Formal Verification of Concurrent Programs Based on Type Theory

Shen-Wei Yu

Ph.D. Thesis

Department of Computer Science
University of Durham

October 1998

Abstract

Interactive theorem proving provides a general approach to modeling and verification of both finite-state and infinite-state systems but requires significant human efforts to deal with many tedious proofs. On the other hand, model-checking is limited to some application domain with small finite-state space. A natural thought for this problem is to integrate these two approaches. To keep the consistency of the integration and ensure the correctness of verification, we suggest to use type theory based theorem provers (e.g. Lego) as the platform for the integration and build a model-checker to do parts of the verification automatically.

We formalise a verification system of both CCS and an imperative language in the proof development system Lego which can be used to verify both finite-state and infinite-state problems. Then a model-checker, LegoMC, is implemented to generate Lego proof terms for finite-state problems automatically. Therefore people can use Lego to verify a general problem with some of its finite sub-problems verified by LegoMC. On the other hand, this integration extends the power of model-checking to verify more complicated and infinite-state models as well.

The development of automatic techniques and the integration of different reasoning methods would directly benefit the verification community. It is expected that further extension and development of this verification environment would be able to handle real life systems. On the other hand, the research gives us some experiences about how to automate proofs in interactive theorem provers and therefore will improve the usability and applicability of the theorem proving technology.

Acknowledgements

First and foremost, I would like to thank my supervisors Zhaohui Luo and Keith Bennett. My work owes a great deal to insights and ideas which Zhaohui shared with me in numerous stimulating and productive discussions. Without Zhaohui's careful guidance and conscientious reading, this thesis would not have been possible. Keith gave me many administrative support including applying the Overseas Research Students Awards.

I would also like to thank my PhD proposal examiners, Cornelia Boldyreff and Maria Fox, for their valuable advice in the very beginning of my research. The Lego club in Durham, Alexander Jones, Paul Callaghan, James McKinna, Steven Bradley, Sanjay Mistry, gave me many helpful suggestions and many of my ideas come from stimulating discussions with them.

I would sincerely like to thank all my friends in Durham and my landlord. Their great help and friendships make my life in this city so interesting, inspiring and enjoyable.

My parents gave me not only financially support but also constant encouragement.

My wife, Jiuan-jiuan, has given me great emotional support through her love and patience. She also take care of my daily life and my lovely daughter so that I can concentrate on my work.

My research was supported in part by Overseas Research Students Awards ORS96012011.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Durham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Copyright Notice

Contents

I Introduction **1**

1 Introduction **2**

1.1 Formal Verification 2

1.2 Model-Checking 4

1.3 Inductive Theorem Proving 6

1.4 Integration 8

1.5 Our Approach 11

1.6 Structure of the Thesis 14

2 Preliminary **17**

2.1 Inductive Data Type 18

2.2 Internal Higher Order Logic 20

2.3 Lego Notations 23

2.4 Set Theory 28

2.5 Fix Points 30

II Formalisation 34

3 Temporal Logics 35

3.1 μ-calculus 36

3.2 Formalisation of μ-calculus 38

3.3 Inference Rules 41

3.4 Embedding of Temporal Logics 44

4 System Modeling and CCS 49

4.1 Concurrent Systems 50

4.2 CCS: Calculus of Communicating System 50

4.3 de Bruijn's Index 53

4.4 Semantics of CCS 54

4.5 Lemmas and Theorems 58

4.6 Example: a Ticking Clock 59

5 An Imperative and Concurrent Programming Language (ICPL) 61

5.1 The Syntax 62

5.2 Shared Memory and Transitional Semantics 67

5.3 Example - A Mutual Exclusion Algorithm 74

III LegoMC 77

6 The Model-Checker, LegoMC 78

6.1 Model-Checking 78

6.2 System Structure and Inference Rules 80

6.3 The Implementation 84

6.4 User Interface 86

6.4.1 CCS 86

6.4.2 ICPL 87

6.4.3 Temporal Logics 89

6.4.4 Commands, Comments and Abbreviations 91

6.5 Examples 91

7 Finite-State Examples **95**

7.1 A Simple Communicating Protocol 96

7.1.1 Modeling in ICPL 97

7.1.2 Modeling in CCS 100

7.1.3 Comparison 102

7.2 Mutual Exclusion Algorithms 103

7.2.1 Dekker's Algorithm 104

7.2.2 Dijkstra's Algorithm 107

7.2.3 Hyman's Algorithm 110

7.2.4 Knuth's Algorithm 112

7.2.5 Peterson's Algorithm 115

7.2.6 Lamport's Algorithm 117

7.2.7 Results and Comments 118

8 Infinite-State Case Studies 120

8.1 Proving by Semantics and Induction 121

8.1.1 Example : an Infinite Counter 122

8.1.2 Example: a Token Ring Network 123

8.2 Composition 129

8.2.1 Compositional Rules for CCS 131

8.2.2 Example 133

8.3 Abstraction 134

8.3.1 Strong Bisimulation 134

8.3.2 Abstraction Mapping 135

8.3.3 An Example 136

8.4 Discussion 137

IV Proof Generation and Future Research 138

9 Automatic Generation of Proof Terms 139

9.1 Proof Term Construction 140

9.2 Automatic Generation of Proof Terms 144

9.2.1 Use of Internal Functions 145

9.2.2 Use of Tacticals 146

9.2.3 Use of External Programs to Generate Proof Scripts 147

9.2.4 External Programs to Generate Proof Terms 147

9.3 Efficiency Issues 148

9.4 Discussion 150

10 Other Automation Issues 152

10.1 Equational Rewriting 153

10.2 BDD Propositional Simplifier 156

10.3 Arithmetic Decision Procedures 159

11 Conclusion 163

11.1 Summary 163

11.2 Future Research 164

List of Figures

1.1 The system structure of LegoMC 12

5.1 A solution for the mutual exclusion problem 74

6.1 The system structure of LegoMC 81

7.1 A simple transport protocol 98

8.1 A token ring network with 12 workstations 124

9.1 On-the-fly lemma generation 150

List of Tables

3.1 Embedding of CTL* in μ-calculus 48

7.1 Verification results of mutual exclusion algorithms 119

Part I

Introduction

Chapter 1

Introduction

Along with the rapid growth of computer application in our daily life, it becomes essential to ensure the correctness of certain computer or computer-embedding systems because of safety, security or cost concerns. Formal verification has become an important technique to ensure the correctness of computer software or hardware. There are roughly two categories of approaches of formal verification, interactive theorem proving and automatic model-checking. Recent researches suggest that the integration of two approaches would be an effective method to verify complicated systems. Following this idea, this thesis presents a verification environment based on type theory based theorem prover, Lego.

1.1 Formal Verification

More and more devices, equipment and machines such as airplanes, cars, etc. are controlled by computers. Along with the growth of complexity in electronic circuits, computer software and computer-controlled systems, it becomes very difficult to ensure that the systems behave as intended. The discovery of 'bugs' in commercial software is very common and therefore companies usually setup a bug report channel. None can claim that their programs are bug-free and computer users have got used to discovering minor bugs. Although bugs for most commercial software can be just an inconvenience to the user, it could be a disaster for hardware or software systems used to monitor and/or control some physical systems such as a nuclear plant and an airplane. Even for a microprocessor design or a financial transaction system, a design error can cost the company huge money.

