!\!\rightarrow & x:=1, cs[1]:=0; \end{array}$$

Given this semantics, we may now express the mutual exclusion requirement for the MeLa program as the logical formula:

$$\Box(\neg((cs[0]=1)\wedge(cs[1]=1)))$$

The formula above asserts that in every reachable state of the program, the program counters of processes $cs[0]$ and $cs[1]$ may never both have the value 1 (i.e., be at the label *critsect*). Note that the (meta-logical) operator *henceforth* (denoted by "$\Box$") implicitly quantifies the predicate $\neg((cs[0]=1)\wedge(cs[1]=1))$ over all program states.

1.5 Formal Verification

Verification is the process of establishing that a program satisfies its requirements. We may do this either by explicitly computing the set of sequences that denotes the program's semantics (also called its *model*), or by *theorem proving* i.e., using reasoning based solely on the program's transition system semantics. The former method, known as *model checking*, has the advantage that it can be performed automatically, by checking properties for their *validity* in a model. This method is only practical for models of reasonable size (current limits are in the region of 10^8 states [Hol91]). The main advantage of the model checking approach is its completeness — if a model checking algorithm reports that a theorem is false, it is false[1]. The main disadvantage of this approach is that it suffers from the *state explosion problem* — by their very nature, the number of reachable states of concurrent programs could be very large in relation to their textual representation. Many abstract programs also have *infinite* models, making model checking inapplicable for these problems.

Theorem proving methods, on the other hand, establish properties by constructing *proofs*. Proof construction, however, is a tedious and error-prone activity. Partial automation of proofs therefore seems to be a reasonable goal to pursue. We shall explore this further in later chapters. In addition, it would be beneficial to augment proof techniques with model-checking algorithms, permitting the two methods to be used together; e.g., proof rules may be used to decompose a large problem into smaller sub-problems, each of which may be automatically verified by model checking. In this thesis, we discuss several ways of doing this. To aid proof construction, systems should include proof management functions: for instance, they should not allow the user to declare a proof complete if there are goals or lemmata still to be proved. We shall take a look at the design of theorem prover SNAP [BS94], to see how these issues may be addressed.

[1]If a theorem prover fails to prove that same theorem, then all one can conclude is that the theorem prover failed to find a proof.

1.6 Snap

In this thesis, we develop a new human-computer interface for theorem proving systems. We also present a system (Snap) which has been implemented with this interface. Our system was designed for efficiency and ease-of-use. Using Snap, proofs may be carried out at a desired level of abstraction. The system interface permits machine assisted proofs to be carried out in a style that is close to "natural" proofs. In addition, Snap allows users to increase the level of abstraction of proofs (i.e., skip steps without compromising correctness), by letting them add new theorems, inference rules, and certain meta rules to the system's rule-base, even when another proof is in progress. Users from specific application areas may develop and maintain libraries of theorems, lemmata, and inference rules, which may be used by others, including novices, without detailed knowledge of their exact form, or their associated names — an intuitive understanding is sufficient to use them.

1.6.1 About the System

We have prototype implementations of system Snap. Our initial implementation was on UNIX; we have also ported the system to MS-DOS. Written in the *C* programming language [KR88] in about three months, Snap uses fewer than 3000 source statements. Our software design method was based on object-based module construction [GHM87, Par84]. We identified modules using the principle of *information hiding* [Par72]. This approach considerably reduced code duplication. Contrary to most reported experience, it also cut down development time. We hypothesise that this may be due to three reasons:

1. We had ready access to an expert in the field.
2. Our time estimate is for coding alone — it does not include effort expended for producing documentation.
3. We had been well-trained in the use of information hiding; in most projects, developers learn as they go.

Chapter 2

MeLa **User Manual**

2.1 Notation and Vocabulary

The basic *vocabulary* of MeLa consists of letters, digits, special symbols, and reserved words. The *special symbols* of the language are:

`+`	`-`	`*`	`%`	`:=`
`.`	`,`	`;`	`:`	`=`
`==`	`!=`	`#`	`{`	`}`
`(`	`)`	`[`	`]`	`&`
`/\`	`|`	`\/`	`~`	`!`
`?`	`<`	`<=`	`>`	`>=`
`=>`	`<=>`			

Reserved Words are interpreted as single symbols, *word symbols*, and may not be used as identifiers. The following words are reserved:

adt	bool	chan	choice	class	event
exit	ff	function	goto	init	int
loop	par	process	tt		

Identifiers are names denoting types, variables, process classes, and functions. They must begin with a letter, which may be followed by any combination and number

We shall specify syntax in the traditional *Backus-Naur Form (bnf)*, where syntactic constructs are denoted by identifiers which are suggestive of the meaning of the construct. Enclosure of a sequence of constructs by the following meta-brackets has the corresponding meaning:

- `{` and `}` implies a selection from a set of choices;
- `[` and `]` with a "`*`" suffix implies their repetition zero or more times;
- `[` and `]` with a "`+`" suffix implies their repetition one or more times;
- `[` and `]` (without a suffix) implies their repetition zero times or once.

Meta-symbol "`|`" stands for choice, symbol "`-`" denotes a range, symbol "`::=`" introduces a production, and terminal symbols are enclosed in quotes "`"`".

Figure 1: The *bnf* notation

of letters, digits, and the underscore ("_") character.

In the bnf notation (Figure 2.1), this may be stated as follows:

```
ident      ::= {"A"-"Z", "a"-"z"}["A"-"Z", "a"-"z", "0"-"9", "_"]*
```

Constants: Decimal notation is used for *numbers*, which may be either signed or unsigned.

Separators: Blanks, tabs, newlines, and comments serve as separators. An arbitrary number of separators may occur between any two consecutive symbols. No separators may occur within identifiers, numbers, special symbols or word symbols. At least one separator must occur between any pair of consecutive identifiers, numbers, or word symbols.

Comments are enclosed between character combinations `/*` and `*/`; comments have the same lexical significance as separators. Comments may not be nested.

2.2 Data

MeLa has two built-in data types — the type `bool` for booleans, and the type `int` for integers. In addition, users may define *abstract data types*, each comprising a type identifier, and related function declarations involving the adt.

2.2.1 The type boolean

A value of type boolean, introduced by the symbol `bool`, may be one of the logical truth values false and true, denoted respectively as `ff` and `tt`.

The following logical operators yield a value of type `bool` when applied to operands of type `bool`:

Operator	Symbol
and	`/\` *or* `&`
or	`\/` *or* `|`
not	`~` *or* `!`
implies	`=>`
iff	`<=>`

Each of the following relational operators yields a value of type `bool`.

Relational Operator	Symbol
equal	`=` *or* `==`
not equal	`#` *or* `!=`
less than	`<`
less than or equal	`<=`
greater than	`>`
greater than or equal	`>=`

2.2.2 The type integer

A value of type integer, introduced by the symbol `int`, is an element of the whole numbers. The following arithmetic operators yield an integer value when applied to integer operands:

Arithmetic Operator	Symbol
sum	`+`
difference	`-`
negation	`-`
product	`*`
modulus	`%`

2.2.3 Expressions

Expressions consist of constant or variable operands, operators, and function activations. An expression is a rule for calculating a value, with syntax and rules for *operator precedence* defined as below.

Terms

Integer valued expressions such as `x + 1` are called *terms*. In the following, symbol `o` stands for an infix binary operator that generically represents functions symbols $+$, $\Leftrightarrow$, $*$, $\%$. We define a *term* as follows:

1. All constants and variables are terms.
2. If $\mathcal{T}$ is a term, then `(-` $\mathcal{T}$ `)` is also a term.
3. If $\mathcal{T}$ and $\mathcal{U}$ are terms, then `(` $\mathcal{T}$ `o` $\mathcal{U}$ `)` is also a term.
4. An expression is a term if and only if it arises from the application of the three rules above.

To avoid the use of too many parentheses, we impose the following ordering on functions: negation, modulus, product, difference, sum, and allow elimination of parentheses according to the rule that negation applies to the smallest expression following it, then modulus applies to the smallest expression surrounding it, and so on. For example, we may write `a + b * c` to mean `a + ( b * c)`.

Logical Expressions

Boolean valued expressions are called *logical expressions*. In the following, we use the symbol `r` to generically represent relational operators. Based on the definition of *term* above, we define logical expressions as follows:

1. If $\mathcal{T}$ and $\mathcal{U}$ are terms, then `(` $\mathcal{T}$ `r` $\mathcal{U}$ `)` is a logical expression.
2. If $\mathcal{A}$ and $\mathcal{B}$ are logical expressions, then `(` $\mathcal{A}$ `/\` $\mathcal{B}$ `)`, `(` $\mathcal{A}$ `\/` $\mathcal{B}$ `)`, `( ~` $\mathcal{A}$ `)`, `(` $\mathcal{A}$ `=>` $\mathcal{B}$ `)`, `(` $\mathcal{A}$ `<=>` $\mathcal{B}$ `)`, are also logical expressions.
3. A logical expression may arise only from the application of the two rules above.

We impose the following ordering on logical operators: not, and, or, implies, if and only if, to allow parentheses elimination in the usual way.

2.3 Declarations and Definitions

A MeLa program may have global declarations, or declarations local to a *process class* definition or the `init` process.

2.3.1 Variable Declarations

Every variable occurring in a statement must be declared either globally or local to a `process class` definition. A variable declaration associates an identifier, a data type, and initial value(s) with a new variable. Initial values may be specified either explicitly as an *initialiser*, or implicitly by specifying an *initialising predicate* which is required to hold in all initial program states. The general form of a variable declaration is:

```
var_decl   ::= type ident_defn ["," ident_defn]* ";"
type       ::= ident | "int" | "bool" | "event"
ident_defn ::= qual_ident ["=" expn | "|" expn] ";"
qual_ident ::= ident | ident "[" number "]" ";"
```

Example:

```
int x = 0, y = 0;
bool done = ff;
int x, y | (x > 0) & (y > 0);
```

The first two declarations specify initialisers; the third declaration specifies an initialising predicate. The first declaration associates identifiers `x` and `y` with two variables of type `int`, each with initial values `0`. The second line declares `done`, a variable of type `bool` with initial value false (`ff`). The third declaration is similar to the first, but for the initial values — here, the variables associated with identifiers `x` and `y` can have any natural number as their initial values.

2.3.2 Channel Declarations

Channels are used to describe *synchronous* transfer of data from one process instance to another. A channel declaration associates an identifier with the datatype `chan`. The actual form of a channel declaration is:

```
channel_decl ::= "chan" ident_defn ["," ident_defn]* ";"
ident_defn   ::= ident | ident "[" number "]" ";"
```

Example:

```
chan to_rcvr, from_sndr;
chan chan_array[5];
```

The first declaration associates identifiers `to_rcvr` and `from_sndr` with two variables of type `chan`. The second one declares `chan_array` as an array of channels (of dimension 5, with indices $0 \ldots 4$).

2.3.3 Event Declarations

Channels are used for the synchronous transfer of either unstructured data or structured data, known as *events*, across process instances. Event declarations are introduced by the symbol `event`. The syntax of an event declaration is as follows:

```
event_decl ::= "event" [ ident_decl ]+ ";"
ident_decl ::= ident "(" [ event_args ] ")"
event_args ::= arg_type ["," arg_type]*
arg_type   ::= ident | "int" | "bool"
```

Example:

```
   event command(bool, int), response(int);
   event sabm(int, int);
```

The first declaration associates identifiers `command` and `response` as events, the former with two data fields of type `bool` and `int`, and the latter with one data field of type `int`. These events may be used for synchronous communication across channels, or assigned to variables of type `event`.

2.3.4 Function Declarations

Functions compute a single value (of a given type) for use in the evaluation of expressions. A function declaration consists of the result type, the function identifier, and zero or more formal arguments specified as a list of (type) identifiers. The syntax of function declarations is:

```
function_decl  ::= "function" [type] ident "(" [formal_types] ")" ";"
formal_types   ::= type ["," type]*
type           ::= ident | "int" | "bool"
```

Example:

```
  function int succ(int);
  function bool odd(int);
```

The activation of a function, known as a *function designator*, consists of the function identifier and a list of actual arguments. The number of arguments and their types must agree with the formal arguments; the function result type must be consistent with the type of the expression in which the function designator occurs.

Example: The above two functions may be activated as follows:

```
  succ(5);
  odd(2+4);
```

2.3.5 `adt` Declarations

A data type in MeLa may be one of the predefined types — `bool` or `int` — or a user-defined type, referenced by a *type identifier*. The symbol `adt` introduces the program part containing an adt declaration.

```
adt_decl  ::= "adt" ident ";"
```

Example:

```
  adt stack;
```

The above declaration introduces identifier "`stack`" as an adt. Subsequent declarations may use it as a (user-defined) data type. A user defined data type in MeLa is axiomatised by providing an equational theory for the data type.

2.3.6 Process Class Definition

A process class definition, introduced by symbols `process class`, defines a program part and associates it with an identifier. Instances of the process class may then be created using the *parallel composition* statement (`par`). A process class definition consists of a *heading*, followed by the *body*. The heading gives the process class a name, and lists its *arguments*. Arguments provide a substitution mechanism, allowing a process class to be instantiated with variations of its arguments. The body of a process class definition may contain additional *local declarations*, followed by one or more *statements*. Identifiers of local declarations have significance only within the process class text. A process class has a fixed number of arguments, each of which may be referenced within the body by identifiers called *formal arguments*. The following is the syntax of process class definitions:

```
proc_class_defn ::= "process class" ident "(" [formal_args] ")"
                    "{" body "}"
formal_args     ::= formal_decl [";" formal_decl]
formal_decl     ::= ("chan" | type) id_defn ["," id_defn]*
body            ::= [declaration]* stmt_list
stmt_list       ::= stmt [ ";" stmt]*
```

Example:

```
process class gcd(int x, y)
{
  loop {
  : x > y, x := x - y
  : y > x, y := y - x
  : x = y, exit
  }
}
```

2.4 Statements

Statements in MeLa are either *single-step* or *compound*. A single-step statement is executed atomically, and is made up of *primitive statements*, separated by commas. There are **six** kinds of primitive statements — the *condition* statement, the *assignment* statement, the *listen command*, the *shout command*, the unconditional branch (`goto`), and `exit`. There are **four** kinds of compound statements — the *sequencing* statement, the *choice* statement, the *loop* statement, and the parallel composition statement `par`.

2.4.1 Single-step Statements

A single-step statement is made up of component primitive statements. A single-step statement is executed only if all its component statements are executable. Executability of a primitive statement is determined as follows:

- for a *condition statement*, it must evaluate to `tt`;
- assignment statements and unconditional branch statements impose no additional executability conditions;
- for a shout (listen) command, the matching listen command(s) (shout command) must be executable, in another process instance.

Label Declarations Any single-step statement may be marked by prefixing it with a *label*, an identifier, followed by a colon (making possible a reference by a `goto` statement).

2.4.2 Primitive Statements

The Condition

A condition is an expression of type `bool`. It controls the execution of the single-step statement in which it occurs — the single-step statement may be executed only if all its component conditions evaluate to `tt`. Conditions are boolean expressions.

Example:

```
  (x > y)
   y > x
 x+1 = y
```

The Assignment

An assignment specifies that a newly computed value be assigned to a variable. An assignment imposes no additional condition on the executability of the single-step statement in which it occurs. The form of an assignment is:

```
assignment     ::= var_ref ":=" expression
var_ref        ::= ident | ident "[" expression "]"
```

Interprocess Communication

The *listen* and *shout* commands perform synchronous transfer of data (or events) between sending and receiving processes. A shout command, matching listen command(s), and all other primitive statements in the single-step statements in which they occur, are all executed in a single step (conditional upon the executability of all other primitive statements in the corresponding single-step statements). The syntax is as follows:

```
listen_command ::= var_ref "?" [var_ref | event_ref]
shout_command  :== var_ref "!" [expression | event_ref]
event_ref      ::= ident "(" [var_ref]* ")"
var_ref        ::= ident | ident "[" expression "]"
```

Example:

```
  to_rcvr!x                     to_rcvr?y
  from_sndr!command(tt, y)    from_sndr?command(done, x)
```

The statements in the first line have the effect of sending the current value of the variable `x` over the channel `to_rcvr`, and assigning it to the variable `y`. It is important to note that *the two statements must necessarily occur in two different process instances*. The statements in the second line have the effect of sending an event `command`, with additional values `tt` and the current value of the variable `y`, to the receiving process where the values are respectively assigned to variables `done` and `x`.

The Unconditional Branch

An unconditional branch consists of the symbol `goto` followed by an identifier, the label. It transfers control, *within a process*, to the single-step statement that appears immediately following the *label*.

2.4.3 Compound Statements

Compound statements are built up from single-step statements as below:

```
statement ::= [label] (par | choice | loop | single_step)
label     ::= ident ":"
par       ::= "par" "{" [":" ident id_defn "(" [var_ref]* ")"]+ "}"
choice    ::= "choice" "{" [":" stmt_list]+ "}"
loop      ::= "loop" "{" [":" stmt_list]+ "}"
id_defn   ::= ident | ident "[" number "]" ";"
var_ref   ::= ident | ident "[" expression "]"
```

The Sequencing Statement

The sequencing statement specifies that its component statements be executed in the same sequence as they are written. Individual statements are separated by the semicolon (sequencing) operator. Note that executions of component statements in a sequence may be interleaved by executions of statements from other process instances when control is at each ";" (but never at ",").

The Choice Statement

The choice statement consists of a (non-empty) set of compound statements. When control flow reaches the choice statement, it is transfered to one of the compound statements whose first single-step statement is executable. After all the components of the compound statement are executed, control is transferred to the statement immediately following the choice statement.

Example:

```
choice
{ : x >= y, x := x - y
  : y >= x, y := y - x
  : x = y
};
out!x
```

When control flow reaches the above choice statement, it is transferred to an executable compound statement (i.e., to one of the three compound statements). For example, if `x` $>$ `y`, control is transferred to the first compound statement, after the execution of which it is transferred to the shout statement that immediately follows the choice statement. If `x = y`, control is nondeterministically transferred to one of the three compound statements (because all of them are executable), after the execution of which it is transferred to the shout statement that immediately following the choice statement. Note that statements from other processes may interleave the execution of the choice statement and the shout statement.

The Loop Statement

The loop statement consists of a (non-empty) set of compound statements. When control flow reaches the loop statement, it is transfered to one of the compound statements whose first single-step statement is executable. After all the components of the compound statement are executed, control is transferred back to the loop statement. However, if one of the components of the compound statement has the `exit` statement, control is transferred to the statement immediately following the loop statement.

Example:

```
loop
{ : x > y, x := x - y
  : y > x, y := y - x
  : x = y, exit, out!x
}
```

When control flow reaches the above `loop` statement, control is transferred to one of the three compound statements, provided it is executable. For the cases where `x > y` or `y > x`, control is resumed at the loop statement after the corresponding compound statement has been executed. For the case where `x = y`, control is transferred to the third compound statement, and proceeds to the statement immediately following the loop statement after its execution, because of the occurring `exit` statement. Note that the third compound statement is executable only if `x = y` *and* if there is an executable *listen* statement on channel `out` in another process instance.

2.4.4 The Parallel Composition Statement

The parallel composition statement, `par`, activates specific instances of processes defined in `process class` definitions. An activation of a process class must contain a list of *actual arguments*, which are substituted (textually) for the corresponding formal arguments defined in the process class definition.

Example: The statement

```
par {
: gcd gcd_instance(p, q)
: get_gcd get_gcd_instance()
}
```

activates `gcd_instance`, an instance of process class `gcd`, and `get_gcd_instance`, an instance of process class `get_gcd`. The arguments `p` and `q` textually replace corresponding formal arguments of process class `gcd` (i.e., `x` and `y`, c.f. section 2.3.6).

2.5 A MeLa Example

We shall look at the alternating bit protocol [BSW69], and see how it can be described in MeLa. Figure 2.5 shows the system architecture of the protocol. There are two communicating entities, `Sender` and `Receiver`, connected via `Medium`, a two-way transmission channel which may occasionally lose messages. There is a one way transfer of data from `Sender` to `Receiver`, and transfer of acknowledgements in the other direction. Each message (data and acknowledgement) carries a bit, called the alternation bit. Each message sent has to be acknowledged by the receiver — on receipt of a message, `Receiver` sends a verification message to `Sender`, with the bit set to the corresponding bit of the message received. At initialisation, both `Sender` and `Receiver` agree on an initial setting of the alternating bit. Let us describe this protocol in MeLa.

To begin with, we may describe messages that are exchanged in the protocol as events:

```
event data0(), data1(), ack0(), ack1();
```

Here, `0` and `1` denote the settings of the alternating bit; `data` stands for data sent from `Sender` to `Receiver`, and `ack` for an acknowledgement from `Receiver` to `Sender`.

We now describe the behaviour of the medium as a process class:

```
process class Medium(chan from, chan to)
{ event buf[N];
  int in = 0, out = 0;

  loop {
  : from?buf[in], in := in+1
  : from?buf[in] /* message loss */
  : in > out, to!buf[out], out := out+1
  }
}
```

Process class `Medium` has two arguments, one for an incoming message channel and one for an outgoing channel (we will have *two* instances of this process class in our protocol, each of which describes the behaviour of the channel in one direction). The process body contains a `loop` statement, with three choices: The first choice may be taken only if another process has been enabled to send a message on channel `from`; making this choice has the effect of receiving the message, storing it in the variable `buf[in]`, and incrementing the variable `in`. The second choice receives the message but does not increment `in` — this has the effect of discarding the message that was received — which models message loss. The last choice may be taken if `in > out`, (i.e., there is at least one received message in `buf` that has not been sent yet), and if there is another process instance which has an enabled statement to receive a message on channel `to`; making this choice has the effect of sending the message to the receiving process, and incrementing the variable `out`.

Let us now describe process class *Sender*. To start with, we define a function `produce_msg()` to describe the act of acquiring the message to be sent.

```
function produce_msg();
```

Next, we declare a variable `send_bit` to denote the setting of the alternating bit, with initial value `1`. We may now describe process class *Sender* in terms of a `loop` statement, with *six* constituent compound statements — the first three to handle the case where the alternating bit is `1` and the remaining three for the case where the alternating bit is `0`. Let us examine the first three compound statements:

```
send_bit = 1, to_rcvr!msg1() /* Denotes sending, timeouts, bit=1 */
send_bit = 1, from_rcvr?ack1(), send_bit := 0;  produce_msg()
send_bit = 1, from_rcvr?ack0()
```

The first statement sends the message `msg1()` over the channel `to_rcvr` provided the channel has been enabled to receive it; further, the executability of this single-step statement is conditional upon the value of variable `send_bit` being 1. The second compound statement receives the message `ack1()` over the channel `from_rcvr` and sets variable `send_bit` to 0, followed by the activation of function `produce_msg()`. Note that statements of other processes may interleave executions of the first single-step statement and the function activation. Finally, in case message `ack0()` is received over the channel `from_rcvr` when `send_bit` is 1, executing the third statement has the effect of discarding the received message.

The last three compound statements handle the case when `send_bit` is 0, and are similar to the ones above:

```
send_bit = 0, to_rcvr!msg0() /* Denotes sending, timeouts, bit=0 */
send_bit = 0, from_rcvr?ack0(), send_bit := 1;  produce_msg()
send_bit = 0, from_rcvr?ack1()
```

Let us now describe process class *Receiver*. To start with, we define a function `consume_msg()` to describe the act of delivering a received message.

```
function consume_msg();
```

Next, we declare a variable `receive_bit` to denote the setting of the alternating bit, with initial value `1`. We may describe process class *Receiver* in terms of a `loop` statement, with *four* constituent compound statements — the first two to handle the case where the alternating bit is `1` and the remaining two for the case where the alternating bit is `0`. Let us examine the first two compound statements:

```
rcv_bit = 1, from_sndr?msg1(), rcv_bit := 0, to_sndr!ack1(); consume_msg()
rcv_bit = 1, from_sndr?msg0(), to_sndr!ack0()
```

The first compound statement consists of two single-step statements: the first statement receives message `msg1()` over the channel `from_sndr`, sets the variable `rcv_bit` to 0, and acknowledges message receipt by sending message `ack1()` over channel `to_sndr` (conditional on the channel being enabled to receive it); further, the executability of this single-step statement is conditional upon the value of variable `rcv_bit` being 1. This statement is followed by an activation of function `consume_msg()`. The second statement receives message `msg0()` over channel `from_sndr`, conditional upon the value of variable `rcv_bit` being 1; since the expected message is `msg1()`, the receiver assumes that an acknowledgement was lost, and merely (re)acknowledges receipt of `msg0()` by sending message `ack0()` over channel `to_sndr`.