A typical approach to enhance the correctness of designs is *testing* or *simulation*, in which test data are given to a model of the design and the results are analyzed against the specification. However, exhaustive testing is impractical for many systems since the possible input combinations can be very huge and even infinite. The assurance therefore relies on a very careful choice of some input data that is supposed to exercise all the features of the design. As the complexity increases, it becomes more difficult to select test data with a good coverage. Moreover, some testing models and environments can cost a lot and some cannot even be built.

Formal verification is another approach to ensure correctness. The approach is to represent and analyse systems formally (mathematically) by proving that a system satisfies its functional specification or some critical properties. The system can be hardware systems such as electronic circuits, software systems such as computer programs or reactive systems such as controlling systems. The system is usually modeled using a system description language such as transition graph, CCS, CSP, or even a programming language. The specification is written using some specification languages. The verification is to prove that the model of the real system satisfies the specification.

The foundation of verification backtracks to Floyd and Hoare's work [Flo67, Hoa69] on sequential programs. Now they are known as Hoare's logic [Hoa69, Apt81, dB80]. A sequential program is usually regarded as a mathematical function over memory states. Given a function, we can always deduce the final state from the start state. The memory is subservient to the program. The story is completely different if other agents, e.g. programs or environments, may interfere and change the state of memory while the program is running. In a concurrent environment with two or more agents interfering with each other, the final state is not only dependent on the start state but on the behaviours of all the agents in the systems. Therefore, the verification of concurrent systems is more complicated than sequential systems and more difficult to deal with. The testing in concurrent systems is more difficult as well and therefore it is more necessary to use verification.

In the early stages, the formal verification is carried on paper and pencil method. Several theorems are proposed to reason about the correctness of programs. The increase in complexity of programs rapidly becomes unmanageable without computer assistance. The development in automatic proving algorithms and *interactive theorem provers* make formal verification more practical in application. A popular automatic verification technique, *model-checking*, is introduced in next section. A general introduction to the application of interactive theorem proving to verification is presented in section 1.3.

1.2 Model-Checking

Over the last decade *model-checking* has emerged as a powerful technique for automatically verifying concurrent systems [CE81, QS81, CES86, VW86, Cle90, And92]. Many different model-checking techniques emerged and various communication protocols and electronic circuits have been verified by model-checkers. The improvement in efficiency has successfully extended the application to more large-scale and complicated systems such as circuits with 10^{20} states [BCM+92] and PDP-11 sized processor [BB94]. It has been extended to probabilistic [Var85, PZ86, CY90] and real time programs and logics [ACD90, AH90, HLP90].

The basic idea is to determine whether or not a system satisfies a property typically expressed as a temporal logic formula by searching the state space of the system thoroughly. When systems have finite-state space, model-checking algorithms can be used to verify the system completely automatically. The restriction is that the state space should be finite. If the model-checking algorithm is efficient, this approach is potentially of wide applicability since a large class of concurrent systems has finite state space. A potentially serious drawback to the entire model-checking approach is the *state-explosion* problem that the size of the global state transition graph grows exponentially while the size of the system grows linearly.

Several techniques have been introduced to cope with the state-explosion problem. *Local model-checking* [Lar90, SW91, Win89] or *on-the-fly* [Hol81, Hol85] model-checking attempts to build only part of the state space of the system, while still maintaining the ability to check the properties of interest. *Partial-order techniques* attempt to avoid the wasteful representation of concurrency by interleaving [GW93]. *Abstraction techniques* replace the system to be checked by a simpler one in which the details irrelevant for the property to be checked have been suppressed [Kur94, CGL92, GL93]. Another direction called *symbolic approach* is to represent implicitly rather than explicitly the states and transition relations of systems [BCM+92]. The usual implicit representation is Binary Decision Diagrams(BDDs) [Bry92] so that the temporal formulas can be model-checked directly on the BDD representation, without ever building an explicit representation of the state space.

Although the improvement in efficiency has significantly widened the application of model-checking techniques, the state enumeration, which is the basic principle of model-checking, still limits it to finite-state systems. Moreover, the above techniques to improve model-checking efficiency still need to be formally proved. A hand written proof attached to an algorithm is not always sufficient. Certain techniques such as abstraction technique involve creating another simpler system to replace the original system. The process of abstraction needs to be formally proved to ensure the correctness of the whole verification. Interactive theorem proving can be used in at least two aspects to benefit model-checking community:

1. It can be used to decompose a verification problem into sub-problems so that each of which is manageable by one of the two methods.
2. It can be used to formally prove the meta-theory of model-checking.

1.3 Inductive Theorem Proving

Instead of checking the state space exhaustively, interactive theorem proving deduces the result by inference rules guided interactively by human being. Therefore, users can choose different reasoning methods which are more suitable for their verification target. The rich build-in library can also simplify their task. Most of the theorem-provers are called *LCF-style* theorem provers [GMW79], these include *HOL* [GM93], *PVS* [ORS92], *Nurpl* [C$^+$86], *Coq* [D$^+$91], *Lego* [LP92], etc.

Edinburgh LCF was developed by Robin Milner and his colleagues around 1977 [GMW79]. Edinburgh LCF was programmable. The user could write programmable meta-language (called ML) functions to process terms, formulae, and theorems. Theorems were not simply created, but proved. Type checking ensures that theorems are only proved by applying rules to axioms and other theorems. It uses tactics for backward proof. Each tactic specifies a backward proof step, reducing a goal that is the conjecture to be proved to sub-goals. A LCF tactic is a function that reduces a goal to zero or more sub-goals. Once all the sub-goals have been proved, some mechanism constructs the corresponding forward proof and yields the desired theorem. Tacticals are the combination of several tactics in various ways. Tactics and tacticals constitute a powerful control language, which can describe search methods. Users choose the tactics to apply and computers reduce goals by applying assigned tactics and return sub-goals to be further proved. In the whole process, users prove theorems by interaction with computers.

The HOL system [GM93], based upon the LCF system, is another interactive theorem prover using classical higher order logic. The deductive machinery is natural deduction proof using the meta language ML for defining tactics and tacticals. Theorems can only be introduced into the system using formal proofs that rely upon the theorems and axioms which are already present within the system. However, subtypes and dependent types are not supported and the insistence on resolving proofs into simple primitive inferences can make HOL slow.

The major feature of PVS [ORS92], beside the common features of LCF-style theorem provers, is a powerful base of primitive inference rules for various decision procedures and rewriting to automate proofs. PVS has also a strategy language for combining inference steps into more complicated proof strategies which are similar to tactics and tacticals. Among decision procedures, there is a symbolic model-checker builded in PVS [RSS95].

Nurpl [C^+86] is based on Martin Löf type theory. One of the features of type theory based theorem provers is that the logic and the system take account of the computational meaning of assertions and proofs. For instance, given a constructive existence proof the system can use the computational information in the proof to build a representation of the object, which demonstrates the truth of the assertion. Such proofs can thus be used to provide data for further computation or display.

Coq [D^+91] is an implementation of the Calculus of Inductive Constructions, which is a non-conservative extension of the Calculus of Constructions with inductive types. It is a goal-directed tactics theorem prover, with a set of predefined tactics, including an Auto tactic which tries to apply previous lemmas declared as hints. The logic mixes a constructive logic and a classical logic. The system automatically extracts the constructive contents of proofs as an executable ML program that permits the development of programs provably consistent with their specification.

Lego is an interactive proof development system designed and implemented by Randy Pollack in Edinburgh [LP92]. It implements several related type systems– the Edinburgh Logical Framework [HHP92], the calculus of constructions [CH88], the Extended Calculus of Constructions [Luo94], and UTT [Luo94]. Lego is a powerful tool for interactive proof development in the natural deduction style and supports refinement proof as a basic operation and a definitional mechanism to introduce definitional abbreviations. Lego also allows the user to specify new inductive data type (computational theories), which supports the computational use of the type theory. General applications of Lego at the moment are to formalise a system and reason about its properties, such as the verification of proof checkers [Pol95].

Although theorem proving is more general in applications, it requires intensive human guidance and only experienced experts can use interactive theorem provers effectively. Rigorous theorem proving requires the user to consider every detail including some obvious assumptions, which is usually omitted in manual proving. It is observable that some proving tasks can be carried out completely by automatic techniques such as model-checking. On the other hand, model-checking is limited to some application domain with small finite-state space. A natural thought for this problem is to integrate those two approaches.

1.4 Integration

Many real life systems are usually very complicated and therefore are very difficult to be dealt with merely by one verification technique. There are in practice demands to divide a complicated problem to smaller parts and then use different verification methods to tackle individual parts. The verification results for individual parts can then be integrated to finish the whole verification task. Many techniques such as deduction, composition, abstraction and induction are proposed and corresponding tools are developed to deal with various classes of infinite-state systems. It is believed that the integration of the above techniques and various automatic techniques would enhance dramatically the application of those individual techniques and therefore be able to deal with real-life problems.

Interactive theorem provers are suitable candidates to serve as the platform of the above integration because most of them are based on higher order logic and therefore easy to encode other logics. Moreover, the inductive data type mechanisms in many theorem provers provide a very convenient way of formalising systems. However, interactive theorem proving requires significant human efforts to deal with many tedious proofs. Even a simple model like the 2-process mutual exclusion problem is fairly complicated to verify. To be used in practice, it is necessary to borrow some automatic techniques, e.g. model-checking.

Wolper and Lovinfosse [WL89] and Kurshan and McMillan [KM89] extended model-checking for inductive proofs by using an invariant to capture the induction hypothesis in the inductive step. Joyce and Seger [JS93] used HOL theorem prover to verify formulas which contain uninterpreted constants as lemmas which are verified by Voss model-checker. Kurshan and Lamport [KL93] proved a multiplier where the 8-bit multiplier can be verified by COSPAN model-checker [Kur94] and the n-bit multiplier composed from 8-bit multipliers can be verified by TLP theorem prover [EGL92]. In principle, these approaches are to divide the whole problem to separated sub-problems and then use different tools to solve individual problems. Their works based on paper and pencils are the early attempts of combining theorem proving and model-checking.

However, the integration of these two approaches is still not tight enough. Müller and Nipkow [MN95] used HOL theorem prover to reduce the alternating bit protocol expressed in I/O automata to a finite state one to be verified by their own model-checker. PVS proof checker [ORS92] even includes a model-checker as a decision procedure which presented the possibility of combining theorem proving and model-checking in a smooth and tight way [RSS95]. However, the correctness of model-checkers is still a big concern since model-checkers themselves are computer software, which could contain bugs. The output of most model-checkers including the model-checker of PVS for a correct system is only a "TRUE." People can only choose to believe that "TRUE" as a pure act of faith, or not at all.

On the other hand, the proofs of type theory based theorem provers, such as LEGO [LP92], ALF [ACN90, Mag92], Coq [D^+91] and Nuprl [C^+86], are proof terms (λ terms) which in principle can be justified by different proof checkers so that people can have more confidence on formal proofs. Moreover, proof terms provide a common interface for different tools so that we can easily integrate various tools to complete more complicated proofs. The integration based on proof terms can also ensure the consistency between different verification techniques. Another issue is that the user interface of general theorem provers seems very complicated for people merely doing verification.

It is believed that for the verification to be realised for industrial applications, a *domain-specific verification environment* should be developed to

1. support high level user-familiar and domain-specific languages for problem description at the appropriate level of abstraction,

2. integrate different methods of reasoning useful and suitable for the particular domains concerned (e.g. inductive reasoning, semantics reasoning, abstraction, composition, and automatic proof generation),

3. provide a relatively high degree of automation.

1.5 Our Approach

Our approach towards the above goals is using type theory based theorem prover, Lego, as the platform for the integration, and a domain-specific interface and automatic tools are built to generate parts of the proof. The interface uses general programming language syntax so that programmers do not have to learn a new syntax dedicated to theorem proving. A model-checker is implemented to do the verification automatically for systems or sub-systems with finite state space. The integration is based on explicit proof terms, which ensure the correctness and consistency of the integration.

One of the major differences between type theory based theorem provers with other theorem provers and automatic verifiers is the *proof terms*. Proof terms are λ-*terms* of which the correctness can be checked by type checking algorithms and therefore give us more confidence on the proof. The proof checking of Lego helps to ensure the correctness of the verification. The expressive higher order logic and inductive data type mechanism in Lego enable us to embed specification languages and formalise system description languages very easily.

The achievement of this thesis in the above direction is building the kernel of a verification environment based on Lego. This involves

1. adapting proof techniques which are usually carried by hands to theorem provers,

2. developing automatic techniques to help the proof in the theorem provers,

3. integrating various proof techniques and automatic algorithms on a consistent platform.

The system structure is as Fig. 1.1.