The last two compound statements handle the case when `rcv_bit` is 0, and are similar to the ones above:

```
rcv_bit = 0, from_sndr?msg0(), rcv_bit := 1, to_sndr!ack0(); consume_msg()
rcv_bit = 0, from_sndr?msg1(), to_sndr!ack1()
```

Finally, we describe process `init` as the parallel composition of instances of `Sender`, `Receiver`, and two instances of `Medium`, with appropriate arguments:

```
init {
  par {
  : Sender S()
  : Receiver R()
  : Medium M1(to_rcvr, from_sndr)
  : Medium M2(to_sndr, from_rcvr)
  }
}
```

This completes the MeLa description of the alternating bit protocol. We reproduce below the complete description of the protocol:

```
event msg0(), msg1(), ack0(), ack1();
chan to_rcvr, from_sndr, to_sndr, from_rcvr;

process class Medium(chan from, chan to)
{ event buf[N]; int in = 0, out = 0;
```

```
  loop {
  : from?buf[in], in := in+1
  : from?buf[in] /* message loss */
  : in > out, to!buf[out], out := out+1
  }
}

process class Sender()
{ int send_bit = 1; function produce_msg();

  produce_msg();
  loop {
  : send_bit = 1, to_rcvr!msg1() /* Denotes sending, timeouts, bit=1 */
  : send_bit = 1, from_rcvr?ack1(), send_bit := 0;  produce_msg()
  : send_bit = 1, from_rcvr?ack0()
  : send_bit = 0, to_rcvr!msg0() /* Denotes sending, timeouts, bit=0 */
  : send_bit = 0, from_rcvr?ack0(), send_bit := 1;  produce_msg()
  : send_bit = 0, from_rcvr?ack1()
  }
}

process class Receiver()
{ int rcv_bit = 1; function consume_msg();

  loop {
  : rcv_bit = 1, from_sndr?msg1(), rcv_bit := 0, to_sndr!ack1(); consume_msg()
  : rcv_bit = 1, from_sndr?msg0(), to_sndr!ack0()
  : rcv_bit = 0, from_sndr?msg0(), rcv_bit := 1, to_sndr!ack0(); consume_msg()
  : rcv_bit = 0, from_sndr?msg1(), to_sndr!ack1()
  }
}

init
{ par {
  : Sender S()
  : Receiver R()
  : Medium M1(to_rcvr, from_sndr)
  : Medium M2(to_sndr, from_rcvr)
  }
}
```

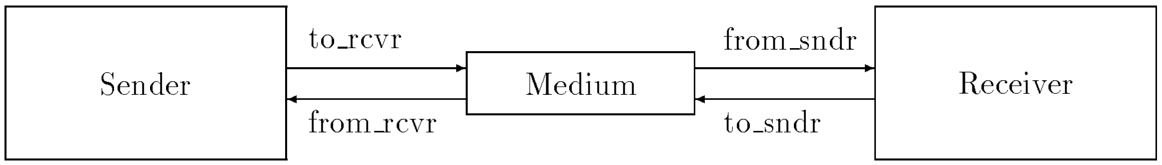

Figure 2: The Alternating Bit Protocol

Chapter 3

Formal Semantics for MeLa

In this chapter, we define a formal semantics for MeLa programs. We provide this in terms of a *transition system.* Our semantic definition may be viewed as the specification for a language processor which takes a MeLa program and translates it into its corresponding transition system semantics.

To start with, we provide a precise definition of a transition system. We then define a correspondence between constructs of MeLa and the transition system, by specifying how components of the transition system may be derived from the text of a MeLa program.

3.1 Semantic Model

Universe: We assume the existence of a non empty set $\mathbf{D}$, interpreted as the set of all states. We shall use symbol s (possibly subscripted) to denote a state i.e. $s \in \mathbf{D}$.

3.1.1 Transition System:

A transition system [MP91b, Plo81] is a triple

$$\{\Pi, \Theta, \mathcal{T}\}$$

where

- Π is a finite set of **state variables** $u_1, u_2, \ldots\ u_n$. Each state $s \in \mathbf{D}$ is an interpretation of Π, assigning to each variable $u_i \in \Pi$ a value over its domain. We denote such an assignment by $s[u_i]$. The state variables of Π are categorised as *data variables* and *control variables.*

- Θ is an *initial condition*, characterising the set of states in which execution of the system can begin.

- $\mathcal{T}$ is a (finite) set of transitions. Each transition $\tau \in \mathcal{T}$ represents a state-transforming action of the system, and is defined as a binary relation R_τ on $\mathcal{D} \times \mathcal{D}$ which relates a state s in $\mathcal{D}$ to a (possibly empty) set of states that may be obtained by executing transition τ in state s. Each state s' such that $sR_\tau s'$ is defined to be a τ-successor of s.

  Binary relation $R_\mathcal{T}$ denotes the transition relation of the entire transition system, i.e., $R_\mathcal{T} = \bigcup\limits_{\tau \in \mathcal{T}} R_\tau$

3.1.2 The Transition Relation

Associated with each transition τ is a relation R_τ which relates the value of the state variables in a state s to their values in a successor state s' obtained by executing transition τ in state s. We may express the semantics of transition τ, in terms of the relation R_τ, as a predicate on the set of unprimed and primed versions of variables in set Π. While more general notations are possible, we denote the transition relation of each transition τ by a boolean expression, or *guard*, the symbol "$\Leftrightarrow\!\rightarrow$", followed by a finite set of assignments (separated by commas), called the *body.* To obtain the predicate which characterises the relation R_τ of such a transition τ, we take the conjunction of the following:

1. the boolean expression (guard)

2. a set of predicates $(y'_i = e_i)$, (where each expression e_i refers only to unprimed state variables), for each assignment $y_i := e_i$ in the body, and

3. the set of predicates $(y'_j = y_j)$ for all variables $y_j \in \Pi$ that are not explicitly assigned values in the body.

A predicate that characterises the transition relation $R_{\mathcal{T}}$ of the transition system is obtained by the disjunction of the transition relation predicates of each transition $\tau \in \mathcal{T}$.

3.1.3 Definitions

For a transition τ in $\mathcal{T}$ and a state $s \in \mathcal{D}$, we say that:

- transition τ is *enabled* in s if there exists s' such that $sR_\tau s'$, i.e., s has a τ-successor;
- transition τ is *disabled* in s if there is no state s' such that $sR_\tau s'$, i.e., s has no τ-successor;

A state s is called a *terminal* state if there is no transition $\tau \in \mathcal{T}$ which is enabled in s.

3.1.4 Behaviours

A *behaviour* of a transition system is a *linear sequence* of states $s_0, s_1, \ldots$ such that the first state s_0 of the sequence is an initial state of the transition system (i.e., it satisfies the initial condition Θ), and for each consecutive pair of states $\langle s_i, s_{i+1} \rangle$ in the sequence, there exists a transition $\tau \in \mathcal{T}$ such that $s_i R_\tau s_{i+1}$. A behaviour either contains infinitely many states, or its last state is a terminal state. For our purpose, we assume all behaviours to be **infinite** sequences, without loss of generality; for, we may extend a finite sequence ending in state s_n into an infinite one by repeating s_n at the end of the sequence (infinitely many times).

3.2 Semantics of MeLa Programs

We shall now relate constructs of a MeLa program p with corresponding elements of the transition system t.

3.2.1 State Variables

Each globally declared variable of a MeLa program p corresponds to a (unique) data variable of the transition system t. Each instance of a process class in p corresponds to a control variable of t which represents the "program counter" associated with that process instance. For every instance of a process class in p, each locally declared variable of that process class corresponds to a (unique) data variable of t.

In addition, each data variable has an associated *type*, which defines the variable's domain (i.e, it defines an invariant condition on values that may be assigned to the variable in any state $s \in \mathcal{D}$). MeLa has two basic data types, and a facility for defining abstract data types. A variable declaration involving a basic data type maps an identifier to exactly one data variable, whose domain is specified by the basic data type. Collectively, the type specifiers of all data variables restrict the domain of each variable $u_i \in \Pi$, denoted by the set $dom(u_i)$, as follows:

$$\begin{aligned} \text{if } u_i \text{ is of type } \textbf{bool } dom(u_i) &= \{0, 1\} \\ \text{if } u_i \text{ is of type } \textbf{int } dom(u_i) &= \textbf{Int} \end{aligned}$$

A variable can also be declared as an *array* of one of the basic data types, of size n. The resulting declaration associates the identifier with n data variables of the corresponding transition system. This allows the use of subscripted identifiers of the form $id[e]$, where e is a term (c.f. chapter 2), wherever variable identifiers may appear.

$$\begin{aligned} &\text{if id corresponds to } u_k, u_{k+1}, \ldots\ u_{k+n-1} \\ &\quad id[e] \text{ maps to } u_{k+e} \text{ where } \ 0 \leq e < n \end{aligned}$$

3.2.2 Process Classes

Process classes define templates for instantiating processes. A process class definition comprises a process heading and a body. A process class heading introduces an identifier which denotes the process class, together with a list of arguments (formal parameters). Arguments serve as placeholders, providing a substitution mechanism which allows a process class to be instantiated with variations in its actual parameters.

Consider a process class definition, *pc*, with formal parameters $f_1, f_2, \ldots f_n$, and a body b. If a process of class *pc* is instantiated with actual parameters $a_1, a_2, \ldots a_n$, the text of its body is equivalent to $b' = b[a_1/f_1, a_2/f_2, \ldots a_n/f_n]$, i.e., the simultaneous substitution of a_i for f_i in b ($i = 1, \cdots, n$), provided that body b' constitutes a (syntactically and semantically) correct MeLa program.

The body of a process class definition consists of one or more MeLa *statements*. Each process class definition has an associated type, which specifies the domain of control variables of process instances of that class. Process instances are created by the `par` command, and have associated identifiers, or names. A process instance consists of a control variable, also known as its *program counter*, and zero or more data variables (which are instances of local variable declarations in the process class body). The control variable, identified by the process instance name, ranges over *label values* — elements of set $L = \{0, 1, \ldots\ m\}$. We assign unique label values to each MeLa statement in the body of a process. Label value 0, known as the *initial label value*, is the value assigned to all control variables in the initial state.

We also associate label identifiers (or *labels*) with certain label values; the function *maplabel*, with signature $labels \mapsto L$, denotes this association.

The `init` Process

The `init` process definition is a special case. It is equivalent to a process class definition (with no arguments), combined with an implicit process instantiation with name `init`.

3.2.3 Initial Condition

The initial condition, Θ, of the transition system corresponds to the **conjunction** of the following:

- initial values of all variables (if defined by initialisers),
- all initialising conditions (c.f. section 2.3.1), and

- a predicate $\pi_i = 0$, for each control variable π_i of process instances in the MeLa program.

3.2.4 The Transition Relation

We now define transition(s) associated with each kind of statement in a MeLa program p. For the purpose of this discussion, we assume that each statement is within the context of a process instance associated with a control variable π.

Expressions: Expressions in MeLa are either boolean expressions or terms as defined in chapter 2. A MeLa expression defines a mapping from interpretations of Π to the set $\{\texttt{tt}, \texttt{ff}\}$ (for boolean expressions), or the set of integers **Int** (for terms).

Primitive Statements

The Condition: The transition associated with a condition e is $\tau : e \Leftrightarrow\!\!\rightarrow skip$, where $skip$ is a null set of assignments.

Assignment: The transition associated with an assignment of the form u_i `:=` e is $\tau : \texttt{tt} \Leftrightarrow\!\!\rightarrow u_i := e$.

Interprocess Communication: Consider a listen command of the form $ch?x$, occurring in a process instance p_i, and a shout command of the form $ch!e$, occurring in a process instance p_j. The two commands are said to *match* one another if the following conditions are satisfied:

- the channel names ch are identical, one of the commands is a shout command and the other a listen command;
- process instance p_i is distinct from instance p_j, i.e., the commands occur in different process instances;
- if x denotes a variable identifier, and e an expression, the then x must be assignment assignment compatible with e, i.e., the declared type of variable x must be the same as the type of expression e;

- if x and e are events, they must have the same event name; if the event has arguments, then each (positionally) corresponding variable in the listen command must be assignment compatible (as defined above) with the expression in the shout command.

If two interprocess communication commands match, then they are deemed to occur in *both* process instances p_i and p_j.

The transition associated with two matching interprocess communication commands may be defined for the following two cases:

1. if x is a variable identifier and e an expression, then $\tau : \texttt{tt} \quad \Leftrightarrow\!\!\rightarrow x := e$;

2. if x and e are events, with arguments $x_1, x_2, \ldots x_m$ and $e_1, e_2, \ldots e_m$ respectively, then $\tau : \texttt{tt} \quad \Leftrightarrow\!\!\rightarrow x_1 := e_1, x_2 := e_2, \ldots x_m := e_m$.

Unconditional Branch: The transition associated with an unconditional branch statement of the form goto *label* is $\tau : \texttt{tt} \Leftrightarrow\!\!\rightarrow \pi := \mathit{maplabel}(\mathit{label})$, i.e., it assigns the label value associated with the identifier *label* to the control variable of the process instance in which it occurs.

The `exit` Statement: We associate transition `tt` $\Leftrightarrow\!\!\rightarrow$ `exit` with the `exit` primitive statement. Embedding an `exit` statement within a loop entails substitution of `exit` in the associated transition by an assignment to control variable π associated with the process instance in which the statement is embedded. Additionally, there is a proviso that the transitions associated with a process instance must have no `exit` statements in them (all of them should have been substituted out; i.e., `exit` statements are not allowed to occur outside the context of `loop` statements).

Single-step Statements

A single-step statement is made up of a set of primitive statements, and is of the form $p_1, p_2, \ldots p_s$, where p_i are primitive statements. Let τ_i, $(1 \leq i \leq s)$, be the transition associated with each primitive statement p_i in the single-step statement. The transition τ associated with a single-step statement is defined as follows: its guard is a predicate that is a conjunction of all the guards of transitions τ_i associated with each primitive statement p_i; its body is the union of all assignments of transitions τ_i associated with each primitive statement p_i.

3.2.5 Compound Statements

We associate a *set* of transitions with each compound statement of MeLa. Compound statements are built from single-step statements using the sequencing, conditional, iterative, and parallel composition constructs. Each compound statement has *two* associated label values: a *pre-label* value l_{pre}, and a *post-label* value l_{post}. In the following, we assume that the compound statement is embedded within the body of a process instance with control variable π.

One Single-step Statement

For a compound statement that consists of one single-step statement, we associate a single transition with the following components:

- Its guard is a conjunction of the guard of the transition associated with the single-step statement and predicate $\pi = l_{pre}$.
- Its body is the union of the body of the transition associated with the single-step statement, and the singleton set with element $\pi := l_{post}$.

Sequencing

The sequencing operator semicolon ";" composes two compound statements within a process instance. Given two compound statements c_1 and c_2, their sequential composition $c_1 ; c_2$ is defined as follows:

1. Let l_{mid} be an additional (unique) label value.
2. Associate label value l_{pre} with the pre-label of c_1 and value l_{mid} with the post-label of c_1.

3. Associate label value l_{mid} with the pre-label of c_2 and value l_{post} with the post-label of c_2.

4. The set of transitions associated with the sequential composition statement is the union of the transitions associated with statements c_1 and c_2.

Conditional Control

The choice statement is of the form

$$\begin{array}{l} choice\{ \\ : c_1 \\ : c_2 \\ \quad \ldots \\ : c_n \\ \} \end{array}$$

where $c_1, c_2, \ldots\ c_n$ are compound statements.

We define the transitions associated with the above conditional control statement as follows:

1. Associate label value l_{pre} with the pre-labels of statements $c_1, c_2, \ldots c_n$.

2. Associate label value l_{post} with the post-labels of statements $c_1, c_2, \ldots c_n$.

3. The set of transitions associated with the conditional control statement is the union of the transitions associated with statements $c_1, c_2, \ldots c_n$.

Iteration

The loop statement is of the form

$$\begin{array}{l} loop\{ \\ : c_1 \\ : c_2 \\ \quad \ldots \\ : c_n \\ \} \end{array}$$

where $c_1, c_2, \ldots\ c_n$ are compound statements.

We define the transitions associated with the above iteration statement as follows:

1. Associate label value l_{pre} with the pre-labels and post-labels of statements $c_1, c_2, \ldots c_n$.

2. Replace every `exit` primitive statement in the transitions associated with statements $c_1, c_2, \ldots c_n$ by the assignment $\pi := l_{post}$.

3. The set of transitions associated with the iteration statement is the union of all transitions obtained in the previous step.

Parallel Composition

The parallel composition statement is of the form

$$\begin{array}{l} par\{ \\ : t_1 \;\; p1 \\ : t_2 \;\; p2 \\ \quad \ldots \\ : t_n \;\; pn \\ \} \end{array}$$

where $t_1, t_2, \ldots\ t_n$ are process class identifiers and $p1, p2, \ldots pn$ are corresponding process instances. In section 3.2.2 we defined the transitions associated with process instances. Let $p1_{post}, p2_{post}, \ldots pn_{post}$ denote the post-labels of process instances $p1, p2, \ldots pn$ respectively, and $p_1, p_2, \ldots p_n$ denote their control variables. The set of transitions associated with the parallel composition statement is the union of the following:

1. transitions associated with process instances $p1, p2, \ldots pn$, with identical body but with an additional conjunct $\pi = l_{pre}$ in each guard;

2. transition $(p_1 = p1_{post}) \wedge (p_2 = p2_{post}) \wedge \ldots (p_n = pn_{post}) \Leftrightarrow\!\!\rightarrow \pi := l_{post},\ p_1 := 0,\ p_2 := 0, \ldots\ p_n := 0$.

3.3 An Example

Consider the MeLa program of chapter 1, reproduced below:

```
int x = 1;
process crit()
{
begin: (x = 1), x := 0;
critsect: x := 1, goto begin
}
init
{ par
  { : crit cs[2] }
}
```

Let us determine the transition system semantics for the above program.

3.3.1 State Variables

The program has one associated data variable (x), and three associated control variables $cs[0]$, $cs[1]$, and `init`, which denote the program counters of process instances `cs[0]`, `cs[1]`, and the `init` process respectively.

3.3.2 Initial Condition

The initialiser `x = 1` specifies the initial condition for data variable x. All control variables implicitly have the initial value 0. The predicate characterising the initial condition therefore is:

$$((x = 1) \wedge (cs[0] = 0) \wedge (cs[1] = 0) \wedge (\texttt{init} = 0))$$

3.3.3 Transition Relation

Process class `crit` comprises two single-step statements, composed by the sequential composition operator "`;`". We associate label value 0 with the label `begin`, and label value 1 with the label `critsect`. Finally, we associate label value 2 with the post-label of the second single-step statement (which does not have an explicit label). The transition associated with the first single-step statement of process instance `cs[i]`, for $i \in \{0,1\}$ is:

$$(cs[i] = 0) \wedge (x = 1) \Leftrightarrow\!\!\rightarrow x := 0, cs[i] := 1;$$

Similarly, we associate the following transition with the second single-step statement of process instance `cs[i]`, for $i \in \{0,1\}$:

$$(cs[i] = 1) \Leftrightarrow\!\!\rightarrow x := 1, cs[i] := 0;$$

Here, the assignment to $cs[i]$ is a result of the `goto` statement in the definition of the process class.

We now associate the following five transitions with the `par` statement (and thereby the entire program):

$$\begin{aligned}
(\texttt{init} = 0) \wedge (cs[0] = 0) \wedge (x = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 0, cs[0] := 1 \\
(\texttt{init} = 0) \wedge (cs[0] = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 1, cs[0] := 0 \\
(\texttt{init} = 0) \wedge (cs[1] = 0) \wedge (x = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 0, cs[1] := 1 \\
(\texttt{init} = 0) \wedge (cs[1] = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 1, cs[1] := 0 \\
(cs[0] = 2) \wedge (cs[1] = 2) &\quad \Leftrightarrow\!\!\rightarrow \quad \texttt{init} := 1, cs[0] := 0, cs[1] := 0
\end{aligned}$$

Simplification: We may optimise the above set of transitions, based on the following observations: Variables $cs[0]$, $cs[1]$, are never assigned the value 2 in the initial state, as well as in any transition. Therefore, the guard of the last transition above can never evaluate to `tt`; the transition may therefore be discarded without affecting the semantics. For the resulting program, predicate `init` $= 0$ remains invariantly true, and may be omitted in the guards of all transitions.

The simplified set of transitions, presented in chapter 1, is reproduced below:

$$\begin{aligned}
(cs[0] = 0) \wedge (x = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 0, cs[0] := 1 \\
(cs[0] = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 1, cs[0] := 0 \\
(cs[1] = 0) \wedge (x = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 0, cs[1] := 1 \\
(cs[1] = 1) &\quad \Leftrightarrow\!\!\rightarrow \quad x := 1, cs[1] := 0
\end{aligned}$$

Chapter 4

TOP — A Verification Method

4.1 Introduction

A common approach for establishing that a program meets its correctness requirements is by *theorem proving*. Recently, *model checking* [CES86, Hol91, Kur89, McM93] has emerged as an alternative for the verification of *finite-state* programs. Model checking is based on the idea of expressing a program's requirements as a formula in temporal logic, and viewing the program as a structure that may be interpreted as the formula's model [Wol89]. In this thesis, we shall focus our attention on correctness requirements expressed as formulae in Linear-time Temporal Logic (LTL) of [MP83, MP91b]. Other popular approaches use branching-time logics such as CTL (see for example [CES86, McM93] for details).

4.1.1 Why Model Checking?

Theorem proving methods for establishing correctness properties of programs tend to be tedious. Manually carried out proofs also tend to be error prone. For example, appendix A presents the proof of a safety property for Dekker's algorithm. The proof is about 700 lines long, and consists of 20 lemmata. In order to increase our degree of confidence in the proof, it was mechanically checked using the system we discuss in later chapters. It is therefore not surprising that the proof took more than *two person-weeks* to carry out. A barrier to the industrial use of theorem proving methods is that few practitioners have the mathematical sophistication or theorem proving skills required to use them.

Model checking, on the other hand, is performed automatically, i.e., "by the push of a button". The main advantage of the model checking approach is its completeness — if a model checker reports that a theorem is false, it is false. On the other hand, if a theorem prover fails to prove that same theorem, then all one can conclude is that the theorem prover failed to find a proof. However, model checking methods inherently suffer from a major limitation, namely, state explosion[1]. As we outline in [BFS95] model checking is most effective when used in combination with theorem proving methods.

In this chapter, we shall investigate a verification method, TOP, which combines the two approaches. In the traditional (theorem proving) approach, proof rules are used to decompose a more general problem into a set of sub-problems (under user guidance). This process may be repeated for each sub-problem. The validity of each sub-problem is then established individually be providing a proof in a formal axiomatic theory. When there are no remaining sub-problems, the original problem will have been verified. In our method, TOP, we present users with a second version of each proof rule. Such rules, called *model checking rules*, have been derived from their classical versions. For a particular sub-problem, therefore, users have the option of using either the classical proof rule or the corresponding model checking rule. In the latter case, the validity of *all* the premises are collectively established by an algorithm we present in this chapter. Termed *direct model checking*, this approach has several advantages over traditional model checking: reasoning is still done in terms of the original proof rules, users do not have to learn a new formalism, and establishing validity under fairness assumptions (a difficult problem for traditional model checkers) is handled just as in theorem proving approaches.

[1]Because they operate directly on the set of computations of a program, these algorithms must concretely denote all the states in the computation, which may be very large when compared to a program's textual representation. Therefore, even if a given verification problem is *decidable*, it may not be *tractable* in practice.

During verification, a question that usually arises is, "which version of the proof rule should I use?" In practice, we have found that the following thumb rule usually works: whenever possible, use the model checking rule; otherwise, (i.e., if there is no finite-state abstraction for a problem, or if the state space is too large), use traditional theorem proving. Note that the latter usually entails "discovering" stronger invariants; model checking rules help us "debug" these invariants, by providing counterexamples after a partial exploration of the state space (even if it is infinite).

The method TOP *integrates* model-checking with proof-theoretic approaches. Our goal is to capitalise on the strengths of each approach, in order to overcome the limitations of the other, thereby extending the range of problems that may be formally verified. By presenting users with a unifying framework for reasoning about their programs, our method seamlessly integrates the two verification approaches. This is a major improvement over existing methods [KL93, RSS95, MN95].

4.1.2 Background

The usual approach for model checking an LTL formula is to construct a finite-automaton on infinite words (a Büchi automaton) for the *negation* of the formula, take a synchronous product of the automaton representing the behaviour of the program and the Büchi automaton representing the negation of the formula, and check that the language accepted by the product automaton is empty [VW86, Wol89]. Emptiness is checked by determining whether the set of accepting states is reachable from the initial set of states, and belongs to a cycle [CVWY92, Hol91]. An advantage of this approach is that using the negation of the formula may result in a smaller state space to be explored. A popular way to implement this algorithm is to compute the program automaton and the synchronous product in the same step, obviating the need to store the whole state-space graph, an approach known as *on-the-fly (OTF)* model checking [CVWY92, Hol91].

A major disadvantage of this approach is that to model check a property expressed in LTL requires the construction of an equivalent Büchi automaton that has, in general, an exponential number of states in the length of the formula [Wol89]. Also, this approach is difficult to use in practice because users, being unfamiliar with the Büchi automaton construction procedure, do not have an intuitive understanding of the generated Büchi automaton — interpreting counterexamples produced by the model checker is therefore very difficult.

Another disadvantage of this method is that fairness assumptions [Fra86, MP91b] cannot be handled very well. A plausible approach to handle fairness is to express the fairness assumption as a formula $\mathcal{F}$, and model check the formula $\mathcal{F} \Rightarrow q$, where q is the property to be checked. This approach is not feasible in practice because even if the original formula q expressing the desired property is short, the formula $\mathcal{F}$ expressing the fairness assumption will have a length proportional to the size of the program — therefore, the formula to be checked will no longer be short. Other approaches that build fairness directly into the state exploration algorithms [Pel93] have the disadvantage that they generate inordinately long counterexamples, making them hard to interpret.

In this chapter, we explore a new approach for the verification of temporal properties. The direct model checking method adapts well-known proof rules used in theorem proving LTL formulae [MP91a], to model checking. Our method has several advantages:

- There is no need to construct (or use) Büchi automata for model checking properties expressed in LTL.
- It unifies model checking with theorem proving methods, vastly increasing the set of tools available to users. In fact, it permits a verification problem to be tackled using a combination of model checking and theorem proving tools.
- Fairness (both weak fairness as well as strong fairness) is handled in a straightforward way, just as in theorem proving approaches.
- The method is compatible with on-the-fly model checking and the bit-state hashing algorithm of [Hol88].

Our approach requires some intuition on the part of users, to be able to come up with auxiliary invariants (and other predicates) which may be necessary for model checking. In our opinion, this is actually an advantage because use of this approach is close to a genuine proof i.e., a chain of argument which will convince a human reviewer; the pitfalls of relying on the mere grunt of assent from an oracle (in this case, the model checking program) are well-known [RvH91].

4.2 Notation

In the following discussion, we associate a set V of *variables* with every program $\mathcal{P}$. Each variable may take on values over a (finite) domain $\mathbf{D}$. A *state* is an interpretation of V, assigning to each variable a value from domain $\mathbf{D}$.

4.2.1 Notation for Programs

Without loss of generality, we represent programs as transition systems, as in UNITY [CM88]. In chapter 3 we have seen how we may mechanically transform programs with constructs such as processes, channels, sequential and parallel composition, and repetition, to this notation.

Recall that a program $\mathcal{P}$ may be denoted as a pair $\langle \Theta, \mathcal{T} \rangle$, where Θ is the *initial predicate* which characterises the set of *initial states*, and $\mathcal{T}$ is a (finite) set of *transitions*. Each transition $\tau \in \mathcal{T}$ consists of a quantifier-free predicate, or *guard*, the symbol "$\Leftrightarrow\!\!\rightarrow$", followed by a finite set of assignments (separated by commas), called the *body*. Operationally, a transition is said to be *enabled* in a state s if its guard evaluates to "true" in that state. For a transition τ, we denote the set of enabled states by predicate $En(\tau)$. If a transition is enabled in state s, its body may be *executed* in that state. Assignments in the body are executed as a single multiple assignment. For example, the body "x:=y, y:=x" exchanges the values of variables x and y.

Also recall that we denote the *behaviour* of program $\mathcal{P}$ as a *linear sequence* of states $s_0, s_1, \ldots$ such that the first state s_0 of the sequence is an initial state of $\mathcal{P}$ (i.e., it satisfies the initial predicate Θ), and for each consecutive pair of states $\langle s_i, s_{i+1} \rangle$ in the sequence, there exists a transition τ of $\mathcal{P}$ such that its guard is *true* in state s_i, and state s_{i+1} results from executing τ's body in state s_i. We assume all behaviours to be **infinite** sequences, without loss of generality; for, we may extend a finite sequence ending in state s_n into an infinite one by repeating s_n infinitely often at the end of the sequence.

4.2.2 Notation for Requirements

This section describes the language for expressing correctness properties or requirements criteria for system descriptions expressed in MELA. We use Linear-time Temporal Logic (LTL) of [MP91b] as our requirements language.

We assume an underlying first-order assertion language $\mathcal{L}$, with interpreted symbols for expressing the standard operations and relations over domain $\mathbf{D}$. We shall refer to formulae in language $\mathcal{L}$ as *state formulae* or *assertions*. We shall denote the proposition "true" by **tt** and proposition "false" by **ff**.

We construct *temporal formulae* out of state formulae by applying the logical operators $\neg$ (negation) and $\vee$ (disjunction), and the temporal operator $\mathcal{U}$ (until). Other logical operators such as $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), and $\wedge$ (conjunction), and temporal operators such as $\Diamond$ (eventually) and $\Box$ (henceforth) can be defined in terms of these elementary operators.

A temporal formula p is interpreted over an infinite sequence of states $\sigma : s_0, s_1, \ldots$, known as a *structure*, where each state s_i is an interpretation for the variables in p. For a structure σ, the general notion of a temporal formula p *holding* at position $j \geq 0$ in σ is written as $(\sigma, j) \models p$. In the following, we provide an inductive definition of a temporal formula p holding for sequence σ at position $j = 0$[2]:

- If p is a state formula (i.e., has no temporal operators),

  $$(\sigma, 0) \models p \Leftrightarrow s_0 \models p$$

  we evaluate p using the interpretation given by state s_0.

- $(\sigma, 0) \models \neg p \Leftrightarrow (\sigma, 0) \not\models p$.

- $(\sigma, 0) \models p \vee q \Leftrightarrow (\sigma, 0) \models p$ or $(\sigma, 0) \models q$.

[2]This is known as a *canonical* interpretation of formula p.

- $(\sigma, 0) \models p \, \mathcal{U} \, q \Leftrightarrow$ for some $k \geq 0$, $(\sigma, k) \models q$ and for all i such that $0 \leq i < k$, $(\sigma, i) \models p$.

In addition, we define two other temporal operators:

- $\Diamond p = \mathbf{tt} \, \mathcal{U} \, p$ (eventually)
- $\Box p = \neg \Diamond \neg p$ (henceforth).

Program Validity of LTL Formulae

The behaviour of a program $\mathcal{P}$, an infinite sequence of states, can serve as a structure over which a temporal formula p may be interpreted.

We define the validity of a temporal formula p for a program $\mathcal{P}$ by interpreting p over all behaviours of program $\mathcal{P}$. if p is valid for all behaviours of program $\mathcal{P}$, we say program $\mathcal{P}$ *satisfies* the temporal property p. We may check this by using proof rules for establishing temporal properties of programs. In this chapter, we outline how these proof rules may be adapted for model checking, and propose an algorithm to accomplish this task.

We shall restrict ourselves to the following groups of properties[3]. These proof rules suffice to establish most temporal properties and are the main working tools during practical verifications [MP90].

One group of rules establishes the validity of the *invariance* formulae $\Box q$ and $\Box(p \Rightarrow \Box q)$; these formulae express the invariance of a state formula q, either throughout a behaviour, or starting from the state in a behaviour in which formula p holds.

Another group of rules establishes the validity of the *eventuality* formulae $\Diamond q$ and $\Box(p \Rightarrow \Diamond q)$; these formulae express the guarantee that q will eventually be true in a behaviour, either once, or following each state in which formula p holds.

These proof rules reduce the problem of establishing an LTL property for a program to the problem of establishing the validity of a (finite) set of state formulae. Some of these are directly expressed as (first-order) predicates. Others, known as

[3]The proof system may be extended to a richer system, which establishes validity of any LTL formula; for details see [MP91a, MP91b].

annotated transitions, are written in terms of individual transitions of the program. They are of the form $\{pre\}\tau\{post\}$, where *pre* and *post* are predicates and τ is a transition. The interpretation of $\{pre\}\tau\{post\}$ is that whenever the guard of τ is true in a state in which *pre* holds, then *post* holds in the state resulting from the execution of the body of τ.

We may translate each annotated transition into a first-order predicate. We do this by using the notion of *weakest precondition* [Dij76]. Let B denote the body $x_1 := e_1, \cdots, x_n := e_n$. The weakest precondition $wp(B, post)$, where *post* is a predicate, is defined as $post[e_1/x_1, \cdots, e_n/x_n]$, where $post[e_1/x_1, \cdots, e_n/x_n]$ denotes the predicate resulting from the simultaneous substitution of e_i for x_i in *post* ($i = 1, \cdots, n$). An annotated transition $\{pre\}\tau\{post\}$, where $\tau = g \Leftrightarrow\!\!\!\rightarrow B$, is equivalent to the predicate $(pre \wedge g) \Rightarrow wp(B, post)$.

4.2.3 Notation for Annotated Programs

An *annotated program* $\mathcal{P}^A$ is a pair $\langle \Theta^A, \mathcal{T}^A \rangle$. It consists of an *annotated initial predicate* Θ^A and a (finite) set of annotated transitions $\mathcal{T}^A$. $\Theta^A = \langle \Theta, Init \rangle$, where Θ and *Init* are predicates, is written as $\Theta\{Init\}$. Each annotated transition $\tau^A \in \mathcal{T}^A$ is a triple $\langle pre, \tau, post \rangle$ written as $\{pre\}\tau\{post\}$. Here, *pre* and *post* are predicates and τ is a transition of the form $g \Leftrightarrow\!\!\!\rightarrow B$ with guard g and body B. We define the enabledness of an annotated transition τ^A in terms of the enabledness of its component transition τ.

Operationally, we may define a behaviour of an annotated program just as we did for a program: a behaviour $\sigma = s_0, s_1, \ldots$ starts from a state s_0 satisfying predicate Θ of Θ^A, and for each pair of consecutive states s_i and s_{i+1} in σ, execution of the transition part of an enabled annotated transition in state s_i results in state s_{i+1}.

Additionally, each behaviour must satisfy the following conditions:

- Predicate *Init* must hold in state s_0. If not, the behaviour is said to have an *assertion violation* in state s_0. In this event, the singleton sequence s_0 is called a *counterexample*.

- If state s_{i+1} results from state s_i by executing the transition part of an enabled annotated transition τ^A, and the predicate part *pre* of τ^A holds in state s_i, then predicate part *post* of τ^A must hold in state s_{i+1}. If not, such a behaviour is said to have an *assertion violation* in state s_i. In such a case, and additionally if there are no assertion violations in states $s_0, \ldots, s_{i-1}$, the sequence of states $s_0, \ldots s_i$ is called a *counterexample*.

4.3 The Model Checking Algorithm

The model checking problem we consider is as follows: given an annotated program $\mathcal{P}^A$ check for assertion violations, if any, and provide a counterexample for the first encountered violation. We call this process *assertion checking*. The algorithm, adapted from [HS82], performs a depth-first search (DFS) of the behaviours of program $\mathcal{P}^A$. In the following, the notation *Pred*(s) where *Pred* is a predicate and s is a state is used to denote the evaluation of predicate *Pred* in state s (yielding a boolean result). We denote the cardinality of set $\mathcal{T}^A$ by $\mid \mathcal{T}^A \mid$. We use the notation τ_i^A to denote the $i^{\mathtt{th}}$ annotated transition of $\mathcal{T}^A$. The components of τ_i^A are respectively denoted by $\tau_i^A.pre$, τ_i, and $\tau_i^A.post$. Predicate *enabled*(τ_i^A, s) denotes the enabledness condition of transition τ_i^A in state s. Function *succ*(τ, s) returns the state resulting from the execution of transition τ in state s.

The main data structures used are the following: Stack S contains the sequence of states of a behaviour, starting from the initial state up to (and excluding) the *current state* (cs). Elements of stack S are pairs $\langle s, n\rangle$ where $s \in \mathbf{D}$ is a state and $n \in \mathbf{Nat}$ is a natural number denoting the first unexplored transition of state s. We denote the empty stack by `empty`; stack operations *push*, *pop*, and *top* have the usual meaning. Set H contains all states that have been visited during the DFS. Set H is usually implemented as a hash-table. Set operations $\cup$ and $\in$ have the usual meaning. We denote the null set by $\emptyset$.

```
$H := \emptyset$; $S :=$ empty
for each $s_0$ such that $\Theta(s_0)$ and $s_0 \notin H$ do
    $cs := s_0$; $assert(Init(s_0))$; $H := H \cup \{s_0\}$; $push(S, \langle s_0, 1\rangle)$;
    while $S \neq$ empty do
        $\langle cs, i\rangle := top(S)$; $pop(S)$
        while $i \leq |\ \mathcal{T}^A\ |$ do
            if $enabled(\tau_i^A, cs)$ then
                $ns := succ(\tau_i, cs)$
                if $\tau_i^A.pre(cs)$ then $assert(\tau_i^A.post(ns))$ fi
                if $ns \notin H$ then
                  $H := H \cup \{ns\}$; $push(S, \langle cs, i+1\rangle)$;
                  $cs := ns$; $i := 1$
                else $i := i + 1$ fi
            else $i := i + 1$ fi
        od
    od
od
```

The algorithm uses an auxiliary procedure *assert*. In this procedure, assume that routine $PrintStates(S)$ prints out the state component of elements of stack S and routine $PrintState(s)$ prints out state s.

```
procedure $assert(Predicate\ \ p)$
    if $\neg p$ then
      /* report assertion failure */
      $PrintStates(S)$; $PrintState(cs)$; exit
    fi
end
```

4.4 Adapting LTL Rules to Model Checking

In this section, we outline how we may interpret LTL proof rules in the context of model checking. We have adapted these proof rules from [MP83, MP90, MP91a, MP91b]. As we mentioned in section 4.2.2, we shall restrict ourselves to proof rules for invariance and eventuality formulae, as our intent is to convey to the reader the central idea. It would be easy to extend our work to the general case of model checking any LTL formula (on the lines of [MP91a, MP91b]).

Using a proof rule, one may infer an LTL property for a program from the validity of a set of formulae known as the *premises*. In this section, we present two versions of a number of proof rules. One version is meant for carrying out proofs using the classical approach, in which these validities are established individually by providing a proof in a formal axiomatic theory. The second version is meant for the direct model checking approach: in this approach, the validity of the premises is established either by translating them into model checking sub-problems (using auxiliary proof rules), or by collectively establishing validity of all premises by assertion checking an annotated program using the algorithm outlined in section 4.3.

In the following, we denote by $\{pre\}T\{post\}$, where pre and $post$ are predicates, and $T \subseteq \mathcal{T}$ is a set of transitions, the condition of requiring $\{pre\}\tau\{post\}$ to hold for every $\tau \in T$.

4.4.1 Rules for Invariance

Example Rule 1: Establishing $\Box q$

(a) Proof Rule for $\Box q$: For a program $\mathcal{P}$, a proof rule to establish the validity of temporal formula $\Box q$, where q is a state formula, is as below.

$$\boxed{\begin{array}{lll}\textbf{INV} & \text{I1.} & \Theta \Rightarrow \phi \\ & \text{I2.} & \phi \Rightarrow q \\ & \text{I3.} & \{\phi\}\mathcal{T}\{\phi\} \\ \hline & & \Box q\end{array}}$$

This rule uses an auxiliary predicate ϕ to establish the invariance of q. We can prove its soundness by showing that ϕ holds for all states s_j of a behaviour, by induction on j, using premises I1 and I3. From this and premise I2 the conclusion follows.

In order to see why an auxiliary predicate ϕ is required to establish invariance, let us examine the mutual exclusion problem we presented in chapter 1 (section 1.3.1). There, we cast the mutual exclusion requirement as the problem of establishing the LTL property: $\Box(\neg((cs[0] = 1) \wedge (cs[1] = 1)))$ for a program $\mathcal{P}$ for which:

$$\Theta = ((x = 1) \wedge (cs[0] = 0) \wedge (cs[1] = 0))$$

and the transitions of set $\mathcal{T}$ are:

$$\begin{array}{rcl}(cs[0] = 0) \wedge (x = 1) & \Leftrightarrow\!\!\!\rightarrow & x := 0, cs[0] := 1; \\ (cs[0] = 1) & \Leftrightarrow\!\!\!\rightarrow & x := 1, cs[0] := 0; \\ (cs[1] = 0) \wedge (x = 1) & \Leftrightarrow\!\!\!\rightarrow & x := 0, cs[1] := 1; \\ (cs[1] = 1) & \Leftrightarrow\!\!\!\rightarrow & x := 1, cs[1] := 0;\end{array}$$

Let us try to directly establish property $\Box q$, where $q = \neg((cs[0] = 1) \wedge (cs[1] = 1))$, without using a stronger auxiliary predicate i.e., let predicate $\phi = q$. We may conclude $\Box q$ (by rule **INV**) if we establish the validity of the following predicates:

$$\begin{array}{rl}I1: & ((x = 1) \wedge (cs[0] = 0) \wedge (cs[1] = 0)) \Rightarrow q \\ I2: & q \Rightarrow q \\ I3a: & \{q\}(cs[0] = 0) \wedge (x = 1) \Leftrightarrow\!\!\!\rightarrow x := 0, cs[0] := 1\{q\} \\ I3b: & \{q\}(cs[0] = 1) \Leftrightarrow\!\!\!\rightarrow x := 1, cs[0] := 0\{q\} \\ I3c: & \{q\}(cs[1] = 0) \wedge (x = 1) \Leftrightarrow\!\!\!\rightarrow x := 0, cs[1] := 1\{q\} \\ I3d: & \{q\}(cs[1] = 1) \Leftrightarrow\!\!\!\rightarrow x := 1, cs[1] := 0\{q\}\end{array}$$

Predicates $I1$ and $I2$ are valid. It is easy to see that predicates $I3b$ and $I3d$ are also valid. Predicates $I3a$ and $I3c$, however, are not valid (to prove them, one is required to prove the absurdity $1 \neq 1$). Therefore, it is clear that predicate q is not strong enough to be "pushed through" all transitions in set $\mathcal{T}$. We leave it as an exercise to the reader to verify that choosing the stronger predicate:

$$\begin{aligned}\phi \;=\; & ((x = 1) \wedge (cs[0] = 0) \wedge (cs[1] = 0)) \vee \\ & ((x = 0) \wedge (cs[0] = 1) \wedge (cs[1] = 0)) \vee \\ & ((x = 0) \wedge (cs[0] = 0) \wedge (cs[1] = 1))\end{aligned}$$

establishes the premises of rule **INV**.

Why is a stronger auxiliary predicate ϕ required for rule **INV** to be applied successfully? The reason is obvious — since we establish assertional validity individually for each transition in premise $I3$ of rule **INV**, even if predicate q holds in all reachable states of the program, it may not necessarily be propagated by a transition $\tau \in \mathcal{T}$ from an enabled state of τ to the next state. This is because a state in which transition τ is enabled is not necessarily a reachable state of program $\mathcal{P}$.

(b) Model Checking Rule for $\Box q$: For a program $\mathcal{P}$, the validity of temporal formula $\Box q$, where q is a state formula, is established by the following *model checking rule INV^**:

$$\boxed{\begin{array}{lll} INV^* & \Theta^A: & \Theta\{q\} \\ & \mathcal{T}^A: & \{q\}\mathcal{T}\{q\} \\ \cline{2-3} & \mathcal{P} \models & \Box q \end{array}}$$

This rule asserts that in order to establish, for program $\mathcal{P} = \langle \Theta, \mathcal{T} \rangle$, the invariance of a state property q (i.e., $\Box q$), it is sufficient to assertion check an annotated program $\mathcal{P}^A$ with annotated initial predicate $\Theta\{q\}$, and annotated transitions $\{q\}\mathcal{T}\{q\}$.

We prove its soundness by establishing the premises of proof rule **INV**, by setting $\phi = q$. From this follows the conclusion of **INV**, and hence the conclusion of INV^*.

Note that in this *model checking* approach, we *collectively* establish *all* the premises of rule **INV** essentially by evaluating predicate q in every reachable state of program $\mathcal{P}$. Because we evaluate predicate q only in the reachable states of program $\mathcal{P}$, an auxiliary (stronger) predicate ϕ is not required in this approach.

Example: Using the above model checking rule, we may establish the invariance property $\Box q$ for the mutual exclusion algorithm, by model checking the following annotated program:

$$\begin{array}{ll}
\Theta^A: & ((x=1) \wedge (cs[0]=0) \wedge (cs[1]=0))\{q\} \\
\mathcal{T}^A1: & \{q\}(cs[0]=0) \wedge (x=1) \Leftrightarrow\!\!\!\rightarrow x := 0, cs[0] := 1\{q\} \\
\mathcal{T}^A2: & \{q\}(cs[0]=1) \Leftrightarrow\!\!\!\rightarrow x := 1, cs[0] := 0\{q\} \\
\mathcal{T}^A3: & \{q\}(cs[1]=0) \wedge (x=1) \Leftrightarrow\!\!\!\rightarrow x := 0, cs[1] := 1\{q\} \\
\mathcal{T}^A4: & \{q\}(cs[1]=1) \Leftrightarrow\!\!\!\rightarrow x := 1, cs[1] := 0\{q\}
\end{array}$$

Example Rule 2: Establishing $\Box(p \Rightarrow \Box q)$

(a) Proof Rule for $\Box(p \Rightarrow \Box q)$: For a program $\mathcal{P}$, a proof rule to establish the validity of temporal formula $\Box(p \Rightarrow \Box q)$, where p and q are state formulae, is given below (**CINV** stands for "conditional invariance").

$$\boxed{\begin{array}{lll}
\textbf{CINV} & \text{C1.} & p \Rightarrow \phi \\
 & \text{C2.} & \phi \Rightarrow q \\
 & \text{C3.} & \{\phi\}\mathcal{T}\{\phi\} \\
\hline
 & & \Box(p \Rightarrow \Box q)
\end{array}}$$

As in rule **INV**, rule **CINV** requires an auxiliary predicate ϕ to establish the invariance of q, starting from a state s_i in which predicate p holds. Its soundness can be easily proved as before.

(b) Model Checking Rule for $\Box(p \Rightarrow \Box q)$: To model check a formula of the form $\Box(p \Rightarrow \Box q)$, where p and q are state formulae, we first formulate the following auxiliary proof rule:

$$\boxed{\begin{array}{lll} \textbf{CINVa} & \text{C1a.} & \Box(p \Rightarrow q) \\ & \text{C2a.} & \Box(q \Rightarrow \Box q) \\ \cline{2-3} & & \Box(p \Rightarrow \Box q) \end{array}}$$

Soundness of this rule follows from the rule of *entailment transitivity*: $\Box(p \Rightarrow q), \Box(q \Rightarrow r) \vdash \Box(p \Rightarrow r)$ (see [MP91b] page 230).

In practice, premise C1a is established either by theorem proving (e.g., by establishing the validity of $p \Rightarrow q$), or by showing that it is an invariant property of program $\mathcal{P}$ using the model checking rule INV^*.

Premise C2a, i.e., a property of the form $\Box(q \Rightarrow \Box q)$ is known as a *stability* property. This formula asserts that if q holds at any state in a behaviour, then it will continue to hold for the rest of the behaviour. We may establish this either by using proof rule **CINV** or by the following model checking rule $CINV^*$:

$$\boxed{\begin{array}{lll} CINV^* & \Theta^A: & \Theta\{\mathbf{tt}\} \\ & \mathcal{T}^A: & \{q\}\mathcal{T}\{q\} \\ \cline{2-3} & \mathcal{P} \models & \Box(q \Rightarrow \Box q) \end{array}}$$

This rule asserts that in order to establish that a state property q is stable for program $\mathcal{P} = \langle \Theta, \mathcal{T} \rangle$, it is sufficient to assertion check an annotated program $\mathcal{P}^A$ with annotated initial predicate $\Theta\{\mathbf{tt}\}$ (where $\mathbf{tt}$ stands for the predicate "true"), and annotated transitions $\{q\}\mathcal{T}\{q\}$.

To prove soundness: Assume that predicate q holds at state s_i of a behaviour of program $\mathcal{P}$. We can show that predicate q will continue to hold for all states s_j $(j \geq i)$ of the behaviour by induction on j.

4.4.2 Rules for Eventuality

Example Rule 3: Basic Response under Weak Fairness

(a) Proof Rule for $\Box(p \Rightarrow \Diamond q)$: This formula asserts that every state s_i where formula p holds is followed by a state s_j $(j \geq i)$ where formula q holds. Response properties are usually established under a *fairness assumption* [Fra86, MP91b]. A common fairness assumption, known as *weak fairness*, formalises the concept of "finite progress" [Dij68] — if a program $\mathcal{P}$ has an enabled operation, then it will be eventually executed. The effect of a fairness assumption is to disallow behaviours that would otherwise be legal behaviours of program $\mathcal{P}$. The following rule relies on weak fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

$$\boxed{\begin{array}{lll}
\textbf{RESP-W} & \text{R-W1.} & \Box(p \Rightarrow (q \vee \phi)) \\
 & \text{R-W2.} & \{\phi\}\mathcal{T}\{q \vee \phi\} \\
 & \text{R-W3.} & \{\phi\}\tau_k\{q\} \\
 & \text{R-W4.} & \Box(\phi \Rightarrow En(\tau_k)) \\
\cline{2-3}
 & & \Box(p \Rightarrow \Diamond q)
\end{array}}$$

We prove soundness by contradiction. Assume that p holds at state s_i, but q does not hold at states s_j $(j \geq i)$. Then, ϕ must hold at all states s_j $(j \geq i)$ (by premises R-W1, R-W2). Therefore, transition τ_k will never be taken (by premise R-W3). This means that transition τ_k is continuously enabled in these states but never taken (by premise R-W4), which violates our fairness assumption.

(b) Model Checking Rule for $\Box(p \Rightarrow \Diamond q)$: In order to adapt rule **RESP-W** to model checking, we first formulate the following auxiliary proof rule **RESP-Wa**:

$$\boxed{\begin{array}{lll} \textbf{RESP-Wa} & \text{R-W1a.} & \Box(p \Rightarrow \phi) \\ & \text{R-W2a.} & \Box(\phi \Rightarrow \Diamond q) \\ \cline{2-3} & & \Box(p \Rightarrow \Diamond q) \end{array}}$$

Soundness of this rule follows again from entailment transitivity.