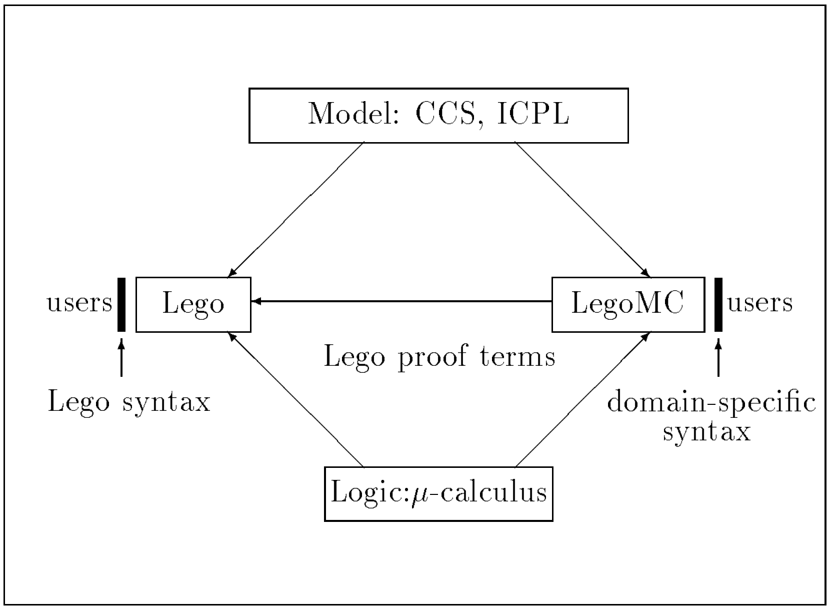

Figure 1.1: The system structure of LegoMC

We have implemented both *message-passing* and *shared-memory* models of concurrent systems. For the message-passing model, the Calculus of Communicating Systems (CCS) [Mil89] is used. We also define a simple imperative and concurrent programming language to model the shared-memory systems. We use the propositional μ-calculus [Koz83] to express temporal properties and specifications and define other temporal logics, CTL and LTL, as syntax abbreviation of μ-calculus formulas. CCS, the imperative language and propositional μ-calculus are formalised and inference rules are proved in Lego to verify both finite and infinite state systems. Furthermore, a model-checker (LegoMC) [YL97] is implemented to automatically verify finite part of a verification problem. It is expected that our approach will provide a more general and efficient framework for the verification of concurrent programs.

Using this environment, we have successfully verified some finite-state CCS processes automatically such as the ticking clock, the vending machine, 2-process mutual exclusion, etc. We have also verified some finite examples in the imperative and concurrent language such as several algorithms for mutual exclusion, the dining philosophers problem. For infinite state problems, we have verified an n-process mutual exclusion problem by integrating LegoMC with inductive reasoning. The same example has also been verified by combining LegoMC with abstraction method. We have also verified an endless counter by semantics with the help of LegoMC to simplify parts of the proof process. The counter example has also been verified by composition method with the assistance of LegoMC.

The Major Contribution

The major contribution of this thesis can be summarized in three parts as follows.

1. The formalisation of a verification environment
   - The formalisation of μ-calculus, CCS and an imperative and concurrent language.
   - The formal proving of related axioms, inference rules, lemmas and theorems.
2. The implementation of LegoMC
   - A domain specific interface
   - Automatic proof terms generation
3. The integration of various verification techniques

- The formal proving of axioms and inference rules of various verification systems.
- The exploration of the integration between various verification techniques.

1.6 Structure of the Thesis

This thesis is divided into four parts. The first part gives an overall introduction and the preliminary of this thesis. Part II introduces the specification language and description languages and their formalisation in Lego. In part III, LegoMC and some examples are presented. Part IV discusses the automation issues and future research.

The present chapter gives an overview, background information, specific problems to tackle and our approaches.

Chapter 2 gives the preliminary materials used in the thesis. We introduces some concepts in type theory UTT [Luo94] including higher order logic and inductive data type which are related to our work. We also introduce some Lego notions. As examples of formalisation in Lego, we formalize set theory and fixed points which will be used to the formalisation of μ-calculus presented in chapter 3.

Chapter 3 introduces temporal logics, which we will use to specify systems. We presents the syntax and semantics of μ-calculus and its formalisation in Lego. Two sets of inference rules for finite-state systems and infinite-state systems respectively are then formally proved in Lego. There is an implicit premise about monotonicity in those inference rules. We developed a monotonicity prover to prove the monotonicity automatically. Following μ-calculus, three temporal logics, PLTL, CTL and CTL* are introduced and finally their embeddings in μ-

calculus are discussed.

Chapter 4 presents CCS. The syntax and semantics of CCS are introduced. Subsequently their formalisation in Lego is described. Several lemmas and equivalence rules, which are formally proved in Lego, are then presented. Finally, a simple example is given to explain how verification can be done with the formalisation directly.

Chapter 5 presents an imperative and concurrent programming language (ICPL). We describe the syntax of ICPL and its formalisaion in Lego. We then define the transitional semantics of ICPL and present its formalisation. Finally an example is given.

Chapter 6 presents LegoMC. After a brief introduction to model-checking, the structure of LegoMC is then described. The implementation is then discussed. We also present the user interface of LegoMC. Two examples for CCS and ICPL respectively are then used to demonstrate the verification process of LegoMC.

Chapter 7 presents two examples with finite-state space. The first example is a simple communicating protocol. We model it in both CCS and ICPL and then use LegoMC to verify the desired properties. This example shows the comparison of verification on CCS and ICPL and demonstrates the process of improving a system design by LegoMC. The second example is a class of mutual exclusion algorithms. Since we use ICPL as the description language and use LegoMC to do verification automatically, we can easily formally verify all of those mutual exclusion algorithms. This example shows how easily to use LegoMC to analyse and compare a group of similar algorithms.

Chapter 8 presents infinite-state space examples. Two examples are given to demonstrate the verification by semantics and induction. The first example is an infinite counter, which have an evolving structure. The second example is a token-ring network which has many identical workstations connected in a network. The introduction of compositional method follows and the counter example is re-verified by compositional method. The abstraction technique is presented and we re-do the verification of token-ring example by abstraction. Finally, some discussion is given.

Chapter 9 gives a general discussion about automatic proof term generation in Type Theory. A general introduction to proof terms has been given in chapter 2. This chapter focuses on automation related issues. A general presentation about the construction of proof terms for assertions is given. We then discuss the automatic methods to construct proof terms. Some efficiency issues are discussed. Finally, some remarks are given.

Other automation issues are discussed in chapter 10. Many decision procedures for data domain are based on equational rewriting. Therefore, We first discuss the equality in type theory and equational rewriting techniques based on proof term generation. *Binary decision diagram* (BDD) technique is then discussed which has been claimed as an efficient technique to manipulate boolean expressions or propositional formulas. Finally, *arithmetic decision procedures* are discussed which can deal with arithmetic operation of natural numbers.

Finally, the conclusion is drawn and some areas for further research are mentioned in chapter 11.

Chapter 2

Preliminary

We use Lego proof checker as the platform to implement the verification environment. Two important features in Lego help us to formalize systems very easily, higher order logic and inductive data type. The expressive higher order logic simplifies the encoding of temporal logics and several concepts such as set, relation, predicate and fixed points. The inductive data type is quite useful to formalize data types.

Lego is based on type theory UTT [Luo94]. Type theory offers a coherent treatment of two related but different fundamental notions in computer science: computation and logical inference so that one can program and prove at the same time. Therefore, type theory may be used as a uniform language for programming, specification and reasoning. It also has nice abstraction and modularisation mechanism so that one can develop programs in the large as in the small and allow direct operational understanding and easy implement on the computer.

This chapter presents the concepts of higher order logic and inductive data type in UTT and introduces some Lego notions. As examples of formalisation in Lego, we also formalize set theory and fixed points which will be used to the formalisation of μ-calculus presented in chapter 3.