To model check a basic response formula, we first apply proof rule **RESP-Wa**. Premise R-W1a can be established by using model checking rule INV^* or by using a combination of model checking and theorem proving methods. Premise R-W2a is established by the following model checking rule $RESP\Leftrightarrow W^*$:

$$\boxed{\begin{array}{lll} RESP\Leftrightarrow W^* & \Theta^A: & \Theta\{\mathbf{tt}\} \\ & \mathcal{T}_1^A: & \{\phi\}\mathcal{T}\{\phi \wedge En(\tau_k)\} \\ & \mathcal{T}_2^A: & \{\phi\}\tau_k\{q\} \\ \cline{2-3} & \mathcal{P} \models & \Box(\phi \Rightarrow \Diamond q) \end{array}}$$

To prove soundness, assume that ϕ holds at state s_i. Then, ϕ holds and τ_k is enabled at all states s_j $(j \geq i)$ (by premise $\mathcal{T}_1^A$). Therefore, by weak fairness, for some $j \geq i$, τ_k is taken at s_j and hence q holds at state s_{j+1} (by premise $\mathcal{T}_2^A$). This gives the conclusion.

Note that program $RESP\Leftrightarrow W^*$ contains annotated transitions $\{\phi\}\tau_k\{\phi \wedge En(\tau_k)\}$ **and** $\{\phi\}\tau_k\{q\}$ (since $\tau_k \in \mathcal{T}$). Our model checking algorithm has been designed in such a way that assertions of both transitions will be checked. An important point is that *behaviours that violate the weak fairness assumption will be generated during model checking.* However, these behaviours will not trigger assertion violations, as we only check for violations under the assumption of weak fairness.

Example Rule 4: Basic Response under Strong Fairness

(a) Proof Rule for $\Box(p \Rightarrow \Diamond q)$: Let us now examine how the basic response property may be established under the assumption of *strong fairness*[4]. The following rule relies on strong fairness to ensure that a helpful transition τ_k, leading to q, will be taken.

$$\boxed{\begin{array}{lll} \textbf{RESP-S} & \text{R-S1.} & \Box(p \Rightarrow (q \vee \phi)) \\ & \text{R-S2.} & \{\phi\}\mathcal{T}\{q \vee \phi\} \\ & \text{R-S3.} & \{\phi\}\tau_k\{q\} \\ & \text{R-S4.} & \Box(\phi \Rightarrow \Diamond(q \vee En(\tau_k))) \\ \cline{2-3} & & \Box(p \Rightarrow \Diamond q) \end{array}}$$

The difference between this rule and the previous rule is the fourth premise. Premise R-W4 requires that ϕ holding at a state s_i implies that the helpful transition τ_k is enabled in state s_i. Premise R-S4, however, only requires formula q to hold or transition τ_k to be enabled in a state s_j, $(j \geq i)$.

We prove soundness by contradiction. Assume that p holds at state s_i, and q does not hold at states s_j, $(j \geq i)$. Then ϕ holds continuously, with transition τ_k never being taken (by premises R-S1 – R-S3). Hence, transition τ_k is enabled infinitely many times (by premise R-S4), which violates our assumption of strong fairness.

(b) Model Checking Rule for $\Box(p \Rightarrow \Diamond q)$: The above rule **RESP-S** may be adapted to model checking on the lines of rule **RESP-W**. The fourth premise, an eventuality formula, is established either by theorem proving or by using model checking rule $RESP\Leftrightarrow W^*$.

[4]Intuitively, the requirement of strong fairness disallows computations in which a transition τ is enabled infinitely many times but taken only finitely many times.

Example Rule 5: Basic Response using Well-Founded Induction

Let us now examine how we may establish eventuality properties without relying on single helpful transitions. Known as *extended response properties*, they are established based on the principle of *well-founded induction*, by an argument similar to proofs of termination for sequential programs.

Let $\prec$ be a partial order on a set $\mathcal{A}$. The structure $(\mathcal{A}, \prec)$ is said to be *well-founded* if there is no infinite sequence $a_0, a_1, a_2, \ldots$ $(a_i \in \mathcal{A})$ such that for $i = 0, 1, 2, \ldots$, $a_{i+1} \prec a_i$. The following rule uses *well-founded induction* to establish eventuality properties. Here δ is a term and z is a *rigid variable*[5]. Typically $(\mathcal{A}, \prec)$ is the naturals with the standard ordering.

$$\boxed{\begin{array}{lll} \textbf{RESP-WF} & \text{R-WF1.} & \Box(p \Rightarrow (q \vee \phi)) \\ & \text{R-WF2.} & \Box((\phi \wedge (\delta = z)) \Rightarrow \Diamond(q \vee (\phi \wedge (\delta \prec z)))) \\ \cline{2-3} & & \Box(p \Rightarrow \Diamond q) \end{array}}$$

Soundness is proved as follows: If predicate p holds at state s_i, then either q or ϕ hold at state s_i (by premise R-WF1). If ϕ holds and $\delta = z$, then eventually we will reach a state in which either q holds, or ϕ is maintained but with δ smaller than z (by premise R-WF2). Since $(\mathcal{A}, \prec)$ is well-founded, the value of δ cannot keep decreasing indefinitely. Therefore, predicate q must hold eventually.

This principle can also be used in model checking, as follows. Premise R-WF1, an invariance property, can be established as outlined in section 4.4.1. Premise R-WF2, an eventuality formula, is usually established by one of the two previous rules.

[5]Rigid variables are those which are introduced in the course of a proof as logical variables, and not as translations of program variables; a rigid variable must have the same value in all states of the behaviour of a program [MP91b].

4.5 Conclusion and Discussion

We have presented TOP, a verification method for temporal logic, which unifies model checking and theorem proving. It allows portions of a theorem proving problem to be verified by model checking methods, thereby increasing the array of tools at the disposal of users.

A well known problem of model checking is that it does not supply an argument which will convince human reviewers — the fact that a model checking algorithm has detected no counterexamples to a temporal claim is not sufficient evidence to convince reviewers of its validity. The approach outlined in this chapter brings model checking closer to theorem proving, in that one may extract a chain of arguments as a result of a model checking effort, allowing a tighter integration of model checking algorithms with deductive rules of inference within a theorem prover. Our method also allows portions of a model checking problem to be verified by theorem proving methods. This increases the array of tools at the disposal of users.

Our model checking algorithm may be used in conjunction with the theorem proving system SNAP which is discussed in chapter 6. By carrying out model checking within the context of such a system, users will be able to employ the theorem prover's book-keeping capabilities to manage complex verifications. Users will also be able to perform reductions and transformations of the verification problems by employing the theorem proving system's automatic rewrite capability (see section 6.5 and 7.2 for details).

Chapter 5

Computer Assistance for Proofs

5.1 Formal Theories

In general, there is no algorithmic way to determine logical validity of formulae in predicate logic. To establish validity of such formulae, an approach based on the notion of a **formal theory** has to be used. A *formal theory*, or *theory* for short, consists of the following:

1. a set of formulae, designated as the *axioms* of the theory and
2. a finite set of relations $R_1, R_2, \ldots R_k$ between formulae, called the *rules of inference.*

For a given theory $\mathcal{K}$, one can usually decide algorithmically whether any formula is an axiom[1]. For any set of formulae , , and a formula $\mathcal{A}$, one can algorithmically check whether , is in a relation R_i to $\mathcal{A}$. If this is the case, $\mathcal{A}$ is called a *direct consequence* of , by virtue of R_i.

A *formal proof* is a sequence of formulae $A_1, A_2, \ldots A_n$ such that for each i, either A_i is an axiom or is a direct consequence of a subset of the preceding formulae by virtue of one of the rules of inference[2].

[1]In such a case, we call $\mathcal{K}$ an *axiomatic* theory.

[2]In general, there may be several rules of inference that meet this criterion; it is sufficient if we can identify one.

A formula $\mathcal{A}$ is a *theorem* if there is a proof such that the last formula is $\mathcal{A}$. Such a proof is called a *proof of* $\mathcal{A}$.

5.1.1 Logical Consequence

A formula $\mathcal{A}$ is said to be a *logical consequence* of a set , of formulae in a formal theory $\mathcal{K}$ if and only if there is a sequence of formulae $A_1, A_2, \ldots A_n$ such that $A_n = \mathcal{A}$ and, for each i, $(1 \leq i \leq n)$, either A_i is an axiom or A_i is in , , or A_i is a direct consequence of a subset of the preceding formulae in the sequence by virtue of a rule of inference of $\mathcal{K}$. This is written as , $\vdash_K \mathcal{A}$. If the context is clear, we abbreviate this as , $\vdash \mathcal{A}$. The formulae in , are called the *premises*, and formula $\mathcal{A}$ is called the *conclusion*.

If , is the empty set $\emptyset$, then $\emptyset \vdash \mathcal{A}$ if and only if $\mathcal{A}$ is a theorem. The notation $\emptyset \vdash \mathcal{A}$, or just $\vdash \mathcal{A}$, therefore asserts that $\mathcal{A}$ is a theorem.

5.1.2 Proofs and Validity

How does a formal theory help in establishing validity of logical formulae? To establish validity, we provide proofs in a formal theory for which a formula is a theorem if and only if it is logically valid. The most interesting example for our purpose is the *first-order predicate calculus*.

For a first-order predicate calculus $\mathcal{K}$, every theorem of $\mathcal{K}$ is logically valid. This is known as the soundness theorem of first-order predicate calculus. Any logically valid formula $\mathcal{A}$ is a theorem of a first-order predicate calculus $\mathcal{K}$. This is known as Gödel's completeness theorem for the first-order predicate calculus.

Even for axiomatic theories, the notion of "theorem" is not necessarily effective; in general, there is no algorithm to determine if a given formula $\mathcal{A}$ is a theorem. Theories for which such algorithms exist are called *decidable*; otherwise they are called *undecidable*. First-order predicate calculus is undecidable, in other words there is no decision procedure for predicate logic. However, we may computably (recursively) enumerate the theorems of predicate logic.

5.1.3 Hilbert-style Axiom Systems

The following is an axiomatic theory L for the propositional calculus [Men87].

1. Axioms[3]: For formulae $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$:
   (A1) $\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A})$.
   (A2) $(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C}))$.
   (A3) $(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B})$.
2. Rule of Inference (**Modus-Ponens** or MP):
   $\mathcal{B}$ is a direct consequence of $\mathcal{A}$ and $\mathcal{A} \Rightarrow \mathcal{B}$. We write this as
   `if` $\vdash \mathcal{A}$ `and` $\vdash \mathcal{A} \Rightarrow \mathcal{B}$ `then` $\vdash \mathcal{B}$.

The theory L is an example of a *Hilbert-style* axiom system. Let us provide a proof of the tautology $\mathcal{A} \Rightarrow \mathcal{A}$ in the axiomatic theory L.

1. $(\mathcal{A} \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A})) \Rightarrow ((\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A})) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A}))$
   Instance of axiom schema (A2)
2. $\mathcal{A} \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A})$ Axiom schema (A1)
3. $(\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A})) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A})$ From 1 and 2 by MP
4. $\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A})$ Axiom schema (A1)
5. $\mathcal{A} \Rightarrow \mathcal{A}$ From 3 and 4 by MP

Notice that the proof consists of a finite number of *steps*, each one of which has an attached *justification* for its inclusion in the proof — each step is either an instance of an axiom schema or a direct consequence of some two previous steps by virtue of the rule of inference, Modus-Ponens. As can be seen, the proof is quite tedious and not very intuitive, i.e., it seems artificial and contrived, and is unlike proofs we see in mathematics textbooks. To address this problem, the logician *Gerhard Gentzen* came up with a formal theory that has been aptly termed "natural deduction". Natural deduction proofs are closer to intuitionistic proofs provided in mathematics.

[3]These are not axioms (they are infinitely many), but *axiom schemas* which specify the **syntactic form** of formulae to be deemed axioms.

5.2 Natural Deduction

Natural Deduction is an axiomatic theory for carrying out formal proofs which resemble conventional mathematical proofs. In mathematics, it is common to prove a statement A on the assumption of a set of statements $B_1, B_2, \ldots$. This is formalised by the notion of a *judgement*, which is of the form , $\vdash A$ where , is a set of formulae and A is a formula. A Natural Deduction proof is a sequence of judgements. There is only one axiom schema, which asserts that a formula $\mathcal{A}$ is a logical consequence of itself ($\mathcal{A}$). There are rules of inference for the "introduction" and "elimination" of each logical connective. A rule of inference is applicable if there are judgements in the proof which satisfy all its premises. Some of the rules of inference "cancel out" formulae in the set , , allowing one to provide proofs of theorems. A Natural Deduction axiomatic theory for the propositional calculus is shown below:

1. A single axiom schema:
   $(DirectConsequence)$ $\mathcal{A} \vdash \mathcal{A}$.

2. Rules of inference: For a set of formulae , and formulae $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$,
   ($\wedge$–**I**) if , $\vdash \mathcal{A}$ and , $\vdash \mathcal{B}$ then , $\vdash \mathcal{A} \wedge \mathcal{B}$.
   ($\wedge$–**E1**) if , $\vdash \mathcal{A} \wedge \mathcal{B}$ then , $\vdash \mathcal{A}$.
   ($\wedge$–**E2**) if , $\vdash \mathcal{A} \wedge \mathcal{B}$ then , $\vdash \mathcal{B}$.
   ($\vee$–**I1**) if , $\vdash \mathcal{A}$ then , $\vdash \mathcal{A} \vee \mathcal{B}$.
   ($\vee$–**I2**) if , $\vdash \mathcal{B}$ then , $\vdash \mathcal{A} \vee \mathcal{B}$.
   ($\vee$–**E**) if , $\vdash \mathcal{A} \vee \mathcal{B}$ and , $\cup \{\mathcal{A}\} \vdash \mathcal{C}$ and , $\cup \{\mathcal{B}\} \vdash \mathcal{C}$ then , $\vdash \mathcal{C}$.
   ($\Rightarrow$–**I**) if , $\cup \{\mathcal{A}\} \vdash \mathcal{B}$ then , $\vdash \mathcal{A} \Rightarrow \mathcal{B}$.
   ($\Rightarrow$–**E**) if , $\vdash \mathcal{A}$ and , $\vdash \mathcal{A} \Rightarrow \mathcal{B}$ then , $\vdash \mathcal{B}$.
   ($\neg$–**I**) if , $\cup \{\mathcal{A}\} \vdash$ `ff` then , $\vdash \neg\mathcal{A}$.
   ($\neg$–**E**) if , $\vdash \mathcal{A}$ and , $\vdash \neg\mathcal{A}$ then , $\vdash$ `ff`.
   ($\neg\neg$–**I**) if , $\cup \{\neg\mathcal{A}\} \vdash$ `ff` then , $\vdash \mathcal{A}$.

Here's the proof of section 5.1.3 in Natural Deduction:

1. $\mathcal{A} \vdash \mathcal{A}$ (DirectConsequence)

2. $\vdash \mathcal{A} \Rightarrow \mathcal{A}$ ($\Rightarrow$–I)

5.2.1 Computers Assistance for Proofs

In general, providing proofs requires ingenuity — there is no effective procedure (algorithm) to find proofs. How, then, can computers help us with proofs? The situation is very similar to programming — program construction is not effective, i.e., ingenuity is required for program development. However, there is hardly a programmer who is unconvinced about computer assisted program construction! Computers are very good at automating the boring and tedious parts of programming — program editors help construction, source code control systems manage versions, programmers build libraries for code sharing and reuse, and so on. Analogously, there are many ways in which computers could be used to assist proof creation:

- *Proof Development:* As in programming, proof construction should proceed by stepwise refinement. To provide the proof of a theorem, we start off by providing proofs for sub-theorems (or "lemmata" as they are called), to make the main proof easier to develop. If lemmata are designed carefully, they may be (re)used in several contexts within the proof. A system could be used to keep track of all previously proved lemmata and also help in (partially) automating proofs by supplying proofs of certain lemmata.

- *Proof Management:* Long proofs are usually split into several files, for efficient access and checking. Ideas of modular programming, such as separation of the interface and implementation (`.h` and `.c` files) may be directly used in proof development. Generic language processors such as the *C* preprocessor are also very useful for increasing readability and reducing proof development time.

- *Proof Debugging:* Proofs could be long and involved; it is not surprising, therefore, that most proofs start off having bugs. Even for undecidable theories the process of *proof checking* is effective; computers may therefore be used to debug proofs. This is particularly important for proofs in software engineering, which are not subject to as much rigorous scrutiny as are proofs of important theorems in mathematics.

- *Proof Libraries:* These are valuable in increasing the level at which proofs are carried out. Using carefully constructed proof libraries, the time to develop proofs encountered in a specific engineering domain can be shortened by several orders of magnitude. Techniques and tools for code reuse and library management are therefore invaluable for proof development.

5.2.2 A Survey of Computer Assisted Proof Systems

LCF — Logic for Computable Functions LCF [GMW79] is an interactive theorem prover which uses subgoaling strategies known as *tactics* for carrying out proofs. Theorem proving primitives such as inference rules, tactics, and simplifiers, as well as all terms, formulae, and theorems of the logic are expressed in a meta-language (ML), a functional programming language. Proofs are carried out in PPLAMBDA, a natural deduction logic based on the domain theory of Dana Scott. This logic is particularly suitable for proofs involving denotational semantics, lazy evaluation, or higher-order functions. Users can extend LCF by programming in ML. The ideas used to implement LCF have served as the basis for several other interactive theorem proving systems, including HOL, COQ, Nuprl and Isabelle.

The HOL System by *Mike Gordon, Konrad Slind et. al.*.

The HOL system [GM93] supports interactive proof development in higher order logic, a polymorphic version of Church's Simple Theory of Types. As in LCF, the formal logic is expressed in ML, in which terms and theorems of the logic are denoted, proof strategies expressed and applied, and logical theories developed. The system provides a lot of automated support including:

- A powerful rewriting package
- Pre-installed theories for booleans, products, sums, natural numbers, lists, and trees
- Definition facilities for recursive types and recursive functions over those types (including mutual recursion)

- Extensive libraries for strings, sets, group theory, integers, reals, well ordered sets, automatic solution of goals in linear arithmetic, tautology checking, Hoare logic, Chandy and Misra's UNITY theory etc.

Other features of HOL include:

- It uses ML (meta-language), a high-level (impure) functional programming language featuring strong polymorphic typing: this allows the user to program custom proof procedures.
- It encapsulates theorems as ML abstract types, whose only constructors are the primitive inference rules of the logic. This provides a high degree of security, since the critical core is small, and all additional theorems, or inference rules are secure extensions of the underlying logic.
- All theories are built up by conservative, definitional extension, which again gives security (no extra axioms are asserted).
- Built-in functions provide backward (goal-directed) theorem proving which can be freely mixed with forward proofs.

COQ System COQ [Dow93] implements the Calculus of Inductive Constructions, a powerful logic accepting higher-order reasoning, and allowing the definition of a wide class of functions. The system also allows the extraction of proven-correct ML programs from proofs.

COQ provides a rich specification language, called Gallina, which blends together higher-order predicate calculus, inductive types and inductive predicates in the style of logic programming, and recursive function definitions with pattern matching, within a uniform type theory. A proof assistant driven by tactics allows the incremental construction of partial proof trees, with some limited automation. The specification language allows the distinction between concrete constructions and abstract reasoning. From the concrete part of a proof an ML program may be extracted, guaranteed to satisfy the specification which has been proved. It is also possible to derive semi-automatically proofs of correctness from the program decorated with invariants and its specification.

Isabelle Isabelle [Pau90] is a generic theorem prover in which new logics may be introduced by specifying their syntax and rules of inference. Proof procedures may be expressed using tactics and tacticals as in LCF. Isabelle can support a wide range of logics, and comes with several built-in ones:

- many-sorted first-order logic, constructive and classical versions
- higher-order logic, similar to that of HOL
- Zermelo-Fraenkel set theory
- an extensional version of Martin-Löf's Type Theory
- the classical first-order sequent calculus LK
- the modal logics T, S4, and S43
- the Logic for Computable Functions, LCF
- the Lambda Cube

Nuprl Proof development system described in [Jac94b], the Nuprl system is implemented in a combination of Common Lisp and (non-standard) ML. The system is replete with features, with its own formula editor, library packages for building mathematical and logical databases of theories, several built-in theories of mathematics, and powerful routines for equational reasoning, decision procedures for propositional logic and a fragment of arithmetic, and conditional term rewriting. The Nuprl logic [Con86] is built for constructive reasoning, and is mostly intended for carrying out proofs in Mathematics. The system is extremely complex and intimidating for casual use by non-experts.

5.2.3 The User Interface

The architecture and user interface of all the systems we surveyed are along the lines of LCF, characterised by the following features:

1. Underlying metalanguage is ML (users are expected to be familiar with the language).

2. Users interact with the system by invoking tactics/tacticals (lengthy lists of tactics are included in user manuals).

3. Proofs are carried out in the "backward" direction.

4. Invocation of a tactic results in the system responding with a list of subgoals.

5. Proofs have a tree structure; users may "navigate" proof trees, albeit very clumsily.

6. Target logics are embedded in an underlying "meta-logic".

7. Safety (i.e., the system ensuring that only sound theorems are deduced) is achieved through ML's type system.

8. Each use of a theorem or derived rule entails redoing every step of its proof.

9. Users interact with the system via ML's "outer loop".

10. Users cannot usually predict system response; each step of a proof therefore can be provided only after examining the system response to previous step(s). However, because systems are deterministic, a proof can be "replayed" non-interactively.

11. Users must be intimately familiar with built-in tactics and tacticals (at least the basic ones) and should be able to write ML code to implement others. Unfortunately, tactics built into one system (including the basic ones) usually bear no resemblance to the ones in other systems. Therefore, although the systems are architecturally similar, switching between them is very difficult.

5.2.4 Some Examples

In this section, we shall compare the user interface of two popular systems — HOL and Nuprl — with the interface reported in this thesis, by looking at illustrative examples derived from tutorial introductions to these systems ([GM93, Jac94b]).

A Proof using HOL (from [GM93], pp. 25–27).

The theorem to be proved is $\vdash t1 \Rightarrow ((t1 \Rightarrow t2) \Rightarrow t2)$. In [GM93], a proof of this theorem using standard notation (sic) is given.

1. $t1 \Rightarrow t2 \vdash t1 \Rightarrow t2$ (Assumption introduction)
2. $t1 \vdash t1$ (Assumption introduction)
3. $t1 \Rightarrow t2, t1 \vdash t2$ (Modus Ponens applied to lines 1 and 2)
4. $t1 \vdash (t1 \Rightarrow t2) \Rightarrow t2$ (Discharging the first assumption of line 3)
5. $\vdash t1 \Rightarrow (t1 \Rightarrow t2) \Rightarrow t2$ (Discharging the only assumption of line 4)

The following is a transcript of a HOL session to carry out this proof. The commands given by the user are prefixed by the pound sign (the system prompt); other lines are the system responses.

```
#let Th3 = ASSUME "t1==>t2";;
Th3 = . |- t1 ==> t2
#dest_thm Th3;;
["t1 ==> t2"], "t1 ==> t2" : goal
#let Th4 = DISCH "t1==>t2" Th3;;
Th4 = |- (t1 => t2) ==> t1 ==> t2
#let Th5 = ASSUME "t1:bool";;
Th5 = . |- t1
#let Th6 = MP Th3 Th5;;
Th6 = .. |- t2
#let Th7 = DISCH "t1==>t2" Th6;;
Th7 = t1 |- (t1 ==> t2) ==> t2
#let Th8 = DISCH "t1:bool" Th7;;
Th8 = |- t1 ==> (t1 ==> t2) ==> t2
```

Note that names of inference rules (“`MP`”, “`DISCH`”) are to be explicitly provided, together with associated arguments in a predefined order. Having to provide exact names of inference rules, including their arguments in the order in which they were defined makes the user interface rather awkward. HOL proponents therefore advocate *backward* proofs, where the theorem to be proved (goal) is successively split into multiple sub-goals by the application of inference rules in the “backward” direction. For example, consider the following Natural Deduction inference rule:

($\wedge$–**I**) if , $\vdash \mathcal{A}$ and , $\vdash \mathcal{B}$ then , $\vdash \mathcal{A} \wedge \mathcal{B}$.

A “backward” proof rule that corresponds to the above rule would split a goal of the form , $\vdash \mathcal{A} \wedge \mathcal{B}$ into the two sub-goals , $\vdash \mathcal{A}$ and , $\vdash \mathcal{B}$.

The following sequence of commands carry out the same proof in our system Snap:

```
prove |- t1 => ((t1 => t2) => t2);
assume t1=>t2;
assume t1;
t2;
t1 |- (t1 => t2) => t2;
|- t1 => ((t1 => t2) => t2);
qed Th3;
```

Note that the above steps are a literal transcription, in the syntax of Snap, of the steps in the standard proof. Also note that the user does not even have to *know* the actual name of an inference rule in the system, nor specify its use explicitly. The example above is meant to illustrate that it is possible to handle standard proofs with an elegant and simple user interface.

Also note that the above proof need not be carried out interactively. The proof may very well be edited as a file using `vi`, and then provided as input to Snap (as was done in this case). Snap provides the following annotated output for the proof, with a run time of under a second:

```
Th3:  |- t1 => ((t1 => t2) => t2)
Proof:
  1. (t1 => t2)                                            Assumption
  2. t1                                                    Assumption
  3. t2                                                 impElim, 2, 1
  4. t1 |- (t1 => t2) => t2                               impIntro, 3
  5. |- t1 => ((t1 => t2) => t2)                          impIntro, 4
QED.
```

A Proof in Nuprl (from [Jac94a]). In [Jac94a], a proof of a theorem of propositional calculus, $\vdash (\neg b \vee \neg c) \Leftrightarrow \neg(b \wedge c)$, is given as a tutorial introduction to the Nuprl system. To begin with, the theorem to be proved has to be entered into the system, using a system-specific formula editor. Following this, the tutorial proceeds to explain how the proof is carried out, using Nuprl's tactics. However, at the end of the tutorial, the proof is left incomplete. Here's a (partial) list of the tactics used for this proof:

```
D 0 <C-z>
D 0 <Return> THENW Auto <C-z>
D 0 <C-z>
D 0 THENW Auto  (for each subgoal)
GenUnivCD THENW Auto
ClassDecide A <C-z>
Sel 1 (D 0)  THENW Auto
Hypothesis     Sel 2 (D 0)    or is it D 3?
```

The point we are trying to make here is that, even after careful perusal of a tutorial introduction to this system, it is very difficult to (independently) carry out a similar proof without additional training (we concede that we still do not have a clear understanding of the above proof steps).

Another point to note is that knowledge of the user interface (set of tactics) of one system does not help in learning the user interface of another.

Now, let us see how a proof of the above theorem can be carried out in SNAP. *This particular theorem can be proved automatically, using* SNAP*'s built-in simplification routines.* However, we shall carry out a proof using elementary rules of deduction, to illustrate how proofs by contradiction are easily carried out in SNAP. These proofs use the following lemma:

```
contradiction:   if ~X |- P and ~X |- ~P then |- X;
```

whose proof is left as an exercise for the reader.

```
Lemma1:  |- ~(b /\ c) => (~b \/ ~c)
Proof:
  1. ~(b /\ c)                                  Assumption
  2. ~(~b \/ ~c)                                Assumption
  3. ~b |- ~b \/ ~c                              disjIntro
  4. b                                  contradiction, 2, 3
  5. ~c |- ~b \/ ~c                              disjIntro
  6. c                                  contradiction, 2, 5
  7. b /\ c                                 conjIntro, 4, 6
  8. ~(b /\ c) |- ~b \/ ~c              contradiction, 1, 7
  9.  |- ~(b /\ c) => (~b \/ ~c)               impIntro, 8
QED.

Lemma2_1: ~b |- ~(b /\ c)
Proof:
  1. ~b                                         Assumption
  2. b /\ c                                     Assumption
  3. b                                         conjElim, 2
  4. ff                                     negElim, 3, 1
  5. ~b |- ~(b /\ c)                           negIntro, 4
QED.

Lemma2_2: ~c |- ~(b /\ c)
Proof:
  1. ~c                                         Assumption
  2. b /\ c                                     Assumption
  3. c                                         conjElim, 2
  4. ff                                     negElim, 3, 1
  5. ~c |- ~(b /\ c)                           negIntro, 4
QED.
```

```
Lemma2:  |- (~b \/ ~c) => ~(b /\ c)
Proof:
  1. ~b \/ ~c                                    Assumption
  2. ~b |- ~(b /\ c)                               Lemma2_1
  3. ~c |- ~(b /\ c)                               Lemma2_2
  4. ~(b /\ c)                              disjElim, 3, 2, 1
  5.  |- (~b \/ ~c) => ~(b /\ c)                impIntro, 4
QED.

DeMorgan:  |- (~b \/ ~c) <=> ~(b /\ c)
Proof:
  1. ~(b /\ c) => (~b \/ ~c)                         Lemma1
  2. (~b \/ ~c) => ~(b /\ c)                         Lemma2
  3. (~b \/ ~c) <=> ~(b /\ c)              biconIntro, 2, 1
QED.
```

An important point to note here is that the motivation and reasoning behind each step of the above proof may be gleaned from a logic textbook (this proof was adapted from [Gri81] pp. 49–50).

Chapter 6

Snap — A Proof Validator

6.1 Why another system?

There is a proliferation of systems for machine aided proofs. At one end of the spectrum, "automatic theorem provers" [SB89] are intended to *find* proofs. Finding a proof, however, cannot be guaranteed — using an automatic theorem prover entails manipulating several system parameters that will (hopefully) guide the search towards a proof. Several semi-automatic systems, known as "interactive theorem provers" also exist. These systems mainly provide proof management functions; for instance, they make sure that there are no unproved lemmata or undischarged assumptions, before declaring a proof complete. Most of these systems also have several built-in "tacticals" — heuristics designed to work in interesting (but, by no means all) cases automatically. Finally, at the other end of the spectrum, "proof checkers" verify that each entered proof step is either in a set , of formulae, or is a direct consequence of a subset of previous steps by virtue of a rule of inference. Note that the set , , proof steps, and inference rule are all explicitly specified by the user.

Can we not use an existing system for our needs? Surprisingly, the answer seems to be no. To understand why, let us examine our requirements for a theorem proving system:

1. Anyone who can prove theorems, in a way prescribed in logic texts, should be able to use the system without much effort.

2. The system should be accessible to engineering professionals who use other tools such as yacc or LaTeX as a part of their work — not just to logicians or mathematicians.

3. The system may be used either interactively or in batch mode.

4. The system should be architected along the lines of UNIX utilities, so that proofs may be edited, processed, managed, and archived using common programming tools such as `vi`, `cpp`, `make`, or `rcs`.

5. First order logic should be built-in. There is no need for a higher-order logic "metalanguage" in which other logics could be embedded.

6. Our purpose is often to prove "junk theorems" [Par93a] encountered in systems engineering, and not just to do advanced mathematics.

7. Proofs such as showing $0 \neq 1$ from a basic set of axioms are of no interest to us; it is sufficient to inspect such assumptions and accept them as being true. Ideally, they should be verified automatically.

8. The system should be able to "import" previously proved theorems without having to enter or redo their proofs.

Of the three kinds of systems mentioned above, interactive theorem provers (surveyed briefly in chapter 5) seem to be the closest to what we want. However, it is widely recognised (for example, see [Pau90] pg. 384) that systems built along the lines of LCF are hard to learn and use. To quote from [Pau90]:

> Isabelle's user interface is no advance over LCF's, which is widely condemned as "user-unfriendly": hard to use, bewildering to beginners. $<\ldots>$ But Edinburgh LCF was invented because real proofs require millions of inferences. Sophisticated tools — rules, tactics and tacticals, the language ML, the logics themselves — are hard to learn, yet they are essential. [Users] may demand a mouse, but [what they really] need [is] better education and training.

We remained unconvinced that these systems necessarily have to be user-unfriendly and bewilder beginners. Our desire for a better user interface made us build system SNAP, which provides an improved user interface for semi-automatic theorem proving.

One of the major limitations of existing systems is their inability to satisfy our first requirement — in addition to users having to know how to prove theorems as in logic texts, they are expected to learn a system's built-in commands (which sometimes number in hundreds) before being able to carry out a proof. More important, these commands are different for each system. On the other hand, to use SNAP only requires the ability to carry out textbook-style proofs. This has been confirmed in demonstrations, in which both computer scientists as well as logicians[1] have considered it easy to use. After some instruction, a few of them (who were familiar with pencil and paper proofs in natural deduction) were immediately able to use the system and carry out simple proofs.

6.2 The Design Objectives of SNAP

- *Ease of Use:* There is a lot of obfuscation in the field of logic; theorem proving systems are no exception. Most systems are implemented using programming languages and environments that are unfamiliar to engineers. More importantly, their system manuals and user interfaces are intimidating even to seasoned mathematicians. This is because they do not use terminology and proof methods as in logic textbooks — the syntax, techniques, and style of proofs are unique to each system. These differences range from minor (e.g., most systems use the "prefix" notation for formulae; the wff `((A /\ B) => A)`, for instance, would be written as `(imp (and (A) (B)) (A)))` to more serious (e.g., proofs in most systems are carried out in a "backward" direction; proofs in textbooks are usually presented in the forward direction).

- *Learning Curve:* Most systems are "heavyweight" — users are expected to learn a new programming language, usually ML, the meta-language of the system.

[1]The list includes Amy Felty, David Gries, Tim Griffin, Gerard Holzmann, Mehdi Jazayeri, Jonathan Ostroff, David Parnas, Doron Peled, Dennis Ritchie and Jeffery Zucker.

In addition, users must familiarise themselves with hundreds of "tactics" and other built-in functions before they can use the system efficiently. Practitioners, especially casual users, cannot be expected to invest this time and effort.

- *Intended Use:* Most systems are tailored for carrying out proofs in mathematics. Our goal is somewhat more mundane — computer assistance for establishing validity of "junk theorems" in first-order logic. Features such as higher-order logics, and the ability to embed arbitrary logics make systems needlessly complex for our purpose.

Snap has been designed for the express purpose of carrying out proofs in a first-order theory. We have sacrificed generality for efficiency and ease of use. The design objective is that users who are able to prove theorems in a way satisfactory to some logic text, should be able to use the system with minimal difficulty. Proofs in Snap should therefore be as "natural" as possible — most steps that are considered "obvious" should be allowed. The system should also allow users to carry out proofs in a style they are used to, with minimal restrictions. Mechanisms, such as definitions of new theorems and inference rules, should be provided to increase the level of abstraction of proofs (i.e., skip steps without compromising correctness). We expect users from specific application areas to develop and maintain libraries of theorems, lemmata, and inference rules tailored to their needs. Most importantly, users must not be expected to remember the names assigned to lemmata and inference rules as entered into the system — an intuitive understanding of the theorem or inference rule should be sufficient to use it. In addition, if a specific step of a proof requires the use of an unproved lemma, users should have the option of either providing an immediate proof of the lemma, or simply using the lemma and finishing the proof, with the proof of the lemma provided at a later stage.

6.3 The User Interface

6.3.1 Conventional Wisdom

Conventionally, computer assisted proofs have been carried out in two ways:

1. *Proof Checking:* The user provides the complete proof (proof steps and inference rules or theorems as justifications) and the system checks them for correctness.

2. *Interactive Theorem Proving:* The user provides detailed instructions to the system for deducing each proof step, by invoking commands (with suitable arguments) that instantiate axiom schemas or use inference rules.

An important point to note is that in either case, the user must be intimately familiar with the names, arguments (including the order in which they are to be given) and the exact syntax of every axiom and inference rule in the system (for instance, to apply the inference rule *modus ponens*, should one provide ($\mathcal{A}$, $\mathcal{A} \Rightarrow \mathcal{B}$) or ($\mathcal{A} \Rightarrow \mathcal{B}$, $\mathcal{A}$) as arguments?).

The problem is exacerbated when more theorems and inference rules are added to the system in order to raise the level of abstraction of proofs. Elaborate listings of "tactics" and "tacticals" are therefore provided, and users are expected to memorise them (or, at least learn where to look). If a user is unaware of the exact name assigned to a specific theorem within the system, it becomes impossible to use, even if the user *knows* that theorem.

6.3.2 The Snap Interface

Snap is based on the **s**imple **n**otion of **a**nnotating **p**roofs. We have noticed that when providing proofs, users generally have a good idea of what the next proof step should be. However, they have to go through a lot of gyrations to *instruct* conventional system to derive the step, leading to a great deal of frustration. Snap explores a new approach — why not let users write down *what* they want, and let the system figure out if it is a valid proof step by searching for a justification. Termed **proof annotation**, this simple idea also finesses the problem of users having to memorise exact names of theorems and inference rules. Snap can have a large *rule-base* of theorems and inference rules, which novices can use at will, even if they have gleaned them from textbooks. Snap is feasible to implement because, given a set , of formulae, a formula $\mathcal{A}$, and a set inference rules R, the problem of deciding whether , $\vdash_{R_i} \mathcal{A}$, where $R_i \in R$, is computable in polynomial time. We have devised such an algorithm for Snap, which is not only extremely frugal, but is also parallelisable for execution on distributed systems.

6.4 The Snap System

We now illustrate system Snap by examples. This section has been adapted from [BS94].

Proof Rules: For the purpose of demonstration, we shall assume that the set of proof rules listed in figure 3 are in Snap's rule-base. In figure 3, $P_1, \cdots, P_n \mid\Leftrightarrow P$ is to be interpreted as $\{P_1, \cdots, P_n\} \vdash P$, i.e., P is derivable from $P_1, \cdots, P_n$.

```
DirectConseq:  P |- P;

conjIntro: A,   B  |-  A /\ B;
conjElim:  A /\ B  |-  A;
conjElim:  A /\ B  |-  B;
disjIntro: A  |-  A \/ B;
disjIntro: B  |-  A \/ B;
disjElim:  if |-  A \/ B  and  A |- C  and  B |- C  then  |- C;
impIntro:  if A |- B  then |- A => B;
impElim:   A, A=>B  |-  B;
negIntro:  if P |- ff then |- ~P;
negElim:   A, ~A |- ff;
doubleNeg:  if ~P |- ff then |- P;
biconIntro: (A  => B), (B => A) |- (A <=> B);
biconElim:  (A <=> B)           |- (A  => B);
biconElim:  (A <=> B)           |- (B  => A);
```

Figure 3: *Basic Set of Proof Rules*

The only proof rule that is hard-coded into Snap's algorithms is the monotonicity property "if , $\subseteq$, $'$ and , $\vdash P$ hold, then , $' \vdash P$ holds". Users interact with Snap by means of *commands*, each command being terminated by a semicolon ";". The system records the status of each proof as follows: set S records formulae assumed during the proof, and sequence L records the proof steps.

Initiating a Proof: A proof is initiated by keyword `prove`, optionally followed by *premises*, followed by the *goal*. Premises are enclosed within `if` ... `then`, and separated by symbol `and`. Each premise and the goal are of the form $P_1, \cdots, P_n \vdash P$. In the following, we refer to a premise or the goal of a proof respectively as "a premise of the proof" or "the goal of the proof". Keyword `goal` is an alias for the goal of the current proof. At initiation, a proof's associated set of assumptions S and sequence of proof steps L are both empty.

Introducing Assumptions: Command `assume` P, where P is a predicate (i.e., does not contain symbol "$\vdash$"), adds P to set S and appends the step $P \vdash P$ to sequence L. The system generates the justification "`Assumption`" for the proof step.

Entering Proof Steps: A Proof step is provided by directly entering a formula (without any keyword). If the formula is a premise of the proof, the system generates the justification "`Hypothesis`" and appends it to sequence L. If a step is justifiable on the basis of a theorem or meta rule in SNAP's rule-base, the system generates an appropriate justification and appends the step to sequence L. Entering $P_1, \cdots, P_n \vdash P$ as a proof step is interpreted as "P is derivable under assumptions $\{P_1, \cdots, P_n\}$ in one step". Additionally, SNAP uses the convention that to enter formula $S \vdash P$, one merely enters P.

A proof is deemed complete if the last step in L is identical to the goal of the current proof (i.e., the formula associated with the keyword "`goal`").

Saving Theorems: When proved, the user may save a theorem (without its proof) in the rule-base, by entering command `qed` followed by the name to be assigned to the corresponding theorem in the rule-base. A theorem saved in this manner is *immediately available* for use (i.e., serve as justifications in subsequent proofs).

To demonstrate the use of SNAP, let us first examine how a proof of $\neg\neg P \vdash P$ is given using pencil and paper: Assume that $\neg\neg P$ holds. Assume that $\neg P$ also holds.

Then, `ff` holds, too (by proof rule `NegElim`). Thus, $\neg\neg P \vdash P$ follows (by proof rule `doubleNeg`). This concludes the proof.

To carry out this proof in Snap, one enters the commands below. Note the direct correspondence between the proof on paper and the proof provided to Snap.

```
prove ~~P |- P;
assume ~~P;
assume ~P;
ff;
~~P|-P;
qed Double_Neg_Elim;
```

The (un-edited) output produced by the system is give below.

```
Double_Neg_Elim: (~(~P))|- P
Proof:

  1: (~(~P))
                                                        (Assumption)
  2: (~P)
                                                        (Assumption)
  3: ff
                                                     (negElim, 2, 1)
  5: (~(~P))|- P
                                                      (doubleNeg, 3)
QED
```

As another example, we present a proof of the cut-rule:

```
prove if |-P and P|-Q then |-Q;
P;
P|-Q;
|-P=>Q;
goal;
qed Cut_Rule;
```

The (un-edited) output produced by the system is:

```
Cut_Rule: if |- P  and  P|- Q then |- Q
Proof:

  1: |- P
                                                  (Hypothesis)
  2: P|- Q
                                                  (Hypothesis)
  3: (P => Q)
                                                 (impIntro, 2)
  4: Q
                                               (impElim, 1, 3)
QED
```

In addition to the commands discussed above, SNAP also offers commands to view proof steps (with associated justifications), inference rules, assumptions, and the proof goal.

In the remainder of this section we discuss how SNAP can be used to provide correctness proofs of programs represented as in chapter 4. We have already seen how proofs of first-order predicates are given using SNAP. Let us now consider how proofs of annotated transitions (c.f. chapter 4) are provided in SNAP.

Consider the following example:

```
prove |-{x=X/\y=Y} x:=y, y:=x {x=Y/\y=X};
assume x=X/\y=Y;
x=X;
y=Y;
y=Y/\x=X;
|- {x=X/\y=Y} x:=y, y:=x {x=Y/\y=X};
qed swap;
```

The (un-edited) output produced by SNAP is:

```
swap: |- (((x = X) /\ (y = Y)) => ((y = Y) /\ (x = X)))
Proof:

  1: ((x = X) /\ (y = Y))
                                                  (Assumption)
  2: (x = X)
                                                 (conjElim, 1)
  3: (y = Y)
                                                 (conjElim, 1)
  4: ((y = Y) /\ (x = X))
                                              (conjIntro, 3, 2)
  5: |- (((x = X) /\ (y = Y)) => ((y = Y) /\ (x = X)))
                                                 (impIntro, 4)
QED
```

6.5 SNAP's *Simplify*

SNAP provides an option for skipping proof steps that are considered "obvious" by users. Using a process called associative-commutative rewriting, the algorithm *Simplify* built into SNAP may be used to derive a justification for such a proof step. As the justification may involve the application of several individual inference rules, the system merely states "by simplification" as the associated justification. Note, however, that *Simplify* is **not** a decision procedure for propositional logic. In general, such decision procedures usually end up doing a lot of useless work, especially in cases where the formula is *not* valid. Using *Simplify*, some trivial theorems may be proved in one step. The following is an example of such a proof:

```
Lemma1.7:  |- ((A /\ B /\ C) => A \/ D \/ X)

Proof:

  1. ((A /\ B /\ C) => A \/ D \/ X)
                                           by simplification
QED.
```

6.6 Application to Dekker's Algorithm

In this section we show how SNAP may be used to validate correctness proofs of programs. For this purpose, we consider Dekker's algorithm, a concurrent program that ensures mutual exclusion for two processes. We shall concentrate on proving a safety property which asserts that the two processes cannot be in their critical section at the same time. The proof is from [MP83, BS94].

```
int c1 = 1, c2 = 1, turn = 1;

process class p(int my_c, int his_c, int his_turn) {
begin:  my_c := 0;
try_again:
    choice {
        : his_c # 0, goto crit
        : his_c = 0                         };
    choice {
        : turn=1, goto try_again
        : turn # 1                          };
    my_c1 := 1;
    loop {
        : turn = his_turn
        : turn # his_turn, goto begin       };
crit:
    turn := his_turn, my_c := 1, goto begin }

init {
    par {
        : p pc1(c1, c2, 2)
        : p pc2(c2, c1, 1)                  } }
```

Figure 4: *Dekker's Algorithm in* MELA

A MELA description of Dekker's algorithm is presented in figure 4. It has been taken from [BA82] and is an abstraction in the sense that the code in the bodies of the critical sections has been removed. For this program, we wish to establish the

LTL formula $\Box\neg(pc1 = crit \wedge pc2 = crit)$, where $pc1$ represents the first process's program counter, $pc2$ represents the second process's program counter, and $crit$ is a label in the program, which plays the role of critical sections.

The program has three global variables: $c1$, $c2$, and $turn$. Initially, each of these variables has value 1. If the first process wishes to enter its critical section, it indicates this by assigning 0 to variable $c1$. Variable $c2$ is used for the same purpose by the second process. Variable $turn$ is used to resolve the conflict that arises when both processes have indicated their intention to enter their critical sections.

To begin with, we translate the program in figure 4 into its equivalent transition system (shown in figure 5), using the definitions of chapter 3.

$\Theta = (pc1 = 0) \wedge (pc2 = 0) \wedge (c1 = 1) \wedge (c2 = 1) \wedge (turn = 1)$

τ_1: $pc1{=}0 \Leftrightarrow\!\!\!\rightarrow c1 := 0,\ pc1 := 1$
τ_2: $pc1{=}1 \wedge c2 = 0 \Leftrightarrow\!\!\!\rightarrow pc1 := 2$
τ_3: $pc1{=}1 \wedge c2 \neq 0 \Leftrightarrow\!\!\!\rightarrow pc1 := 5$
τ_4: $pc1{=}2 \wedge turn = 1 \Leftrightarrow\!\!\!\rightarrow pc1 := 1$
τ_5: $pc1{=}2 \wedge turn \neq 1 \Leftrightarrow\!\!\!\rightarrow pc1 := 3$
τ_6: $pc1{=}3 \Leftrightarrow\!\!\!\rightarrow c1 := 1,\ pc1 := 4$
τ_7: $pc1{=}4 \wedge turn = 2 \Leftrightarrow\!\!\!\rightarrow pc1 := 4$
τ_8: $pc1{=}4 \wedge turn \neq 2 \Leftrightarrow\!\!\!\rightarrow pc1 := 0$
τ_9: $pc1{=}5 \Leftrightarrow\!\!\!\rightarrow turn := 2,\ c1 := 0,\ pc1 := 0$

τ_{10}: $pc2{=}0 \Leftrightarrow\!\!\!\rightarrow c2 := 0,\ pc2 := 1$
τ_{12}: $pc2{=}1 \wedge c1 = 0 \Leftrightarrow\!\!\!\rightarrow pc2 := 2$
τ_{13}: $pc2{=}1 \wedge c1 \neq 0 \Leftrightarrow\!\!\!\rightarrow pc2 := 5$
τ_{14}: $pc2{=}2 \wedge turn = 2 \Leftrightarrow\!\!\!\rightarrow pc2 := 1$
τ_{15}: $pc2{=}2 \wedge turn \neq 2 \Leftrightarrow\!\!\!\rightarrow pc2 := 3$
τ_{16}: $pc2{=}3 \Leftrightarrow\!\!\!\rightarrow c2 := 1,\ pc2 := 4$
τ_{17}: $pc2{=}4 \wedge turn = 1 \Leftrightarrow\!\!\!\rightarrow pc2 := 4$
τ_{18}: $pc2{=}4 \wedge turn \neq 1 \Leftrightarrow\!\!\!\rightarrow pc2 := 0$
τ_{19}: $pc2{=}5 \Leftrightarrow\!\!\!\rightarrow turn := 1,\ c2 := 1,\ pc2 := 0$

Figure 5: *Transitions of the* MeLa *program describing Dekker's Algorithm. Transitions* $\tau_1, \cdots, \tau_9$ *are of the first process; transitions* $\tau_{10}, \cdots, \tau_{19}$ *are of the second process.*

Next, we apply rule INV described in chapter 4; we choose ϕ (as in [BS94]) to be the conjunction of:

(a) $turn = 1 \vee turn = 2$, and

(b) $c1 = 0 \vee c1 = 1$, and

(c) $c2 = 0 \vee c2 = 1$, and

(d) $(pc1 = 1 \vee pc1 = 2 \vee pc1 = 3 \vee pc1 = 5) \Rightarrow c1 = 0$, and

(e) $(pc2 = 1 \vee pc2 = 2 \vee pc2 = 3 \vee pc2 = 5) \Rightarrow c2 = 0$, and

(f) $(pc1 = 0 \vee pc1 = 4) \Rightarrow c1 = 1$, and

(g) $(pc2 = 0 \vee pc2 = 4) \Rightarrow c2 = 1$, and

(h) $pc1 = 5 \Rightarrow (pc2 = 1 \vee pc2 = 2 \vee c2 = 1 \vee turn = 1)$, and

(i) $pc2 = 5 \Rightarrow (pc1 = 1 \vee pc1 = 2 \vee c1 = 1 \vee turn = 2)$.

Predicate ϕ expresses that variable $turn$ can take only 1 or 2 as its value (clause (a)); that the variables $c1, c2$ take the values 1 or 2 (clauses (b) and (c)); that $c1 = 0$ if the program counter of the first process is at location 1, 2, 3, or 5 (clause (d)); that $c1 = 1$ if the program counter of the first process is at location 0 or 4 (clause (f)); and that the following holds: if the program counter of the first process is at location 5, then the program counter of the second process is at location 1 or 2, or the value of variable $c2$ is 1, or the value of variable $turn$ is 1 (clause (h)). An intuitive explanation of the clauses (e), (g), and (i) may be provided along the same lines.

We have carried out the proof (presented in Appendix A), and validated it using SNAP. The proof consists of 20 lemmata: 18 show that the invariant is preserved under all transitions, one shows the invariant holding initially, and one shows that the invariant implies the desired mutual exclusion property.

Chapter 7

Snap **Internals**

7.1 The Proof Annotation Algorithm

In this section we describe the algorithm responsible for the generation of justifications of proof steps in SNAP.

Our proof annotation algorithm relies on a function we call `match` (also known as one-way unification), described in [Sie90]. Let a substitution be a sequence of pairs $\langle x, e \rangle$, where x is a variable and e an expression. Variable x can stand either for an arbitrary predicate or term. Expression e is either a predicate or a term. (The actual types of x and e are inferred from their contexts and determined by the parser.) Given two expressions e_1, e_2 and a substitution σ, $\texttt{match}(e_1, e_2, \sigma)$ returns $true$ if there exists a substitution σ' extending[1] σ such that $\sigma'(e_1) = e_2$ (cf. [Sie90]). (Here $\sigma'(e_1)$ denotes the expression obtained from e_1 by simultaneous replacement of every variable x by e when $\langle x, e \rangle$ is in σ'.) If this is the case, then, in addition, variable σ is assigned such a value σ'. Otherwise, i.e., when no such substitution σ' exists, `match` returns $false$ and leaves σ unchanged.