2.1 Inductive Data Type

Objects and *types* are two basic concepts in type theory. The relationship between objects and types is represented by the judgements of the form

$$a : A$$

that asserts that "object a is of type A." Some objects of a type are called *canonical objects*, which are the values of objects of the type under computation. A canonical object cannot be further computed and has itself as value. An object a being of type A means that a computes into a canonical object of type A.

A new type is defined by the *formation rules*, *introduction rules*, *elimination* and *computation rules*. The formation rules define what the types are and the introduction rules determine what the canonical objects are. The elimination rules determine how one can use a correctly asserted judgement $a : A$ to assert other judgements by introducing a recursive operator. The computation rules determine the computational meaning of the recursive operator by specifying how computation performs when the recursive operator is applied to a canonical object.

For example, an inductive type N of natural numbers can be introduced by the following rules:
the formation rule

- N is a type.

the introduction rule

1. 0 is of type N.
2. If n is of type N, so is $suc(n)$.

the elimination rule introduces a recursion operator Rec_N such that

- for any family of types $C[x]$ indexed by natural numbers x, any object of $C[0]$, and any function f that returns an object $f(m,c)$ of type $C[suc(m)]$ for any objects m of type N and c of type $C[m]$, $Rec_N(c,f)$ is a function which for any natural number n returns an object of type $C[n]$.

The computation rules give the computational meaning of Rec_N as follows:

1. $Rec_N(c,f)(0)$ computes to c.

2. $Rec_N(c,f)(suc(n))$ computes to $f(n, Rec_N(c,f)(n))$.

Lego has implemented a mechanism to simplify the definition of inductive data types which is presented in section 2.3.

Computation

One of the major features of type theory is that it is itself a computational language which enables us to do computing and logical reasoning in the same platform. The notion of computation and computational equality are basic concepts in type theory, which are captured by *reduction* and *conversion*, respectively. The computation rules, such as β-reduction, can be regarded as expressing certain schemata of definitions, and computation may be regarded as evaluation of defined function when applied to its arguments.

An important property of the computation in type theory, called *strong normalisation property*, is that:

> *Every well-typed term is strongly normalisable (i.e. every computation starting from a well-typed term terminates).*

The computation in type theory is very useful for automatic proof generation techniques which is presented in chapter 9.

2.2 Internal Higher Order Logic

Based on the principle of propositions-as-types and viewing logical formulas as types, the notion of formula in the internal logic is given by the notion of proposition. The usual logical operators and propositional equality can be defined using the universal quantifier, represented by the type constructor Π. Their definitions are briefly listed as follows and refer to [Luo94] for details.

$$
\begin{aligned}
\forall x : A.P(x) &=_{df} \Pi x : A.P(x) \\
P_1 \supset P_2 &=_{df} \forall x : P_1.P_2 \\
\mathbf{true} &=_{df} \forall X : Prop.X \supset X \\
\mathbf{false} &=_{df} \forall X : Prop.X \\
P_1 \wedge P_2 &=_{df} \forall X : Prop(P_1 \supset P_2 \supset X) \supset X \\
P_1 \vee P_2 &=_{df} \forall X : Prop.(P_1 \supset X) \supset (P_2 \supset X) \supset X \\
\neg P_1 &=_{df} P_1 \supset false \\
\exists x : A.P(x) &=_{df} \forall X : Prop.(\forall x : A.(P(x) \supset X)) \supset X \\
a =_A b &=_{df} \forall P : A \to Prop.P(a) \supset P(b)
\end{aligned}
$$

The internal higher order logic is very convenient to be used to encode other logic concepts. We have used higher order logic to code set theory, fixed point and μ-calculus. A formula is provable if and only if it is inhabited by some proof which is called the *proof term*.

Proof Terms

Proof terms are λ terms which are the proof objects in type theory. Logical formulas or propositions and logical inference in type theory are achieved by the idea of *propositions–as–types*, discovered by Curry [CF58] and Howard [How80]. This idea states that any proposition P corresponds to a type $\mathbf{Prf}(P)$, the type of its proofs, and a proof of P corresponds to an object of type $\mathbf{Prf}(P)$. To assert that a proposition is true, one have to find (construct) a proof object of the proposition. For example, the conjunction

$$P_1 \wedge P_2$$

has its type $\mathbf{Prf}(P_1 \wedge P_2)$ as

$$\forall X : Prop.(P_1 \supset P_2 \supset X) \to X.$$

One of its proof objects is

$$\lambda X : Prop \lambda h : P_1 \to P_2 \to X.\ h\ a\ b$$

where a is a proof object of P_1 and b a proof object of P_2.

Since every object in type theory can be computed into a unique canonical object, an object being a proof of a proposition P means that it computes into a canonical proof of P as well. A proposition P is true if and only if there is a canonical proof of it. To determine whether a given proof is indeed a proof of a given proposition is decidable and can be checked by type checking algorithms. Therefore, the proofs can be checked rigorously by computers to ensure the correctness of proofs. The details of proof checking and correctness issues are discussed in the next sub-section.

Therefore, the proof task for a proposition P is to find or construct a proof term which can compute into a canonical proof of P. Although they can compute into a unique value, the proof terms for a proposition can have different forms. Find a proof term of a proposition is not easy in general and therefore some interactive theorem provers are developed to guide users to construct proof terms. The details of proof term construction are given in chapter 9.

Proof Checking

There are two issues about proof terms that are essential for verification. One is the correctness of the proof, another one is the consistency of the integration between various verification techniques. Before discussing the correctness of proof terms, we first discuss some concepts about "proof".

"Proofs" are the evidences used to convince others that an augment is correct. To accept a proof depends on one's confidence on the evidences. There are basically three forms of "proof" in computer aided theorem proving: truth value, proof derivations and proof terms. Truth value is the answer produced by automatic reasoning tool, which will be an answer of "true" or "false" according their build-in algorithms. People have to believe the algorithm and its implementation to accept the "proof". Proof derivations are the reasoning sequence from axioms and primitive inference rules to conclusions. People accept the "proof" by checking the correctness of the reasoning sequence. HOL's proofs belong to this category and the correctness of their proof derivations can be checked by ML's type checking mechanism.

The proofs in type theory are proof terms which have their intended propositions as their types. The correctness of proofs can be checked by type checking algorithms. Because the correctness of the final generated proof term can always be type checked rigorously by computers, we can implement more efficient algorithms for automatic generation of proof terms or more sophisticated interface for interactive generation of proof terms without losing the correctness of proof terms.

Another important issue is the consistency of the integration of different verification techniques. To be able to verify real life systems, it is necessary to apply various verification techniques and integrate them together. One problem for the integration is usually the consistency between different techniques. Because they have different assumptions, axioms and inferences rules, it is very easy to have inconsistency between them. PVS has implemented many automatic techniques in the same platform and they claim they have to be very careful to integrate those techniques to maintain their consistency.

Our implementation does not have this problem. We formally encode the semantics of a verification system by the internal higher order logic of Lego. The axioms and inference rules are formally proved in Lego. Different axioms and inference rules from different verification systems have their own proof objects. The integration is therefore combining those proof objects in certain ways and the final proofs to original verification problems are bigger proof terms of which the correctness can be type checked.