Let $Expn$ denote the type of expressions. Expressions will be denoted by e, possibly subscripted or primed. We also define a type $Gamma$, whose domain is interpreted as a (finite) set of expressions. For convenience, in the discussion below we represent sets as lists. From now on, , $_1, \cdots,$, $_n$ denote variables of type $Gamma$.

[1]The extension of a substitution –a sequence– has its usual meaning.

A proof rule R is a non-empty (finite) sequence of elements of type $Gamma \times Expn$, represented as a list. Sequence $\langle, {}_1, e_1\rangle, \cdots, \langle, {}_n, e_n\rangle$ has the following interpretation: $, {}_1 \vdash e_1$ is derivable under the assumptions $, {}_2 \vdash e_2, \cdots, , {}_n \vdash e_n$. In case $n = 1$, it means that $, {}_1 \vdash e_1$ is derivable under no assumptions. The above sequence has an associated identifier, its *name*. We call $, {}_1 \vdash e_1$ the conclusion of the proof rule, and denote this by *concl*(R). We call $, {}_2 \vdash e_2, \cdots, , {}_n \vdash e_n$ the premises of the proof rule. For a pair $P = \langle, , e\rangle$, e is denoted by $expn(P)$ and , by $hyp(P)$.

Recall that every proof is represented as a (finite) sequence L, as described in Section 6.4, that corresponds to the proof steps so far. For the description of the annotation algorithm, we assume that the user has input a proof step of the form $, \vdash e$. This is converted by the system into a *step* $\langle, , e\rangle$. In the following, we only consider checking for applicability of a single rule R. (It is a trivial extension to check for a set of rules.)

For a rule R, identified by $, {}_1 \vdash e_1, \cdots, , {}_n \vdash e_n$, the annotation algorithm checks whether e_1 can be matched with e under substitution σ. Initially, σ is the empty substitution ϵ. An attempt is then made to match each element of $, {}_1$ with the expression part of a step in the proof. For every such match found, at say position i in sequence L, the algorithm proceeds to check that the hypothesis of that step is a subset of , (modulo substitution). If this is not the case, the algorithm checks whether every element e' of $, {}_1$ can be directly matched with an element of , . (Recall that we assume the monotonicity property: if $, \subseteq , '$ and $, \vdash P$ hold, then $, ' \vdash P$ holds.) If rule R also has premises, the algorithm checks that all of them can be matched in a similar way as above with steps in sequence L.

If the above procedure is successful, the algorithm returns the name of the matched rule. In addition, if all the premises of rule R have been justified on the basis of steps in sequence L, the algorithm also returns a (finite) sequence of natural numbers indicating those steps in L.

If the above process is unsuccessful in finding a justification, the algorithm reports failure.

The following is a description in pseudo-code of the algorithm outlined above. As discussed above, a sequence of natural numbers may be provided as part of the justification. In order to concentrate on the essentials of the algorithm, we have not shown how such sequences are generated in the code. Also, the function merely checks whether a given rule applies, and does not return the rule's name if it applies. Below, *Subst* denotes the type of substitutions. For list ℓ, $\mid \ell \mid$ denotes its length, and $\ell[i]$ denotes its i^{th} element. Operations on sets are denoted as usual; the notation $, \Leftrightarrow e$ denotes $, \Leftrightarrow \{e\}$. The annotation function is called `is_justification`. We also define four auxiliary functions. (Observe how these functions rely heavily on backtracking.)

Function `match_steps`$(, _1, , _2, L, N, \sigma)$ checks, for each of the first N elements e of $, _1$, whether there exists a step *step* in L such that the gamma part of *step* is a subset of $, _2$ and e matches the expression part of *step* under substitution σ.

```
integer function match_steps$(, _1, , _2, L, N, \sigma)$:
    Subst $\sigma'$;
    if $N = 0$ then return true fi;
    for $i := 1$ to $\mid L \mid$
    do $\sigma' := \sigma$;
        if $hyp(L[i]) \subseteq , _2$ and match$(, _1[N], expn(L[i]), \sigma)$
        then if match_steps$(, _1, , _2, L, N \Leftrightarrow 1, \sigma)$
                then return true
                fi
        fi;
        $\sigma := \sigma'$
    od;
    return false
```

Function `match_gamma`$(, _1, , _2, N, \sigma)$ checks whether each of the first N elements of $, _1$ matches at least one element of $, _2$ under substitution σ.

```
integer function match_gamma$(, _1, , _2, N, \sigma)$:
    Subst $\sigma'$;
    **if** $N = 0$ **then return** $true$ **fi**;
    **for** $i := 1$ **to** $| \ ,_2 |$
    **do** $\sigma' := \sigma$;
        **if** match$(,_1[N], ,_2[i], \sigma)$
        **then if** match_gamma$(,_1, ,_2, N \Leftrightarrow 1, \sigma)$
              **then return** $true$
              **fi**
        **fi**;
        $\sigma := \sigma'$
    **od**;
    **return** $false$
```

Function `match_premises`$(R, M, , , step, \sigma)$ checks whether, for every element $,', e'$ in sequence $R[M], \cdots, R[| R |]$, e' matches the expression part of $step$ and whether `extra_check`$(,', hyp(step), , , \sigma)$ holds.

```
**integer function** match_premises$(R, M, , , step, \sigma)$
    **subst** $\sigma' := \epsilon$;
    **if** match$(expn(R[M]), expn(step), \sigma)$ **and**
        extra_check$(hyp(R[M]), hyp(step), , , \sigma)$
    **then if** $M <| R |$
          **then for** $i := 1$ **to** $| L |$
                **do** $\sigma' := \sigma$;
                    **if** match_premises$(R, M + 1, , , step, \sigma)$
                    **then return** $true$
                    **fi**;
                $\sigma := \sigma'$
                **od**;
                **return** $false$
          **else return** $true$
    **else return** $false$
```

Function `extra_check`$(,_1, ,_2, , , \sigma)$ returns *true* iff the elements e_2 of $,_2$ for which no element of $,_1$ matches e_2 under substitution σ constitute a subset of $,$.

```
**integer function** extra_check$(,_1, ,_2, , , \sigma)$:
    $Gamma\ ,'_2 := ,_2$;
    **for** $i := 1$ **to** $\mid ,_2 \mid$
    **do** $j := 1$;
        **while** $j \leq \mid ,_1 \mid \wedge \neg$match$(,_1[j], ,_2[i], \sigma)$;
        **do** $j := j + 1$ **od**;
        **if** $j \leq \mid ,_1 \mid$ **then** $,'_2 := ,'_2 \Leftrightarrow ,_2[i]$ **fi**
    **od**;
    **return** $(,'_2 \subseteq ,)$
```

Finally, we present the main function `is_justification`$(R, , , e)$, whose description has already been given.

```
**integer function** is_justification$(R, , , e)$:
    **subst** $\sigma, \sigma' := \epsilon$;
    **if** match$(expn(R[1]), e, \sigma)$ **and**
        (match_steps$(hyp(R[1]), , , L, \mid hyp(R[1]) \mid, \sigma)$ **or**
        match_gamma$(hyp(R[1]), , , \mid hyp(R[1]) \mid, \sigma)$)
    **then if** $\mid R \mid > 1$
          **then for** $i := 1$ **to** $\mid L \mid$
                **do** $\sigma' := \sigma$;
                    **if** match_premises$(R, 2, , , L[i], \sigma)$
                    **then return** *true*
                    **fi**;
                $\sigma := \sigma'$
                **od**;
                **return** *false*
        else return $true$
    else return $false$
```

7.2 How Simplify Works

Snap's *Simplify* is based on *equational rewriting* [DJ89, HD83]. An *equational theory* is a set Σ of equations[2], that defines a congruence relation on the set of variables and operators, i.e., the smallest relation that contains the equations in Σ and that is closed under reflexivity, symmetry, transitivity, instantiation of free variables, and substitution of equals for equals. An equation $\mathcal{P} = \mathcal{P}'$ is in the equational theory of Σ, or is an *equational consequence* of Σ, if $\mathcal{P}$ is congruent to $\mathcal{P}'$.

7.2.1 Rewriting Systems

A *rewrite rule* is an operational view of an equation. To obtain a rewrite rule from an equation, one *orients* it in one of the two possible directions; for example, the equation $\mathcal{P} = \mathcal{P}'$ may be oriented into $\mathcal{P} \Leftrightarrow\!\!\rightarrow \mathcal{P}'$. A *rewriting system* is a set Ω of rewrite rules, that defines a **rewrite relation** — a binary relation on the set of formulae (usually written as $\leadsto_\Omega$). This may be defined operationally as follows: $\mathcal{A} \leadsto_\Omega \mathcal{A}'$ if there is a rule $\omega \in \Omega$ that **rewrites**[3] the formula $\mathcal{A}$ to $\mathcal{A}'$. The relation $\leadsto_\Omega^*$ is the reflexive transitive closure of $\leadsto_\Omega$.

Definitions:

1. A rewriting system is **Noetherian** (or *terminating*), if it allows no infinite sequence $\mathcal{A} \leadsto_\Omega \mathcal{A}' \leadsto_\Omega \ldots$. If such is the case, the last formula in the sequence is called the **terminal form** of $\mathcal{A}$.

[2]Informally, an equation $\mathcal{P} = \mathcal{P}'$ is another way of stating that $\mathcal{P} \Leftrightarrow \mathcal{P}'$ is a tautology.

[3]Formally, *rewriting* is defined as follows: Let $\mathcal{P} \Leftrightarrow\!\!\rightarrow \mathcal{P}'$ be a rewrite rule in Ω. If $\mathcal{B}$ is a subformula of $\mathcal{A}$, that is *matched* by $\mathcal{P}$, i.e., $\mathcal{B} \equiv \sigma(\mathcal{P})$ for some substitution σ, we obtain $\mathcal{A}'$ by replacing zero or more occurrences of $\mathcal{B}$ in $\mathcal{A}$ by a formula $\mathcal{B}' \equiv \sigma(\mathcal{P}')$.

2. A rewriting system is said to be **canonical** when every formula has a unique terminal form.

3. If a rewriting system is Noetherian and canonical, it is called **convergent**, and constitutes a **decision procedure** for the corresponding equational theory Σ.

Theorem 1 *The problem of deciding whether a set Ω of rewrite rules is Noetherian, is undecidable.*

Theorem 2 *If $\mathcal{B}$ is a subformula of $\mathcal{A}$, and we obtain $\mathcal{A}'$ by replacing zero or more occurrences of $\mathcal{B}$ in $\mathcal{A}$ by a wff $\mathcal{B}'$, then*

$$\textbf{if} \vdash B \Leftrightarrow B' \textbf{ and} \vdash A \textbf{ then} \vdash \mathcal{A}'.$$

Theorem 3 (Soundness Theorem for *Simplify***)** *Given a set of rewrite rules Ω, if for every rewrite rule $(\mathcal{A} \rightsquigarrow_{\Omega} \mathcal{A}') \in \Omega$, the formula $\mathcal{A} \Leftrightarrow \mathcal{A}'$ is valid, then the relation $\rightsquigarrow_{\Omega}^{*}$ preserves validity.*

Simplify works with a set Ω of rewrite rules, that may be provided by users, as long as they make sure that it satisfies `Theorem 3`. Users must also ensure that Ω is Noetherian; if not, *Simplify* could get into an infinite loop. Although convergent systems for propositional logic exist, they make *Simplify* painfully slow and are therefore best avoided.

When SNAP is invoked with the *Simplify* option, the system maintains terminal forms (in Ω) for each formula in the proof. When a proof step is entered, the proof annotation algorithm as outlined in section 7.1 is first invoked. If it fails to find a justification for the proof step, the algorithm is invoked again, this time with rewritten (terminal forms) of all formulae. If the algorithm returns a "yes", the step is considered valid. However, the attached justification in this case is "`by simplification`" and not the name of the associated inference rule.

Chapter 8

Case Studies

8.1 Event Tables

The A-7 requirements document [AFB+88] introduces tabular notations for specifying system requirements [Hen80]. In this model, system requirements are specified in terms of a set of *mode machines* [Fau89]. Each mode machine partitions the set of system states into equivalence classes called *modes*, and specifies transitions between modes, which are conditional upon the occurrence of *events*. Events are state changes in the system caused by the environment or by system actions. Events are denoted by $@T(Cond)$ and $@F(Cond)$, where $Cond$ is a predicate on system states and the event $@T(Cond)$ ($@F(Cond)$) signifies an instance of time when predicate $Cond$ becomes `true` (`false`). We present the requirements specification for a monitor which encapsulates data shared by two processes (from [AG93]).

The monitor is meant to mediate access to data shared by two processes. The specification has a single mode class with three modes — EMPTY (data not being accessed), INUSE1 (data being accessed by process1), and INUSE2 (data being accessed by process2). The initial mode of the system is EMPTY. Condition $Request1$ ($Request2$) indicates a request from process1 (process2) to access shared data. In the table, events are denoted by $@T$ and $@F$; symbol "`tt`" ("`ff`") signifies the truth (falsity) of a condition when an event occurs. The hyphen "`-`" indicates a *"don't care"* condition.

Current Mode	Request1	Request2	New Mode
EMPTY	@T	–	INUSE1
	–	@T	INUSE2
INUSE1	@F	ff	EMPTY
	@F	tt	INUSE2
INUSE2	ff	@F	EMPTY
	tt	@F	INUSE1

Figure 6: *Event table for monitor*

The system is required to satisfy certain invariant properties. Called *safety assertions* in [AG93], the system's correctness criterion asserts the invariance of the conjunction of the following three predicates:

$$\begin{aligned} \textsc{Empty} &\Rightarrow (\neg Request1 \wedge \neg Request2) \\ \textsc{InUse1} &\Rightarrow Request1 \\ \textsc{InUse2} &\Rightarrow Request2 \end{aligned}$$

8.1.1 A MELA Description of the System

We begin with the following *declarations*:

```
#define Request1 1
#define Request2 2

#define Empty  0
#define InUse1 1
#define InUse2 2

event atT(int), atF(int);
chan  Interface;
                                                         10
bool R1_val = 0, R2_val = 0;
```

We assign distinct integer values to inputs *Request1* and *Request2* of the system. In addition, (boolean) variables *R1_val* and *R2_val* record *current values* of these inputs. We expect events `atT(x)` and `atF(x)`, where x denotes either *Request1* or *Request2*, to occur on channel `Interface`.

We posit that the event table specifying the monitor describes relation *REQ* [PM91], described by the following MeLa program:

```
process class Monitor(chan in)
{ int state = Empty;

  loop
  {
  : state=Empty,   in?atT(Request1),                          state := InUse1;
  : state=Empty,                        in?atT(Request2),     state := InUse2;

  : state=InUse1, in?atF(Request1), R2_val = 0,        state := Empty;
  : state=InUse1, in?atF(Request1), R2_val = 1,        state := InUse2;

  : state=InUse2, R1_val = 0,       in?atF(Request2), state := Empty;
  : state=InUse2, R1_val = 1,       in?atF(Request2), state := InUse1;

  }

}
```

Here, the system is modelled as a MeLa process class with a `loop` construct having *six* single-step statements, each of which corresponds to one row in the tabular description.

8.1.2 A MeLa description of the Environment

The system we described is an example of an *embedded system*, i.e., the system is meant to work in an environment with certain (implicitly assumed) properties. For example, by examining the definition of events $@T(Cond)$ and $@F(Cond)$, one realises that it is impossible for an environment to cause two $@T$ events in succession (without an interleaving $@F$ event). These assumptions are made explicit when we describe the environment as a MeLa program:

```
process class Environment(chan out)
{
  loop
  {
  : R1_val = 0, out!atT(Request1), R1_val := 1;
  : R1_val = 1, out!atF(Request1), R1_val := 0;

  : R2_val = 0, out!atT(Request2), R2_val := 1;
  : R2_val = 1, out!atF(Request2), R2_val := 0;
  }
}
```

Finally, the system is a parallel composition of one instance each of process class `Monitor` and process class `Environment`, which communicate via channel `Interface`:

```
init
{ par
  {
  : Monitor mon(Interface)
  : Environment env(Interface)
  }
}
```

Our semantic model corresponds to the one given in [AG93]. Recently, one of the authors of this paper has been investigating the use of SMV [McM93] for analysing event tables[1]. The semantic models, however, are derived manually; the author proposes to investigate methods to automatically derive them. In our opinion, automatic derivation of SMV-style descriptions would be very difficult, because of SMV's predicate-like notation for expressing transition relations. We contend that automatic translation of tabular specifications to the MeLa notation should be more straightforward. To substantiate our claim, let us write the semantics of the above MeLa description in the SMV notation:

[1]J. M. Atlee, personal communication.

```
MODULE Monitor
VAR  Mode : {Empty, InUse1, InUse2};
     Request1, Request2 : {0, 1};

INIT ((Mode = Empty) & (Request1 = 0) & (Request2 = 0))
TRANS
  ((Mode=Empty)&(Request1=0)&(next(Mode)=InUse1)&(next(Request1)=1)&(next(Request2)=Request2)
   | (Mode=Empty)&(Request2=0)&(next(Mode)=InUse2)&(next(Request2)=1)&(next(Request1)=Request1)
   | (Mode=Empty)& !((Request1=0)&(next(Mode)=InUse1)&(next(Request1)=1)&(next(Request2)=Request2))
              & !((Request2=0)&(next(Mode)=InUse2)&(next(Request2)=1)&(next(Request1)=Request1))
              & (next(mode)=Empty)
   | (Mode=InUse1)&(Request1=1)&(Request2=0)&(next(Mode)=Empty)&(next(Request1)=0)&(next(Request2)=Request2)
   | (Mode=InUse1)&(Request1=1)&(Request2=1)&(next(Mode)=InUse2)&(next(Request1)=0)&(next(Request2)=Request2)
   | (Mode=InUse1)& !((Request1=1)&(Request2=0)&(next(Mode)=Empty)&(next(Request1)=0)&(next(Request2)=Request2))
              & !((Request1=1)&(Request2=1)&(next(Mode)=InUse2)&(next(Request1)=0)&(next(Request2)=Request2))
              & (next(mode)=InUse1)
   | (Mode=InUse2)&(Request1=0)&(Request2=1)&(next(Mode)=Empty)&(next(Request1)=Request1)&(next(Request2)=0)
   | (Mode=InUse2)&(Request1=1)&(Request2=1)&(next(Mode)=InUse1)&(next(Request1)=Request1)&(next(Request2)=0)
   | (Mode=InUse2)& !((Request1=0)&(Request2=1)&(next(Mode)=Empty)&(next(Request1)=Request1)&(next(Request2)=0))
              & !((Request1=1)&(Request2=1)&(next(Mode)=InUse1)&(next(Request1)=Request1)&(next(Request2)=0))
              & (next(mode)=InUse2)
  )

SPEC
  AG (Mode=Empty) -> ((Request1=0) & (Request2=0))
```

Adopting the notation proposed in this thesis results in a compact and less-cluttered semantic description than the one above. Our approach also has the advantage that "miracles", (transition relations which are equivalent to `ff`, which vacuously satisfy *all* requirements), may be spotted with relative ease e.g., by animation. Another advantage is that semantics expressed in this notation have an efficient operational interpretation, thereby permitting animation (abstract execution) and (state enumerative) model checking to be carried out on them. We present the semantics of the MeLa description above in terms of a transition system:

```
#define Empty  0
#define InUse1 1
#define InUse2 2

INIT
(state = Empty) /\ (R1_val = 0) /\ (R2_val = 0);

TRANS

(state=Empty /\ R1_val=0) -> R1_val := 1, state := InUse1;
(state=Empty /\ R2_val=0) -> R2_val := 1, state := InUse2;

(state=InUse1 /\(R1_val=1)/\ R2_Val=0) -> R1_val := 0, state := Empty;
(state=InUse1 /\(R1_val=1)/\ R2_Val=1) -> R1_val := 0, state := InUse2;

(state=InUse2 /\ R1_val=0 /\(R2_Val=1))-> R2_val := 0, state := Empty;
(state=InUse2 /\ R1_val=1 /\(R2_Val=1))-> R2_val := 0, state := InUse1;

INV
(state=Empty => (R1_val=0 /\ R2_val=0)) /\
(state=InUse1 => R1_val=1) /\
(state=InUse2 => R2_val=1);
```

We use the model checking rule INV^* of chapter 4 to establish the above invariant property. When performing model checking, we noticed that the following two transitions were never executed in all behaviours of the system.

```
(state=InUse1 /\(R1_val=1)/\ R2_Val=1) -> R1_val := 0, state := InUse2;
(state=InUse2 /\ R1_val=1 /\(R2_Val=1))-> R2_val := 0, state := InUse1;
```

When translated in terms of the original tabular description, this result implies that the *fourth* and *sixth* rows of the table are redundant — with the given semantics, it is impossible for those mode transitions to occur. An explanation for this is as follows: when the current mode is InUse1 (InUse2), the value of `Request2` (`Request1`) can never be *true*, as the semantic model "refuses" events, in the sense of [Hoa85], that are not explicitly stated in the table. Conversations with users of these tables, however, revealed that this was not the accepted (informal) meaning.

We therefore came to the conclusion that the semantic model presented in [AG93] is incorrect. To correct the problem, several solutions are possible. We propose the following "fix": events that are not explicitly handled in the table may occur — their occurrence will not change a system's current mode. In the above example, the table has four additional (implicit) transitions, yielding the following MeLa description for process class `Monitor`:

```
process class Monitor(chan in)
{ int state = Empty;
  loop
  {
  : state=Empty,   in?atT(Request1),                            state := InUse1;
  : state=Empty,                              in?atT(Request2), state := InUse2;

  : state=InUse1, in?atF(Request1), R2_val = 0,       state := Empty;
  : state=InUse1, in?atF(Request1), R2_val = 1,       state := InUse2;
  : state=InUse1,                         in?atT(Request2), state := InUse1;
  : state=InUse1,                         in?atF(Request2), state := InUse1;

  : state=InUse2, R1_val = 0,       in?atF(Request2), state := Empty;
  : state=InUse2, R1_val = 1,       in?atF(Request2), state := InUse1;
  : state=InUse2, in?atT(Request1),                       state := InUse2;
  : state=InUse2, in?atF(Request1),                       state := InUse2;
  }
}
```

Theorem Proving: We also established the invariant *directly* by using rule **INV** of [MP91b]. This may be done by proving the following theorems using Snap:

```
#define Empty  0
#define InUse1 1
#define InUse2 2

#define q (state=Empty => (R1_val=0 /\ R2_val=0))

#define Init (state=Empty/\R1_val=0/\R2_val=0)

#define Phi1 (state=Empty => (R1_val=0 /\ R2_val=0))
```

```
#define Phi2 (state=InUse1 => R1_val=1)
#define Phi3 (state=InUse2 => R2_val=1)
#define Phi  (Phi1 /\ Phi2 /\ Phi3)

prove |- Init => Phi;
prove {Phi} state=Empty/\R2_val=0 -> R2_val:=1,state:=InUse2{Phi};

prove {Phi} state=InUse1/\R1_val=1/\R2_Val=0 -> R1_val:=0,state:=Empty  {Phi};
prove {Phi} state=InUse1/\R1_val=1/\R2_Val=1 -> R1_val:=0,state:=InUse2 {Phi};

prove {Phi} state=InUse2/\R1_val=0/\R2_Val=1 -> R2_val:=0,state:=Empty  {Phi};
prove {Phi} state=InUse2/\R1_val=1/\R2_Val=1 -> R2_val:=0,state:=InUse1 {Phi};
prove |- Phi => q;
quit;
```

8.2 The DQDB Protocol

The distributed queue dual bus (DQDB) protocol [HCM92] has a dual-bus topology; each bus supports unidirectional communications in opposite directions. We depict this topology in figure 8.2.

```
 ________                "data bus"             __________
|        | udb    _______  idb    _______  ddb |          |
|Upstream|----->| Node1 |----->| Node2 |----->|Downstream|
|        |<-----|_______|<-----|_______|<-----|          |
|________| urb             irb            drb |__________|
                       "request bus"
```

DQDB resembles a slotted ring with free access, where stations transmit in every empty slot if they have data. The protocol uses the channel in the opposite direction from which data is sent to reserve slots for stations that are farther from the head-end (upstream end) of the bus. In addition, DQDB has three associated priority levels, which we shall not consider for the purpose of this discussion. In this thesis, we only model the protocol for a single priority level. To simplify our task, we exploit symmetry in the protocol and only model transfer of data in one direction (with reservations flowing in the other direction).