In summary, we can say the correctness of our verification result is ensured by type checking algorithm. Since assertion $p : P$ means proof object p is an object of proposition P, the correctness of p can be checked by the type checking algorithm to see if p indeed has type P. Given any term and any context, the type checking algorithm checks whether the term is well typed in the context, and if so, it computes the principal type of the term in the context.

2.3 Lego Notations

This section introduces some Lego notations and terminology which will be used in the thesis.

Type Universes

`Prop`, Type(n) ($n \geq 0$ is a natural number), and `Type` are called *type universes*. The type universes constitute a hierarchy of type structure as follows:

$$Prop : Type(0) : Type(1) : \ldots.$$

That is, `Prop` is of type `Type(0)`, `Type(0)` is of type `Type(1)`, ...etc.. Any term which has a universe as its type is called a *type*. When a term has type `Prop`, it is called a *proposition*. `Type` is the type of all types and is the type of itself.

Π-types

`{x:A}B` is the notation for dependent Π-types ($\Pi x : A.B$) in type theory. `{x:A}B` is often written as `A->B` when the identifier `x` does not occur free in `B`. When `{x:A}B` is a proposition, it intuitively stands for "$\forall x : A.B$". If both `A` and `B` are propositions, the proposition `A->B` intuitively means "A implies B. When `{x:A}B` is not a proposition, it intuitively denotes a class of function f with domain A such that for $a : A$, $f(a)$ is of type $B[a/x]$, which is the term obtained by substituting all free occurrences of x by a in B. If neither `A` nor `B` is a proposition, `A->B` intuitively stands for the function type from A to B.

λ-abstraction and Pairs

`[x:A]M` is the notation for λ-form ($\lambda x : A.M$). `[x:A]M` intuitively denotes a function which returns the value of $M[a/x]$ for $a : A$. We can write `[_:A]M` when `x` does not appear in `M`.

`M N` intuitively denotes the result of *function application* of function M to value N. We use `N.M` to abbreviate `(M N)`.

`[x|A]M` is used to define polymorphic functions `[x:A]M`. When Lego can deduce the appropriate types x from M, we can omit an argument x while applying this function to terms.

`a,b` is the form of a pair in Lego. The type of `a,b` is `A#B` where `A` and `B` are types of `a` and `b` respectively.

Declarations, Definitions and Contexts

A *declaration* `[x:M]` declares that x is of type M.

A *definition* `[c C = M : A]` defines c under the contexts (see below) C to be M with type A where A is optional. Suppose C is of the form

$$b1\ b2\ \ldots\ bn$$

where bi is either a declaration or a definition. Then `[c C = M : A]` is equivalent to

$$[\mathtt{c} = \mathtt{b1\ b2\ \ldots\ bn(M:A)}]$$

Contexts are (possibly empty) sequences of declarations and definitions.

Inductive Data Type in Lego

There is an `Inductive` command in Lego [Pol94] to simplify the declaration of inductive types and relations by automatically constructing the basic Lego syntax from a 'high level' presentation. The syntax is as follows.

```
Inductive [T1:M1] ... [Tm:Mm]
Constructors [CONS1:L1] ... [CONSn:Ln]
<Options>
```

This command declares the mutually recursive data type `T1 ... Tm` with the constructors `CONS1 ... CONSn` which have corresponding types `L1 ... Ln`. There are several `Options` for declaring a recursive data type. Lego will automatically generate some corresponding theorems and recursive operators. Lego has built in many inductive data types as libraries. We list some inductive data types and logical definitions which are used in this thesis.

Natural Numbers

```
Inductive [nat:Type(0)]
Constructors [zero:nat][suc:nat->nat];

[nat_rec = [T|Type]nat_elim ([_:nat]t)
             : {T|Type}T->(nat->T->T)->nat->T ]
[nat_iter = [T|Type][x:T][f:T->T]nat_rec x ([_:nat]f)
             : {T|Type}T->(T->T)->nat->T ]
[plus = [m,n:nat] nat_iter n suc m   : nat->nat->nat ]
[times = [m,n:nat] nat_iter zero (plus n) m   : nat->nat->nat ]
[pred = nat_rec zero [n,_:nat]n (* truncated pred *)  : nat->nat ]
[minus = [m,n:nat] nat_iter m pred n   : nat->nat->nat ]
```

where `nat_elim` is the recursive operator of natural numbers, which is generated by Lego.

Booleans

```
Inductive [bool: type(0)]
Constructors [true: bool][false:bool];
```

```
[bool_rec = [T|Type]bool_elim ([_:bool]T)
            : {T|Type}T->T->bool->T ]
[if = [t|Type][b:bool][tCase,fCase:t] bool_rec tCase fCase b
            : {t|Type}bool->t->t->t ]
[andalso = [a,b:bool] if a b false    : bool->bool->bool ]
[orelse =  [a,b:bool] if a true b      : bool->bool->bool ]
[inv = [u:bool] if u false true         : bool->bool ]
[implies = [a,b:bool] if b true (if a false true)
            : bool->bool->bool ]
[is_true = [b:bool] Eq b true          : bool->Prop ];
[is_false = [b:bool] Eq b false        : bool->Prop ];
```

where `bool_elim` is the recursive operator of booleans, which is generated by Lego.

Lists

```
Inductive [list:Type(0)] Parameters [t:Type(0)]
Constructors [nil:list][cons1:t->list->list];

[cons [t|Type(0)][x:t][l:list t]= cons1 t x l];
```

where `Parameters` is one of options of inductive data type which is used to give parameters for inductive data type.

Some Logical Definitions

```
[A,B,C,D,E,F,G|Prop]
```

```
[T,S,U,V,W,X|Type(0)];
[a:A][b:B][c:C][d:D][e:E][f:F][g:G];
[trueProp = {P:Prop}P->P    : Prop ]
[Id = [t:T]t                : T->T ]
[PropId = [a:A]a            : A->A ]
[absurd = {A:Prop}A         : Prop ]
[not = [A:Prop]A->absurd    : Prop->Prop ];
[and = [A,B:Prop]{C|Prop}(A->B->C)->C    : Prop->Prop->Prop];
[or = [A,B:Prop]{C|Prop}(A->C)->(B->C)->C : Prop->Prop->Prop];
[pair = [C|Prop][h:A->B->C]h a b      : and A B ];
[inl = [C|Prop][h:A->C][_:B->C]h a    : or A B ]
[inr = [C|Prop][_:A->C][h:B->C]h b    : or A B ];
[fst = [h:and A B]h ([g:A][_:B]g)     : (and A B)->A ]
[snd = [h:and A B]h ([_:A][g:B]g)     : (and A B)->B ];
[iff = [A,B:Prop]and (A->B) (B->A)    : Prop->Prop->Prop]
[All = [P:T->Prop]{x:T}P x            : (T->Prop)->Prop ]
[Ex = [P:T->Prop]{B:Prop}({t:T}(P t)->B)->B  : (T->Prop)->Prop]
[ExIntro = [wit:T][P:T->Prop][prf:P wit][B:Prop][gen:{t:T}
                  (P t)->B]gen wit prf
          : {wit:T}{P:T->Prop}(P wit)->Ex P ]
[ExElim = [P|T->Prop][M:Ex P][N|Prop][prf:{t:T}(P t)->N]M N prf
          : {P|T->Prop}(Ex P)->{N|Prop}({t:T}(P t)->N)->N];
```

2.4 Set Theory

Set theory is a very useful concept which has many application in various area. In Lego, the objects in the universe are categorised into different types and there is no overlap between distinct types. Therefore, the set theory we use is *typed* *set theory* We use logical predicates to represent sets. Therefore, in the following discussion, the notation of sets will be $A.pred$ which means a set with elements of type A.