It has been observed [HCM92] that the DQDB reservation process is imperfect. If the span of a DQDB link is long enough to allow many slots to be in transit between any two nodes, the nodes may have an inconsistent view of the reservation process. When this happens, the link bandwidth can be unevenly divided among nodes; in the worst case, some nodes may be allocated no bandwidth at all (this is sometimes called *starvation*). The following is an (informal) description of (a single priority section of) the DQDB protocol:

> *This section has a local FIFO queue to store priority-p data segments generated by local users while these segments wait for the data inserter (DI) to find the appropriate empty slots for them on the data bus. The data inserter operates on one local data segment at a time; once the local FIFO queue forwards a segment to the data inserter, the local FIFO queue may not forward another segment until the data inserter has written the current segment onto the data bus. When the data inserter takes a segment from the local FIFO queue, first it orders the request inserter (RI) to send a priority-p request on the request bus. Then the data inserter determines the appropriate empty slot for the local segment by inserting the segment into the data inserter's transmit queue (TQ). All the other elements of this queue are requests of priority p or greater from downstream nodes. (The data inserter ignores all requests of priority less than p.) The transmit queue orders its elements according to their priority level, with elements of equal priority ordered by the times they arrive at the data inserter. The data inserter serves its transmit queue whenever an empty slot comes in on the data bus. If the element at the head of the queue is a request, then the data inserter lets the empty slot pass. If the head element is the local data segment, then the busy bit is set and the segment is transmitted in that slot. The transmit queue is implemented with two counters, called the request counter and the countdown counter. When there is no local data segment in the queue, the request counter keeps track of the number*

of unserved reservations from downstream nodes in the transmit queue. When the data inserter accepts a local data segment, the request counter value is moved to the countdown counter, which counts the number of reservations that are ahead of the local data segment in the transmit queue, and the request counter is then used to count reservations behind the local data segment. The request inserter sends one reservation of priority p for each data segment taken by the data inserter from the local FIFO queue. Since the incoming priority-p request bits may have been set already by downstream nodes, the request inserter sometimes needs to queue the internally generated reservations until vacant request bits arrive. Thus, it is possible for a data segment to be transmitted before its reservation is sent.

8.2.1 Preliminaries

To start with, we declare the following events and channels:

```
event not_busy(), busy(), no_req(), req();
chan udb, idb, ddb, urb, irb, drb;
```

We model slots as MeLa events. Each slot in the direction of data transfer is either `busy()` or `not_busy()`, indicating the presence or absence of data. Similarly, slots moving in the reverse direction either have requests (`req()`) or are free (`no_req()`). We describe the protocol as having two `StreamHead`s, one at the upstream end and another at the downstream end. There are *six* channels, with names as in figure 8.2.

The StreamHeads are described as a MeLa process class definition:

```
process class StreamHead(chan in, chan out, event gen)
{ event sink;
  loop {
  : in?sink
  : out!gen
  }
}
```

8.2.2 The DQDB Protocol in MeLa

We now present the MeLa description of a DQDB node:

```
process class Node(chan db_in, chan db_out, chan rb_in, chan rb_out)
{ bool local_data = ff;
  int  send_req = 0;

  /* Components of Transmit Queue (TQ) */
  int request = 0, countdown = 0;

  loop {
  /* Transitions of Data Inserter (DI) */
T1   : ~local_data, local_data := tt, send_req := send_req + 1,
       countdown := request, request := 0
T2   : db_in?busy(), db_out!busy()
T3   : db_in?not_busy(), ~local_data, request = 0,
       db_out!not_busy()
T4   : db_in?not_busy(), ~local_data, request # 0,
       db_out!not_busy(),
       request := request - 1         /* <- only my guess! */
T5   : db_in?not_busy(), local_data, countdown # 0,
       db_out!not_busy(), countdown := coundown - 1
T6   : db_in?not_busy(), local_data, countdown = 0,
       db_out!busy(), local_data := ff

    /* Transitions of Request Inserter (RI) */
T7   : rb_in?req(), rb_out!req(), request := request + 1
T8   : rb_in?no_req(), send_req = 0, rb_out!no_req()
T9   : rb_in?no_req(), send_req # 0, rb_out!req(), send_req := send_req - 1
  }
}
```

The MeLa description above closely matches the informal description of the protocol we presented earlier. Let us now examine the case where the spans of links are short enough not to allow slots to be in transit between adjacent nodes. For this case, the behaviour of each link can be expressed as a single MeLa channel. The following `init` process instantiates the necessary processes for this case:

```
init {
  par {
  : StreamHead Upstream(urb, udb, not_busy())
  : StreamHead Downstream(ddb, drb, no_req())
  : Node Node1(udb, idb, irb, urb)
  : Node Node2(idb, ddb, drb, irb)
  }
}
```

It can be shown by model checking that the behaviour of the above protocol description includes execution sequences in which the downstream node never encounters free slots, thereby leading to starvation. To see this, consider the claim:

$$\Box(\texttt{Node2.local\_data} \Rightarrow \Diamond\neg\texttt{Node2.local\_data})$$

which guarantees that the downstream node (Node 2) will have its local variable `local_data` reset to "false", following each state in which it is set to "true", i.e., if the node has local data to send, then eventually it will get sent. In the following, we describe a scenario in which the above property is violated. We present the scenario as a sequence of two actions, which may be repeated an indefinite number of times. Transition names refer to labels in the description of process class *Node*.

- *Node1* executes transition *T1*, which sets its local variable `local_data` to "true". This means that *Node1* has data to send. The node's local variables `countdown` and `request` both remain 0.

- Node *Upstream* puts out a `not_busy()` slot on channnel `udb`. *Node1* executes transition *T6*, which receives the `not_busy()` slot from channel `udb` and sends locally generated data (i.e., a `busy()` slot) on channel `idb`. Additionally, *Node1* sets its local variable `local_data` to "false". *Node 2* executes transition *T2*, which receives the `busy()` slot on channel `idb` and sends it on channel `ddb`. Node *Downstream* "consumes" the `busy()` slot by receiving it on channel `idb`.

Repeating the above two actions will result in *Node2* receiving a series of `busy()` slots, thereby only executing transition *T2*.

Note that in this scenario nothing is sent on the request bus. The "practical interpretation" of this scenario is as follows (this interpretation is strictly outside the formal model):

> If the span of the DQDB link `irb` is long enough to allow an arbitrary number of slots to be in transit between *Node1* and *Node2*, then *Node2* may receive an arbitrarily long series of `busy()` slots on link `idb`, imposing an arbitrary delay in sending its local data.

We call this the *starvation problem.* In the next section, we shall examine a proposal that is intended to rectify this situation, and verify that it corrects the problem.

8.2.3 Bandwidth Balancing

In [HCM92], the authors describe *bandwidth balancing*, a proposal to correct the starvation problem. The authors present their solution by providing the following description (in informal prose):

> *One way to implement this scheme is to add a bandwidth balancing counter (BC) to the data inserter; the counter counts local data segments transmitted on the bus. After M segments have been transmitted, the bandwidth balancing counter resets itself to zero and generates a signal that the data inserter treats exactly like a request from a downstream node. This (artificial) request causes the data inserter to let a slot go unallocated (the request inserter is not aware of this signal; the node therefore does not send a request upstream which corresponds to the extra idle slot it sends downstream).*

With this modification, the DQDB protocol does not violate the liveness claim we presented in the previous section. To see this, we model the modified protocol in MeLa, and establish the property for the modified program.

The following is a modified MeLa description of a node which includes a balancing counter:

```
process class Node(chan db_in, chan db_out, chan rb_in, chan rb_out)
{ bool local_data = ff;
  int  send_req = 0;

  /* Components of Transmit Queue (TQ) */
  int request = 0, countdown = 0, BC = 0;

  loop {
  /* Transitions of Data Inserter (DI) */
T1  : ~local_data, local_data := tt, send_req := send_req + 1,
      countdown := request, request := 0
T2  : db_in?busy(), db_out!busy()
T3  : db_in?not_busy(), ~local_data, request = 0,
      db_out!not_busy()
T4  : db_in?not_busy(), ~local_data, request # 0,
      db_out!not_busy(),
      request := request - 1        /* <- only my guess! */
T5  : db_in?not_busy(), local_data, countdown # 0,
      db_out!not_busy(), countdown := coundown - 1
T6  : db_in?not_busy(), local_data, countdown = 0, BC < M-1,
      db_out!busy(), local_data := ff, BC := BC + 1
T7  : db_in?not_busy(), local_data, countdown = 0, BC >= M-1,
      db_out!busy(), local_data := ff, BC := 0, request := request + 1

  /* Transitions of Request Inserter (RI) */
T8  : rb_in?req(), rb_out!req(), request := request + 1
T9  : rb_in?no_req(), send_req = 0, rb_out!no_req()
T10 : rb_in?no_req(), send_req # 0, rb_out!req(), send_req := send_req - 1
  }
}
```

For the modified system, it can be shown that its behaviour does not include execution sequences which lead to starvation of the downstream node. We prove this by establishing that the upstream node sends a `not_busy()` event to the downstream node, infinitely often. To establish this, it is sufficient to show that transition *T5* will be taken infinitely often.

We first prove the following lemma:

$$L1: \quad \Box((\texttt{Node1.local\_data} \ \wedge (\texttt{Node1.request} = 0)) \Rightarrow \Diamond \ \texttt{Node1.request} = 1)$$

We do this by establishing the following chain of argument:

$$\begin{array}{rl}
1.1: & \Box((\texttt{Node1.local\_data} \wedge (\texttt{Node1.BC} = 0)) \Rightarrow \Diamond \ \texttt{Node1.BC} = 1) \\
1.2: & \Box((\texttt{Node1.local\_data} \wedge (\texttt{Node1.BC} = 1)) \Rightarrow \Diamond \ \texttt{Node1.BC} = 2) \\
\vdots & \\
1.k: & \Box((\texttt{Node1.local\_data} \wedge (\texttt{Node1.BC} = M \Leftrightarrow 2)) \Rightarrow \Diamond \ \texttt{Node1.BC} = M \Leftrightarrow 1) \\
1.M: & \Box((\texttt{Node1.local\_data} \wedge (\texttt{Node1.BC} = M \Leftrightarrow 1) \wedge (\texttt{Node1.request} = 0)) \\
 & \Rightarrow \Diamond \ \texttt{Node1.request} = 1)
\end{array}$$

We establish lemma *L1* by showing that the bandwidth balancing counter (`BC`) of Node 1 is incremented each time it has local data to send. Finally, when the value of `BC` reaches the value $M \Leftrightarrow 1$, the local variable `request` gets incremented to 1.

Next, we prove the following lemma:

$$L2: \quad \Box((\texttt{Node1.request} = 1) \Rightarrow \Diamond \ \texttt{Node1.countdown} = 1)$$

which states that the local variable `countdown` will have the value 1 in a state following each state in which local variable `request` has value 1. This is because transition *T1* is enabled infinitely often, whose effect is to assign the current value of variable `request` to `countdown`. We have therefore shown that transition *T5* will eventually be enabled, thereby establishing that the upstream node (Node 1) will send the event `not_busy()` to the downstream node (Node 1). Thus, we have proved that there is no starvation of the downstream node.

Chapter 9

Conclusion and Future Work

In this thesis, we present a method, supported by tools, to verify system descriptions expressed in a programming language-like notation. Our method may be applied to a wide range of problems, including, but not restricted to the verification of parallel algorithms, embedded systems, distributed applications, communications protocols, and computer hardware. Our method, however, is restricted to the verification of *logical* properties, and does not address qualitative issues such as throughput or delay.

The problem of establishing logical properties of a design is known as the "correctness problem". To start with, the design has to be expressed in an unambiguous notation, with a well-defined formal semantics. We have designed a programming language-like notation called MeLa for this purpose, for which we provide a formal semantics based on the notion of transition systems. For a *formal system description* expressed in MeLa, the designer expresses desired logical properties, or requirements, which the system description is supposed to satisfy. In our approach, these logical properties are written as predicates in a formal logic. The correctness problem is therefore reduced to the problem of establishing that the requirements criteria hold for a MeLa program.

We propose a verification method, TOP, to analyse a MeLa system description and show that it satisfies its requirements. Our method utilises theorem proving as well as model checking verification approaches, and usefully combines the two techniques. In our method, we augment theorem proving methods with model-checking algorithms, thereby permitting the two techniques to be used together — proof rules are used to decompose a large problem into smaller sub-problems, each of which may be automatically verified by model checking. We do this by (re-)interpreting well-known proof rules for LTL in the context of model checking. An interesting question to ask is "But what are the *general* rules for interpreting *arbitrary* proof rules as model checking rules?" An answer to this question would be a generalisation of our work, and an interesting area of research.

In this thesis, we also propose a new human-computer interface for theorem proving systems. We have implemented a system (SNAP) which has this interface. Our system has been designed for efficiency and ease-of-use. SNAP allows proofs to be carried out at a desired level of abstraction, and its interface permits machine assisted proofs to be carried out in a style that is close to "natural" proofs. SNAP users may increase the level of abstraction of proofs by adding new theorems, inference rules, and certain meta rules to the system's rule-base, even when another proof is in progress. Users from specific application areas may develop and maintain libraries of theorems, lemmata, and inference rules, which may be used by others, including novices, without detailed knowledge of their exact form, or their associated names — an intuitive understanding is sufficient to use them.

We also present two case studies which used our verification method. In the first study, system requirements presented in a tabular notation were described in MELA, and model checking and theorem proving methods proposed in this thesis were used to verify certain "safety assertions" for a tabular specification. In the second study, we analysed liveness violations in a (published) communications protocol standard, and verified that suggested changes to the standard have fixed the problem.

In order to turn this thesis work into a system that could help real programmers solve real problems, there a number of things that remain to be done. To start with, we have to integrate our model checking algorithms into the theorem prover SNAP. Additionally, we have to enhance the level of automation offered by SNAP. We have established proof of concept by integrating term rewriting and traditional proof rules within SNAP. To build upon this idea, we will have to implement decision procedures for tautology checking, and decidable fragments of arithmetic. To be able to verify more complex protocols would entail adding theories for sequences and queues. Additionally, we will have to include decision procedures for these theories.

To use our method for the verification of complex SCR specifications, SNAP's term rewriting subsystem has to be extended to simplify SCR event expressions, and to perform constant arithmetic. To be able to reason about multiple SCR tables, the language MELA has to be extended. At the moment, we have the restriction that in a single-step statement, we cannot use the newly assigned value of a variable in assignment expressions of other variables. This restriction has to be relaxed, as SCR event expressions may refer to both the old and the new values of variables. To avoid circularity, we will have to impose a partial order on the evaluation of such expressions as in [HJL95]. In [Bha96], we report our preliminary work in this direction. The notation proposed in [Bha96] is a natural extension of the notion of multiple assignments.