First, we define predicate operators as follow.

$$\begin{aligned}
Pred &= \lambda A : Type(0).A \to Prop \\
True &= \lambda A : Type(0)\lambda s : A\ trueProp : A.Pred \\
False &= \lambda A : Type(0)\lambda s : A\ absurd : A.Pred \\
Imply &= \lambda A : Type(0)\lambda C, D : A.Pred\lambda s : A\ s.C \to s.D : A.Pred \\
Iff &= \lambda A : Type(0)\lambda C, D : A.Pred\lambda s : A \\
&\quad and(Imply\ C\ D\ s)(Imply\ D\ C\ s) : A.Pred \\
And &= \lambda A : Type(0)\lambda C, D : A.Pred\lambda s : A\ and\ s.C\ s.D : A.Pred \\
Or &= \lambda A : Type(0)\lambda C, D : A.Pred\lambda s : A\ or\ s.C\ s.D : A.Pred \\
Not &= \lambda A : Type(0)\lambda C : A.Pred\lambda s : A\ not\ s.C : A.Pred
\end{aligned}$$

Then, the various concepts of sets can be defined over the predicate operators as follows.

$$\begin{aligned}
Fullset &= True \\
Emptyset &= False \\
Union &= Or \\
Meet &= And \\
Minus &= \lambda A : Type(0)\lambda C, D : A.Pred\ And\ C\ (Not\ D) \\
Singl &= \lambda A : Type(0)\lambda x : A\ Eq\ x : A.Pred \\
Subset &= \lambda A : Type(0)\lambda C, D : A.Pred\forall x : A\ x.C \to x.D \\
Eqset &= \lambda A : Type(0)\lambda C, D : A.Pred\ and\ (Subset\ C\ D)(Subset\ D\ C)
\end{aligned}$$

We can therefore prove some properties of set operators as follow.

- $union\_assoc : \forall A : Type(0) \forall C, D, E : A.Pred$
  $Eqset(Union\ C(Union\ D\ E))(Union(Union\ C\ D)E)$
- $singl\_lemma : \forall A : Type(0) \forall x, s : A \forall C : A.Pred$
  $(not(Eq\ x\ s)) \to x.(Union\ C\ s.Singl) \to x.C$
- $union\_lemma1 : \forall A : Type(0) \forall B, C : A.Pred$
  $Subset\ B\ (Union\ B\ C)$
- $minus\_lemma1 : \forall A : Type(0) \forall B, C, D : A.Pred$
  $(Subset\ B\ C) \to Subset\ (Minus\ B\ D)C$
- $minus\_lemma2 : \forall A : Type(0) \forall B, C : A.Pred$
  $Subset\ (Minus\ B\ C)\ B$
- $minus\_union\_lemma : \forall A : Type(0) \forall B, C, D : A.Pred\ \forall s : A$
  $(Subset\ (Minus\ B\ C)\ D) \to Subset\ (Minus\ B(Union\ C(s.Singl)))\ D$

2.5 Fix Points

The theorems of fixed points are very useful in computer science, giving semantics of programming languages, program analysis, program verification, etc. In this section, I list some formalisation and theorems which are formally proved in Lego.

Definition 2.5.1 (Prefixed point and postfixed point)

A subset $S \subseteq E$ is a *prefixed point* of Φ if

$$\Phi(S) \subseteq S$$

and a *postfixed point* of Φ if

$$S \subseteq \Phi(S).$$

S is a fixed point of Φ if S is both a prefixed point and postfixed point of Φ.

Theorem 2.5.1 (Tarski [Tar55]) *Let E be a set, $P(E)$ be the power set of E and $\Phi : P(E) \to P(E)$ be a monotonic function i.e.*

$$S \subseteq S' \to \Phi(S) \subseteq \Phi(S')$$

for all $S, S' \in P(E)$. Then Φ has a minimum fixed point $\mu S.\Phi(S)$ and a maximum fixed point $\nu S.\Phi(S)$ given by

$$\mu S.\Phi(S) = \cap\{S' \subseteq E | \Phi(S') \subseteq S'\}$$

$$\nu S.\Phi(S) = \cup\{S' \subseteq E | S' \subseteq \Phi(S')\}$$

$\mu S.\Phi(S)$ is the *least prefixed point* since it is the meet of all the prefixed point. $\nu S.\Phi(S)$ is the *greatest postfixed point* since it is the union of all the postfixed points.

We have formally proved many theorems of fixed point in Lego. The set is defined as a predicate over a data type A. Therefore for a monotonic function F, we can define its prefixed point and postfixed point as

```
[prefp [F:A.Pred->A.Pred][P:A.Pred] = Subset P.F P];
[postfp [F:A.Pred->A.Pred][P:A.Pred] = Subset P P.F];
```

We can then define its least fixed point as

```
[lfp [F:A.Pred->A.Pred] = [x:A]{P:A.Pred}(prefp F P)->x.P:A.Pred]
```

And prove the following theorems of least fixed point.

Theorem 2.5.2 *For every prefixed point P, least fixed point is a subset of P.*

$$\forall P.prefp(F, P) \to lfp(F) \subseteq P$$

Theorem 2.5.3 *Least fixed point is a prefixed point.*

$$prefp(F, lfp(F))$$

Theorem 2.5.4 *Least fixed point is a postfixed point.*

$$postfp(F, lfp(F))$$

The greatest fixed point can be defined as

```
[gfp [F:A.Pred->A.Pred] = [x:A]Ex [P:A.Pred]and(P.Subset P.F)(x.P)
                          : A.Pred]
```

Theorem 2.5.5 *Every postfixed point P is a subset of greatest fixed point.*

$$\forall P. postfp(F, P) \to P \subseteq gfp(F)$$

Theorem 2.5.6 *Greatest fixed point is a postfixed point.*

$$postfp(F, gfp(F))$$

Theorem 2.5.7 *Greatest fix point is a prefixed point.*

$$prefp(F, gfp(F))$$

Using the above formalisation, we can prove in Lego the following lemma and theorems, which will be used to prove model-checking rules presented in chapter 3.