Appendix A

Proof of Dekker's Algorithm

```
#include "Assumptions.sn"
#define I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define Inv1 ((pc1 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define Inv2 ((pc2 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define Turn (turn = 1 || turn =2)
#define Inv (I1&&J1&&I2&&J2&&Inv1&&Inv2&&Loc1&&Loc2&&Turn)

#define req ~(pc1=L5 && pc2=M5)

#define T1I1 ((L1 = L1 || L1 = L2 || L1 = L3 || L1 = L5) => 0 = 0)
#define T1J1 ((L1 = L0 || L1 = L4) => 0 = 1)
#define T1I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T1J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T1Inv1 ((L1 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T1Inv2 ((pc2 = M5) =>(L1 = L1 || L1 = L2 || 0 = 1 || turn = 2))
#define T1Loc1 (L1=L0||L1=L1||L1=L2||L1=L3||L1=L4||L1=L5)
#define T1Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T1Turn (turn = 1 || turn =2)
#define T1Inv \
(T1I1&&T1J1&&T1I2&&T1J2&&T1Inv1&&T1Inv2&&T1Loc1&&T1Loc2&&T1Turn)
```

```
#define T2I1 ((L2 = L1 || L2 = L2 || L2 = L3 || L2 = L5) => c1 = 0)
#define T2J1 ((L2 = L0 || L2 = L4) => c1 = 1)
#define T2I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T2J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T2Inv1 ((L2 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T2Inv2 ((pc2 = M5) =>(L2 = L1 || L2 = L2 || c1 = 1 || turn = 2))
#define T2Loc1 (L2=L0||L2=L1||L2=L2||L2=L3||L2=L4||L2=L5)
#define T2Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T2Turn (turn = 1 || turn =2)
#define T2Inv \
(T2I1&&T2J1&&T2I2&&T2J2&&T2Inv1&&T2Inv2&&T2Loc1&&T2Loc2&&T2Turn)

#define T3I1 ((L5 = L1 || L5 = L2 || L5 = L3 || L5 = L5) => c1 = 0)
#define T3J1 ((L5 = L0 || L5 = L4) => c1 = 1)
#define T3I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T3J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T3Inv1 ((L5 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T3Inv2 ((pc2 = M5) =>(L5 = L1 || L5 = L2 || c1 = 1 || turn = 2))
#define T3Loc1 (L5=L0||L5=L1||L5=L2||L5=L3||L5=L4||L5=L5)
#define T3Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T3Turn (turn = 1 || turn =2)
#define T3Inv \
(T3I1&&T3J1&&T3I2&&T3J2&&T3Inv1&&T3Inv2&&T3Loc1&&T3Loc2&&T3Turn)

#define T4I1 ((L1 = L1 || L1 = L2 || L1 = L3 || L1 = L5) => c1 = 0)
#define T4J1 ((L1 = L0 || L1 = L4) => c1 = 1)
#define T4I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T4J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T4Inv1 ((L1 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T4Inv2 ((pc2 = M5) =>(L1 = L1 || L1 = L2 || c1 = 1 || turn = 2))
#define T4Loc1 (L1=L0||L1=L1||L1=L2||L1=L3||L1=L4||L1=L5)
#define T4Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T4Turn (turn = 1 || turn =2)
#define T4Inv \
(T4I1&&T4J1&&T4I2&&T4J2&&T4Inv1&&T4Inv2&&T4Loc1&&T4Loc2&&T4Turn)

#define T5I1 ((L3 = L1 || L3 = L2 || L3 = L3 || L3 = L5) => c1 = 0)
#define T5J1 ((L3 = L0 || L3 = L4) => c1 = 1)
#define T5I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T5J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
```

```
#define T5Inv1 ((L3 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T5Inv2 ((pc2 = M5) =>(L3 = L1 || L3 = L2 || c1 = 1 || turn = 2))
#define T5Loc1 (L3=L0||L3=L1||L3=L2||L3=L3||L3=L4||L3=L5)
#define T5Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T5Turn (turn = 1 || turn =2)
#define T5Inv \
(T5I1&&T5J1&&T5I2&&T5J2&&T5Inv1&&T5Inv2&&T5Loc1&&T5Loc2&&T5Turn)

#define T6I1 ((L4 = L1 || L4 = L2 || L4 = L3 || L4 = L5) => 1 = 0)
#define T6J1 ((L4 = L0 || L4 = L4) => 1 = 1)
#define T6I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T6J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T6Inv1 ((L4 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T6Inv2 ((pc2 = M5) =>(L4 = L1 || L4 = L2 || 1 = 1 || turn = 2))
#define T6Loc1 (L4=L0||L4=L1||L4=L2||L4=L3||L4=L4||L4=L5)
#define T6Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T6Turn (turn = 1 || turn =2)
#define T6Inv \
(T6I1&&T6J1&&T6I2&&T6J2&&T6Inv1&&T6Inv2&&T6Loc1&&T6Loc2&&T6Turn)

#define T7I1 ((L4 = L1 || L4 = L2 || L4 = L3 || L4 = L5) => c1 = 0)
#define T7J1 ((L4 = L0 || L4 = L4) => c1 = 1)
#define T7I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T7J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T7Inv1 ((L4 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T7Inv2 ((pc2 = M5) =>(L4 = L1 || L4 = L2 || c1 = 1 || turn = 2))
#define T7Loc1 (L4=L0||L4=L1||L4=L2||L4=L3||L4=L4||L4=L5)
#define T7Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T7Turn (turn = 1 || turn =2)
#define T7Inv \
(T7I1&&T7J1&&T7I2&&T7J2&&T7Inv1&&T7Inv2&&T7Loc1&&T7Loc2&&T7Turn)

#define T8I1 ((L0 = L1 || L0 = L2 || L0 = L3 || L0 = L5) => c1 = 0)
#define T8J1 ((L0 = L0 || L0 = L4) => c1 = 1)
#define T8I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T8J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T8Inv1 ((L0 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1))
#define T8Inv2 ((pc2 = M5) =>(L0 = L1 || L0 = L2 || c1 = 1 || turn = 2))
#define T8Loc1 (L0=L0||L0=L1||L0=L2||L0=L3||L0=L4||L0=L5)
#define T8Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
```

```
#define T8Turn (turn = 1 || turn =2)
#define T8Inv \
(T8I1&&T8J1&&T8I2&&T8J2&&T8Inv1&&T8Inv2&&T8Loc1&&T8Loc2&&T8Turn)

#define T9I1 ((L0 = L1 || L0 = L2 || L0 = L3 || L0 = L5) => 1 = 0)
#define T9J1 ((L0 = L0 || L0 = L4) => 1 = 1)
#define T9I2 ((pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5) => c2 = 0)
#define T9J2 ((pc2 = M0 || pc2 = M4) => c2 = 1)
#define T9Inv1 ((L0 = L5) =>(pc2 = M1 || pc2 = M2 || c2 = 1 || 2 = 1))
#define T9Inv2 ((pc2 = M5) =>(L0 = L1 || L0 = L2 || 1 = 1 || 2 = 2))
#define T9Loc1 (L0=L0||L0=L1||L0=L2||L0=L3||L0=L4||L0=L5)
#define T9Loc2 (pc2=M0||pc2=M1||pc2=M2||pc2=M3||pc2=M4||pc2=M5)
#define T9Turn (2 = 1 || 2 =2)
#define T9Inv \
(T9I1&&T9J1&&T9I2&&T9J2&&T9Inv1&&T9Inv2&&T9Loc1&&T9Loc2&&T9Turn)

#define T10I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T10J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T10I2 ((M1 = M1 || M1 = M2 || M1 = M3 || M1 = M5) => 0 = 0)
#define T10J2 ((M1 = M0 || M1 = M4) => 0 = 1)
#define T10Inv1 ((pc1 = L5) =>(M1 = M1 || M1 = M2 || 0 = 1 || turn = 1))
#define T10Inv2 ((M1 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T10Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T10Loc2 (M1=M0||M1=M1||M1=M2||M1=M3||M1=M4||M1=M5)
#define T10Turn (turn = 1 || turn =2)
#define T10Inv \
(T10I1&&T10J1&&T10I2&&T10J2&&T10Inv1&&T10Inv2&&T10Loc1&&T10Loc2&&T10Turn)

#define T11I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T11J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T11I2 ((M2 = M1 || M2 = M2 || M2 = M3 || M2 = M5) => c2 = 0)
#define T11J2 ((M2 = M0 || M2 = M4) => c2 = 1)
#define T11Inv1 ((pc1 = L5) =>(M2 = M1 || M2 = M2 || c2 = 1 || turn = 1))
#define T11Inv2 ((M2 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T11Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T11Loc2 (M2=M0||M2=M1||M2=M2||M2=M3||M2=M4||M2=M5)
#define T11Turn (turn = 1 || turn =2)
#define T11Inv \
(T11I1&&T11J1&&T11I2&&T11J2&&T11Inv1&&T11Inv2&&T11Loc1&&T11Loc2&&T11Turn)
```

```
#define T12I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T12J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T12I2 ((M5 = M1 || M5 = M2 || M5 = M3 || M5 = M5) => c2 = 0)
#define T12J2 ((M5 = M0 || M5 = M4) => c2 = 1)
#define T12Inv1 ((pc1 = L5) =>(M5 = M1 || M5 = M2 || c2 = 1 || turn = 1))
#define T12Inv2 ((M5 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T12Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T12Loc2 (M5=M0||M5=M1||M5=M2||M5=M3||M5=M4||M5=M5)
#define T12Turn (turn = 1 || turn =2)
#define T12Inv \
(T12I1&&T12J1&&T12I2&&T12J2&&T12Inv1&&T12Inv2&&T12Loc1&&T12Loc2&&T12Turn)

#define T13I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T13J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T13I2 ((M1 = M1 || M1 = M2 || M1 = M3 || M1 = M5) => c2 = 0)
#define T13J2 ((M1 = M0 || M1 = M4) => c2 = 1)
#define T13Inv1 ((pc1 = L5) =>(M1 = M1 || M1 = M2 || c2 = 1 || turn = 1))
#define T13Inv2 ((M1 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T13Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T13Loc2 (M1=M0||M1=M1||M1=M2||M1=M3||M1=M4||M1=M5)
#define T13Turn (turn = 1 || turn =2)
#define T13Inv \
(T13I1&&T13J1&&T13I2&&T13J2&&T13Inv1&&T13Inv2&&T13Loc1&&T13Loc2&&T13Turn)

#define T14I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T14J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T14I2 ((M3 = M1 || M3 = M2 || M3 = M3 || M3 = M5) => c2 = 0)
#define T14J2 ((M3 = M0 || M3 = M4) => c2 = 1)
#define T14Inv1 ((pc1 = L5) =>(M3 = M1 || M3 = M2 || c2 = 1 || turn = 1))
#define T14Inv2 ((M3 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T14Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T14Loc2 (M3=M0||M3=M1||M3=M2||M3=M3||M3=M4||M3=M5)
#define T14Turn (turn = 1 || turn =2)
#define T14Inv \
(T14I1&&T14J1&&T14I2&&T14J2&&T14Inv1&&T14Inv2&&T14Loc1&&T14Loc2&&T14Turn)

#define T15I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T15J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T15I2 ((M4 = M1 || M4 = M2 || M4 = M3 || M4 = M5) => 1 = 0)
#define T15J2 ((M4 = M0 || M4 = M4) => 1 = 1)
```

```
#define T15Inv1 ((pc1 = L5) =>(M4 = M1 || M4 = M2 || 1 = 1 || turn = 1))
#define T15Inv2 ((M4 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T15Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T15Loc2 (M4=M0||M4=M1||M4=M2||M4=M3||M4=M4||M4=M5)
#define T15Turn (turn = 1 || turn =2)
#define T15Inv \
(T15I1&&T15J1&&T15I2&&T15J2&&T15Inv1&&T15Inv2&&T15Loc1&&T15Loc2&&T15Turn)

#define T16I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T16J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T16I2 ((M4 = M1 || M4 = M2 || M4 = M3 || M4 = M5) => c2 = 0)
#define T16J2 ((M4 = M0 || M4 = M4) => c2 = 1)
#define T16Inv1 ((pc1 = L5) =>(M4 = M1 || M4 = M2 || c2 = 1 || turn = 1))
#define T16Inv2 ((M4 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T16Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T16Loc2 (M4=M0||M4=M1||M4=M2||M4=M3||M4=M4||M4=M5)
#define T16Turn (turn = 1 || turn =2)
#define T16Inv \
(T16I1&&T16J1&&T16I2&&T16J2&&T16Inv1&&T16Inv2&&T16Loc1&&T16Loc2&&T16Turn)

#define T17I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T17J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T17I2 ((M0 = M1 || M0 = M2 || M0 = M3 || M0 = M5) => c2 = 0)
#define T17J2 ((M0 = M0 || M0 = M4) => c2 = 1)
#define T17Inv1 ((pc1 = L5) =>(M0 = M1 || M0 = M2 || c2 = 1 || turn = 1))
#define T17Inv2 ((M0 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2))
#define T17Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T17Loc2 (M0=M0||M0=M1||M0=M2||M0=M3||M0=M4||M0=M5)
#define T17Turn (turn = 1 || turn =2)
#define T17Inv \
(T17I1&&T17J1&&T17I2&&T17J2&&T17Inv1&&T17Inv2&&T17Loc1&&T17Loc2&&T17Turn)

#define T18I1 ((pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5) => c1 = 0)
#define T18J1 ((pc1 = L0 || pc1 = L4) => c1 = 1)
#define T18I2 ((M0 = M1 || M0 = M2 || M0 = M3 || M0 = M5) => 1 = 0)
#define T18J2 ((M0 = M0 || M0 = M4) => 1 = 1)
#define T18Inv1 ((pc1 = L5) =>(M0 = M1 || M0 = M2 || 1 = 1 || 1 = 1))
#define T18Inv2 ((M0 = M5) =>(pc1 = L1 || pc1 = L2 || c1 = 1 || 1 = 2))
#define T18Loc1 (pc1=L0||pc1=L1||pc1=L2||pc1=L3||pc1=L4||pc1=L5)
#define T18Loc2 (M0=M0||M0=M1||M0=M2||M0=M3||M0=M4||M0=M5)
```

```
#define T18Turn (1 = 1 || 1 =2)
#define T18Inv \
(T18I1&&T18J1&&T18I2&&T18J2&&T18Inv1&&T18Inv2&&T18Loc1&&T18Loc2&&T18Turn)

#define Init1 (pc1=L0 && pc2=M0)
#define Init2 (c1=1 && c2=1)
#define InitTurn (turn=1)
#define Init (Init1 && Init2 && InitTurn)

prove |- Init => Inv;
assume Init;
Init1;
pc1 = L0;
~(pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5);
I1;
Init2;
c1 = 1;
J1;
pc2 = M0;
~(pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5);
I2;
c2 = 1;
J2;
~(pc1 = L5);
Inv1;
~(pc2 = M5);
Inv2;
Loc1;
Loc2;
turn = 1;
Turn;
Inv;
|- Init => Inv;
qed T1;

prove |- {Inv} pc1=L0 -> pc1:=L1, c1:=0 {Inv};
assume Inv && pc1=L0;
0 = 0;
T1I1;
~(L1 = L0 || L1 = L4);
```

```
T1J1;
Inv;
T1I2;
T1J2;
~(L1=L5);
T1Inv1;
T1I1 && T1J1 && T1I2 && T1J2 && T1Inv1;
L1 = L1;
L1 = L1 || L1 = L2 || 0 = 1 || turn = 2;
T1Inv2;
T1Loc1;
T1Loc2;
T1Turn;
T1Inv;
|- {Inv} pc1=L0 -> pc1:=L1, c1:=0 {Inv};
qed T2;

prove Inv => req;
assume Inv;
assume (pc1=L5 && pc2=M5);
pc2=M5;
pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5;
I2;
c2=0;
Inv1;
pc1=L5;
pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1;
~(pc2 = M1);
~(pc2 = M2);
~(c2 = 1);
turn = 1;
Inv2;
pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2;
~(pc1=L1);
~(pc1=L2);
pc1 = L1 || pc1 = L2 || pc1 = L3 || pc1 = L5;
I1;
c1 = 0;
~(c1 = 1);
turn = 2;
turn = 1 && turn = 2;
ff;
Inv |- ~(pc1=L5 && pc2=M5);
|- Inv => req;
qed Requirement;

prove |- {Inv} pc1=L1 && c1=0 -> pc1:=L2 {Inv};
assume Inv && (pc1=L1 && c1=0);
c1=0;
T2I1;
pc1=L1;
~(L2 = L0 || L2 = L4);
T2J1;
T2I2;
T2J2;
~(L2=L5);
T2Inv1;
L2=L2;
L2 = L1 || L2 = L2 || c1 = 1 || turn = 2;
T2Inv2;
T2Loc1;
T2Loc2;
T2Turn;
T2Inv;
|- {Inv} pc1=L1 && c1=0 -> pc1:=L2 {Inv};
qed T2;

prove |- {Inv} pc1=L1&&~c2=0->pc1:=L5{Inv};
assume Inv &&(pc1=L1&&~c2=0);
I1;
pc1=L1;
pc1=L1||pc1=L2||pc1=L3||pc1=L5;
c1=0;
T3I1;
~(L5=L0||L5=L4);
T3J1;
T3I2;
T3J2;
~(c2=0);
I2;
```

```
~(pc2=M1||pc2=M2||pc2=M3||pc2=M5);
~(pc2=M1)&&~(pc2=M2)&&~(pc2=M3)&&~(pc2=M5);
Loc2;
pc2=M0||pc2=M4;
c2=1;
(pc2 = M1 || pc2 = M2 || c2 = 1 || turn = 1);
T3Inv1;
~(pc2=M5);
T3Inv2;
L5=L5;
T3Loc1;
T3Loc2;
T3Turn;
T3Inv;
|- {Inv} pc1=L1&&~c2=0->pc1:=L5{Inv};
qed T3;

prove |- {Inv} pc1=L2 && turn=1->pc1:=L1 {Inv};
assume Inv && (pc1=L2 && turn=1);
pc1=L2;
pc1=L1||pc1=L2||pc1=L3||pc1=L5;
I1;
c1=0;
T4I1;
~(L1 = L0 || L1 = L4);
T4J1;
T4I2;
T4J2;
~(L1=L5);
T4Inv1;
L1=L1;
(L1 = L1 || L1 = L2 || c1 = 1 || turn = 2);
T4Inv2;
L1=L1;
T4Loc1;
T4Loc2;
T4Turn;
T4Inv;
|- {Inv} pc1=L2 && turn=1->pc1:=L1 {Inv};
qed T4;
```

```
prove |- {Inv} pc1=L2 && ~(turn=1)->pc1:=L3 {Inv};
assume Inv && (pc1=L2 && ~(turn=1));
pc1=L2;
pc1=L1||pc1=L2||pc1=L3||pc1=L5;
I1;
c1=0;
T5I1;
~(L3 = L0 || L3 = L4);
T5J1;
T5I2;
T5J2;
~(L3 = L5);
T5Inv1;
~(turn=1);
Turn;
turn=2;
(L3 = L1 || L3 = L2 || c1 = 1 || turn = 2);
T5Inv2;
L3=L3;
T5Loc1;
T5Loc2;
T5Inv;
|- {Inv} pc1=L2 && ~(turn=1)->pc1:=L3 {Inv};
qed T5;

prove |- {Inv} pc1=L3 ->c1:=1, pc1:=L4 {Inv};
assume Inv && pc1=L3;
~(L4 = L1 || L4 = L2 || L4 = L3 || L4 = L5);
T6I1;
1=1;
T6J1;
T6I2;
T6J2;
~(L4=L5);
T6Inv1;
(L4 = L1 || L4 = L2 || 1 = 1 || turn = 2);
T6Inv2;
L4=L4;
T6Loc1;
```

```
T6Loc2;
T6Turn;
T6Inv;
|- {Inv} pc1=L3 ->c1:=1, pc1:=L4 {Inv};
qed T6;

prove |- {Inv} pc1=L4 && turn=2-> pc1:=L4 {Inv};
assume Inv && (pc1=L4&&turn=2);
~(L4 = L1 || L4 = L2 || L4 = L3 || L4 = L5);
T7I1;
I1;
J1;
pc1=L4;
(pc1 = L0 || pc1 = L4);
c1=1;
T7J1;
T7I2;
T7J2;
~(L4=L5);
T7Inv1;
(L4 = L1 || L4 = L2 || c1 = 1 || turn = 2);
T7Inv2;
L4=L4;
T7Loc1;
T7Loc2;
T7Turn;
T7Inv;
|- {Inv} pc1=L4 && turn=2-> pc1:=L4 {Inv};
qed T7;

prove |- {Inv} pc1=L4 && ~(turn=2)-> pc1:=L0 {Inv};
assume Inv && (pc1=L4 && ~(turn=2));
~(L0 = L1 || L0 = L2 || L0 = L3 || L0 = L5);
T8I1;
pc1=L4;
(pc1 = L0 || pc1 = L4);
J1;
c1=1;
T8J1;
T8I2;
```

```
T8J2;
~(L0 = L5);
T8Inv1;
(L0 = L1 || L0 = L2 || c1 = 1 || turn = 2);
T8Inv2;
L0=L0;
T8Loc1;
T8Loc2;
T8Turn;
T8Inv;
|- {Inv} pc1=L4 && ~(turn=2)-> pc1:=L0 {Inv};
qed T8;

prove |- {Inv} pc1=L5-> turn:=2, c1:=1, pc1:=L0 {Inv};
assume Inv && pc1=L5;
~(L0 = L1 || L0 = L2 || L0 = L3 || L0 = L5);
T9I1;
1=1;
T9J1;
T9I2;
T9J2;
~(L0=L5);
T9Inv1;
(L0 = L1 || L0 = L2 || 1 = 1 || 2 = 2);
T9Inv2;
L0=L0;
T9Loc1;
T9Loc2;
2=2;
T9Turn;
T9Inv;
|- {Inv} pc1=L5-> turn:=2, c1:=1, pc1:=L0 {Inv};
qed T9;

prove |- {Inv} pc2=M0 -> pc2:=M1, c2:=0 {Inv};
assume Inv && pc2=M0;
T10I1;
T10J1;
0=0;
T10I2;
```

```
~(M1=M0 || M1=M4);
T10J2;
M1=M1;
(M1 = M1 || M1 = M2 || 0 = 1 || turn = 1);
T10Inv1;
~(M1=M5);
T10Inv2;
T10Loc1;
T10Loc2;
T10Turn;
T10Inv;
|- {Inv} pc2=M0 -> pc2:=M1, c2:=0 {Inv};
qed T10;

prove |-{Inv} pc2=M1 && c1=0 ->pc2:=M2{Inv};
assume Inv && (pc2=M1 && c1=0);
Inv;
T11I1;
T11J1;
pc2=M1;
(pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5);
I2;
c2=0;
(M2 = M1 || M2 = M2 || M2 = M3 || M2 = M5) => c2 = 0;
T11I2;
~(M2 = M0 || M2 = M4);
T11J2;
M2=M2;
(M2 = M1 || M2 = M2 || c2 = 1 || turn = 1);
T11Inv1;
~(M2 = M5);
T11Inv2;
T11Loc1;
T11Loc2;
T11Turn;
T11Inv;
|-{Inv} pc2=M1 && c1=0 ->pc2:=M2{Inv};
qed T11;

prove |- {Inv}pc2=M1 && ~(c1=0) -> pc2:=M5{Inv};
```

```
assume Inv && (pc2=M1 && ~(c1=0));
T12I1;
T12J1;
pc2=M1;
I2;
(pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5);
c2 = 0;
T12I2;
~(M5 = M0 || M5 = M4);
T12J2;
~(c1=0);
~(pc1=L1 || pc1=L2 || pc1=L3 ||pc1=L5);
~(pc1=L5);
T12Inv1;
Loc1;
pc1 = L0 || pc1 = L4;
Inv2;
c1=1;
(pc1 = L1 || pc1 = L2 || c1 = 1 || turn = 2);
T12Inv2;
T12Loc1;
M5=M5;
T12Loc2;
T12Turn;
T12Inv;
|- {Inv}pc2=M1 && ~(c1=0) -> pc2:=M5 {Inv};
qed T12;

prove |-{Inv} pc2=M2 && turn=2 ->pc2:=M1 {Inv};
assume Inv && (pc2=M2 && turn=2);
T13I1;
T13J1;
I2;
pc2=M2;
(pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5);
c2=0;
T13I2;
~(M1 = M0 || M1 = M4);
T13J2;
M1=M1;
```

```
(M1 = M1 || M1 = M2 || c2 = 1 || turn = 1);
T13Inv1;
~(M1 = M5);
T13Inv2;
T13Loc1;
T13Loc2;
T13Turn;
T13Inv;
|-{Inv} pc2=M2 && turn=2 ->pc2:=M1 {Inv};
qed T13;

prove |-{Inv} pc2=M2 && ~(turn=2) ->pc2:=M3 {Inv};
assume Inv && (pc2=M2 && ~(turn=2));
T14I1;
T14J1;
pc2=M2;
I2;
pc2=M2;
(pc2 = M1 || pc2 = M2 || pc2 = M3 || pc2 = M5);
c2 = 0;
T14I2;
~(M3 = M0 || M3 = M4);
T14J2;
~(turn=2);
Turn;
turn = 1;
(M3 = M1 || M3 = M2 || c2 = 1 || turn = 1);
T14Inv1;
~(M3 = M5);
T14Inv2;
T14Loc1;
M3=M3;
T14Loc2;
T14Turn;
T14Inv;
|-{Inv} pc2=M2 && ~(turn=2) ->pc2:=M3 {Inv};
qed T14;

prove |- {Inv} pc2=M3 -> c2:=1, pc2:=M4 {Inv};
assume Inv && pc2=M3;
```

```
T15I1;
T15J1;
~(M4 = M1 || M4 = M2 || M4 = M3 || M4 = M5);
T15I2;
1=1;
T15J2;
(M4 = M1 || M4 = M2 || 1 = 1 || turn = 1);
T15Inv1;
~(M4 = M5);
T15Inv2;
T15Loc1;
M4=M4;
T15Loc2;
T15Turn;
T15Inv;
|- {Inv} pc2=M3 -> c2:=1, pc2:=M4 {Inv};
qed T15;

prove |- {Inv} pc2=M4 && turn=1 -> pc2:=M4{Inv};
assume Inv && (pc2=M4 && turn=1);
T16I1;
T16J1;
~(M4 = M1 || M4 = M2 || M4 = M3 || M4 = M5);
T16I2;
pc2=M4;
pc2=M0 || pc2=M4;
J2;
c2=1;
T16J2;
turn=1;
(M4 = M1 || M4 = M2 || c2 = 1 || turn = 1);
T16Inv1;
~(M4 = M5);
T16Inv2;
T16Loc1;
M4=M4;
T16Loc2;
T16Turn;
T16Inv;
|- {Inv} pc2=M4 && turn=1 -> pc2:=M4{Inv};
```

```
qed T16;

prove |- {Inv} pc2=M4 && ~(turn=1) -> pc2:=M0{Inv};
assume Inv && (pc2=M4 && ~(turn=1));
T17I1;
T17J1;
~(M0 = M1 || M0 = M2 || M0 = M3 || M0 = M5);
T17I2;
pc2=M4;
pc2=M0 || pc2=M4;
J2;
c2 = 1;
T17J2;
(M0 = M1 || M0 = M2 || c2 = 1 || turn = 1);
T17Inv1;
~(M0 = M5);
T17Inv2;
T17Loc1;
M0=M0;
(M0=M0||M0=M1||M0=M2||M0=M3||M0=M4||M0=M5);
T17Loc2;
T17Turn;
T17Inv;
|- {Inv} pc2=M4 && ~(turn=1) -> pc2:=M0{Inv};
qed T17;

prove |- {Inv} pc2=M5 -> turn :=1, c2:=1, pc2:=M0 {Inv};
assume Inv && pc2=M5;
T18I1;
T18J1;
~(M0 = M1 || M0 = M2 || M0 = M3 || M0 = M5);
T18I2;
1=1;
T18J2;
(M0 = M1 || M0 = M2 || 1 = 1 || 1 = 1);
T18Inv1;
~(M0 = M5);
T18Inv2;
T18Loc1;
M0=M0;
```

```
T18Loc2;
T18Turn;
T18Inv;
|- {Inv} pc2=M5 -> turn :=1, c2:=1, pc2:=M0 {Inv};
qed T18;
```

Bibliography

[AFB+88] T. Alspaugh, S. Faulk, K. Britton, R. Parker and D. L. Parnas. *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory, 1988.

[AG93] J. Atlee and J. Gannon. *"State-based model checking of event-driven system requirements."* *IEEE ToSE*, 19(1):24–40, 1993.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice-Hall 1982.

[Bha94] R. Bharadwaj. *TOP/*SNAP *– A Transition Oriented Prover for PROMELA.* Technical Memorandum TM 94-24199C, AT&T Bell Laboratories, Murray Hill, NJ 1994.

[Bha96] R. Bharadwaj. *"Verification of SCR Specifications."* In *Proc. 5th Int'l Software Cost Reduction Workshop.* Bell-Northern Research Ltd., Ottawa, Ontario Canada, Feb 1996.

[BFS95] R. Bharadwaj and A. Felty and F. A. Stomp. *"Formalizing Inductive Proofs of Network Algorithms."* In *Proc. 1st Asian Computing Science Conference* Pathumthani, Thailand, December 1995.

[BS94] R. Bharadwaj and F. A. Stomp. *Towards a Checker/Annotator for Proofs of Distributed Programs.* Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ 1994.

[BSW69] K.A. Bartlett, R.A. Scantlebury and P.T. Wilkinson. *"A note on reliable full-duplex transmission over half-duplex lines."* CACM 12(5):260–265, 1969.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[Con86] R. L. Constable. *Implementing Mathematics.* Prentice-Hall, 1986.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. *"Automatic verification of finite-state concurrent systems using temporal logic specifications."* ACM TOPLAS 8(2):244–263, 1986.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. *"Memory efficient algorithms for the verification of temporal properties."* In *Formal Methods in System Design*, 1: 275–288, Kluwer Academic Publishers, 1992.

[Dij68] E. W. Dijkstra. *"Cooperating sequential processes."* In *Programming Languages*, F. Genuys (Ed.), Academic Press, NY 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976

[DJ89] N. Dershowitz and J.-P. Jouannaud. *"Rewrite Systems"* Handbook of Theoretical Computer Science, Volume *B*, Chapter 15, 1989.

[Dow93] G. Dowek, et. al. *The Coq Proof Assistant User's Guide.* Technical Report 154, INRIA, 1993.

[Fau89] S. Faulk. *State Determination in Hard-Embedded Systems.* Ph. D. Thesis, Univ. of N. Carolina, 1989.

[Fel93] A. Felty. *"Implementing tactics and tacticals in a higher-order logic programming language."* Journal of Automated Reasoning, 11(1):43–81, 1993.

[Fra86] N. Francez. *Fairness.* Springer Verlag, 1986.

[GHM87] J. D. Gannon, R. G. Hamlet and H. D. Mills. *"Theory of modules."* IEEE Trans. on Software Engg. SE-13, July 1987, pp. 820–829.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GMW79] M. J. Gordon, A. J. Milner and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, New York 1981.

[HCM92] E. L. Hahne, A. K. Choudhury and N. F. Maxemchuk. *"DQDB networks with and without bandwidth balancing."* IEEE Transactions on Communications, 40(7): 1192–1204, July 1992.

[HD83] J. Hsiang and N. Dershowitz. *"Rewrite methods for clausal and nonclausal theorem proving"* Proc. of 10th EATCS International Colloquium on Automata, Languages, and Programming, LNCS-154, pp. 331–346, 1983.

[Hen80] K. L. Henninger. *"Specifying software requirements for complex systems: new techniques and their application." IEEE ToSE*, SE-6(1), 1980.

[HJL95] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. *Tools for analyzing SCR-style requirements specifications: A formal foundation.* Technical Report NRL-7499, NRL, Washington DC, 1995. In preparation.

[Hoa78] C. A. R. Hoare. *"Communicating sequential processes." CACM*, 21(8):666–697, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[Hol88] G. J. Holzmann. *"An improved protocol reachability analysis technique."* Software Practice and Experience, 18(2): 137–161, 1988.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Software Series, 1991.

[Hol93] Gerard J. Holzmann. *"Design and validation of protocols: a tutorial."* Computer Network and ISDN Systems 25 (1993), pp. 981–1017.

[HS82] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press 1984.

[Jac94a] P. Jackson. *The Nuprl Proof Development System Version 4.1 — Introductory Tutorial.* Department of Computer Science, Cornell University, Apr 15, 1994.

[Jac94b] P. Jackson. *The Nuprl Proof Development System Version 4.1 — Reference Manual and User's Guide.* Department of Computer Science, Cornell University, Apr 15, 1994.

[KL93] R. P. Kurshan and L. Lamport. *"Verification of a Multiplier: 64 Bits and Beyond."* In *Proceedings of the 5th International Workshop on Computer-Aided Verification*, pages 166–179. LNCS 697, Springer Verlag, 1993.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, N. J. 2^{nd} ed. 1988.

[Kur89] R. P. Kurshan. *"Analysis of discrete event coordination."* Lecture Notes in Computer Science 430, pp. 414–453.

[McC90] W. W. McCune. *Otter 2.0 Users Guide.* Report ANL-90/9, Argonne National Laboratory, 1990.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Men87] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole, 1987.

[MN95] Olaf Müller and Tobias Nipkow. *"Combining model checking and deduction for I/O-automata."* In *Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Technical Report NS-95-2, BRICS Notes Series, Aarhus, 1995.

[MP83] Z. Manna and A. Pnueli. *"Verification of concurrent programs: A temporal proof system."* Foundations of Computer Science IV, Distributed Systems: Part 2, Mathematical Centre Tracts 159, Center for Mathematics and Computer Science, Amsterdam, 163–255, 1983.

[MP90] Z. Manna and A. Pnueli. *"Tools and rules for the practicing verifier."* In *Carnegie Mellon Computer Science: A 25-year Commemorative*, ACM Press 1990.

[MP91a] Z. Manna and A. Pnueli. *"Completing the temporal picture."* Theoretical Computer Science, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[MS91] K. L. McMillan and J. Schwalbe. *"Formal verification of the encore gigamax cache consistency protocol."* International Symposium on Shared Memory Multiprocessors, 1991.

[Par72] D. L. Parnas. *"On the criteria to be used in decomposing systems into modules."* CACM, Vol. 15, Dec 1972, pp. 1053–1058.

[Par84] D. L. Parnas. *"Software engineering principles."* INFOR, Vol. 22, Nov 1984, pp. 303-316.

[Par93a] D. L. Parnas. *"Some Theorems We Should Prove"*. Proc. Int'l Meeting on Higher Order Logic, Theorem Proving and its Applications, Vancouver, BC, pp. 156 – 163, 1993.

[Pau90] L. C. Paulson. *"Isabelle: the next 700 theorem provers."* In *Logic and Computer Science*, The APIC Series 31, P. Odifreddi (ed.), Academic Press 1990.

[Pel93] D. Peled. *"All from one, one for all: on model checking using representatives."* In *Proc. 5th Workshop on Computer Aided Verification*, Elounda, Springer Verlag June 1993.

[Plo81] G. D. Plotkin. *A Structural Approach to Operational Semantics.* DAIMI FN-19, September 1981, reprinted April 1991.

[PM91] D. L. Parnas and J. Madey. *Functional Documentation for Computer Systems Engineering.* T.R. 237, Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada September 1991.

[RvH91] J. Rushby and F. von Henke. *"Formal verification of algorithms for critical systems."* Proc. of the *ACM SIGSOFT'91 Conference on Software for Critical Systems*, pp. 1–15.

[RSS95] S. Rajan and N. Shankar and M. K. Srivas. *"An integration of model-checking with automated proof checking."* In *Proceedings of the 7th International Workshop on Computer-Aided Verification.* LNCS, Springer Verlag, 1995.

[SB89] P. A. Subrahmanyam, Graham Birtwistle, Editors. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer Verlag, 1989.

[Sie90] J. H. Siekmann. *"An introduction to unification theory."* Formal Techniques in Artificial Intelligence, R. B. Banerji, Editor, Elsevier Science Publishers BV, North-Holland, 1990.

[VW86] M. Y. Vardi and P. Wolper. *"An automata-theoretic approach to automatic program verification."* In Proceedings of a *Symposium on Logic in Computer Science*, pp. 322–331, Cambridge June 1986.

[Wol89] P. Wolper. *"On the relation of programs and computations to models of temporal logic."* In *Proceedings of Temporal Logic in Specification*, LNCS 398 B. Banieqbal, H. Barringer and A. Pneuli (Eds.), pp. 75–123, Springer Verlag 1989.