Theorem 2.5.8 (Reduction lemma [Koz83, Win89])

$$\forall P. P \subseteq gfp(F) \leftrightarrow P \subseteq F(gfp(\lambda Q.(P \cup F(Q)))).$$

Theorem 2.5.9 (Least fix point fold and unfold)

$$\forall P.P \subseteq gfp(F) \leftrightarrow P \subseteq F(gfp(F) \cup P)$$

Theorem 2.5.10 (Greatest fix point base)

$$\forall P.P \subseteq P' \rightarrow P \subseteq gfp(\lambda Q.(P' \cup F(Q))).$$

Theorem 2.5.11 (Greatest fix point fold and unfold)

$$\forall P.P \subseteq gfp(F) \leftrightarrow P \subseteq F(gfp(F) \cup P)$$

Part II

Formalisation

Chapter 3

Temporal Logics

This chapter presents our formalisation of temporal logics in Lego. Temporal logic is a special branch of modal logic that deals with the truth values of assertions which change over time. Whereas an ordinary logic is adequate for describing a static situation, temporal logic enables us to discuss how a situation changes due to the passage of time. An execution of a program is precisely a chain of situations, called *execution states*. That suggests that temporal logic is the appropriate tool for reasoning about the execution of programs.

Concurrent programs have long been a difficult subject to formalise and have often been dealt with by the methods that worked perfectly for sequential programs. Temporal logic offers special advantages for the formalisation and analysis of the behaviour of concurrent programs since it is designed to reason about the on-going behaviour as sequences of actions or state changes.

Temporal logics can be classified as *linear temporal logic* and *branching temporal logic*. The first one regards the sequences of time as linear: at each moment there is only one possible future moment. The other one is that time has a branching, tree-like nature: at each moment, time may split into alternate sequences representing different possible futures. Both approaches have been applied to program reasoning, and it is a matter of debate as to say whether branching or linear time is preferable [EH86, Lam80, Pnu85].

Kozen's (propositional) modal μ-calculus (μK) [Koz83] has expressive power subsuming many modal and temporal logics such as LTL and CTL [BCM+92, CGH94, EL85]. Therefore, it is a natural choice to use μ-calculus to specify the properties. However, it can be difficult to express properties in μ-calculus since its semantics is not natural in terms of people's understanding. One way to tackle this problem is to give the translations from various perhaps more easily accessible temporal logics [EL85, Dam90, Sti92].

In the next section the syntax and semantics of μ-calculus are presented. Their formalisation in Lego is then presented in section 3.2. Subsequently two proof systems for finite and infinite state models are introduced. The inference rules of both systems have been formally proved in Lego. There is an implicit premise about monotonicity in those inference rules. We developed a monotonicity prover to prover the monotonicity automatically. Following μ-calculus, three temporal logics, PLTL, CTL and CTL* are introduced and finally their embeddings in μ-calculus are discussed.

3.1 μ-calculus

Because the double negation rule in classic logic does not exist in intuitionistic logic such as type theory, we have to adapt μ-calculus to positive version which does not contain negation operators. Theoretically all formulas with negation operators can be transformed to some kinds of normal forms with negation operators only occuring before atomic formulas [Wal95], therefore positive μ-calculus should be enough to express all the temporal properties we need. We also use Winskel's construction of *tagging* fixed points [Win89] to simplify the formalisation of inference rules.

The assertions are constructed from the following grammar:

$$\Phi \ ::= \ A \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle K \rangle \Phi \mid [K]\Phi | \mu Z.U\Phi \mid \nu Z.U\Phi$$

where U is called *tag* which is a subset of states, A ranges over atomic assertions, and K ranges over subsets of labels. The tag-free fixed points $\mu Z.\Phi$ and $\nu Z.\Phi$ are special cases with empty tag.

Semantics

The operational semantics of μ-calculus formulas is given via a *labeled transition system*

$$(S, L, \{\xrightarrow{t} : t \in L\})$$

which consists of a set S of *states*, a set L of *transition labels*, and a *transition relation* $\xrightarrow{t} \subseteq S \times S$ for each $t \in L$. We shall use the notation $s \xrightarrow{\alpha} s'$ for $\xrightarrow{\alpha} (s, s')$ and use $s \xrightarrow{K} s'$ to abbreviate $\exists \alpha \in K.s \xrightarrow{\alpha} s'$. The semantics of assertions $[\![\Phi]\!]_\rho \subseteq S$ is given by induction on the structure of Φ as follows.

$$\begin{array}{lcl}
[\![A]\!]_\rho & = & \rho(A) \\
[\![\Phi_1 \vee \Phi_2]\!]_\rho & = & [\![\Phi_1]\!]_\rho \cup [\![\Phi_2]\!]_\rho \\
[\![\Phi_1 \wedge \Phi_2]\!]_\rho & = & [\![\Phi_1]\!]_\rho \cap [\![\Phi_2]\!]_\rho \\
[\![\langle K \rangle \Phi]\!]_\rho & = & \{s \in S | \exists s' \in S.s \xrightarrow{K} s' \text{ and } s' \in [\![\Phi]\!]_\rho\} \\
[\![[K]\Phi]\!]_\rho & = & \{s \in S | \forall s' \in S.s \xrightarrow{K} s' \text{ implies } s' \in [\![\Phi]\!]_\rho\} \\
[\![\nu Z.U\Phi]\!]_\rho & = & \{s \in S | \exists P \subseteq S.P \subseteq [\![\Phi[P/Z]]\!]_\rho \cup U \text{ and } s \in P\} \\
[\![\mu Z.U\Phi]\!]_\rho & = & \{s \in S | \forall P \subseteq S. [\![\Phi[P/Z]]\!]_\rho \backslash U \subseteq P \text{ implies } s \in P\}
\end{array}$$

where the map ρ is an evaluation function which assigns to each atomic assertion A a subset of S. In the following discussion, We shall omit ρ when the evaluation function for atomic assertions is not a major concern. $\Phi[P/Z]$ is the substitution of Z by P in Φ. Satisfaction between a state s and an assertion Φ is now defined by: $s \models_\rho \Phi$ iff $s \in [\![\Phi]\!]_\rho$.

Some Useful Assertions

Some other useful assertions can be abbreviated as follows.

- tt $=_{def}$ $\nu Z.Z$, true formula
- ff $=_{def}$ $\mu Z.Z$, false formula
- able K $=_{def}$ $\langle K \rangle$tt, a capability for performing the labels in K
- inable K $=_{def}$ $[K]$ff, an inability to perform the labels in K
- $[-]\Phi =_{def} [L]\Phi, \langle - \rangle \Phi =_{def} \langle L \rangle \Phi$
- $[-K]\Phi =_{def} [L \backslash K]\Phi, \langle -K \rangle \Phi =_{def} \langle L \backslash K \rangle \Phi$
- deadlock $=_{def}$ $[-]$ff, cannot perform any labels
- deadlockfree $=_{def}$ $\nu Z.([-]Z \wedge \langle - \rangle \mathrm{tt})$, always can perform some labels.
- only K $=_{def}$ $\langle - \rangle$tt$\wedge[-K]$ff, only K can be performed

In section 3.4 we will find the other temporal logic properties can be defined in μ-calculus as well.

3.2 Formalisation of μ-calculus

Modality

First of all, we formalise the label sets of [ ] and $\langle\ \rangle$ operators as an inductive data type *Modality*. The modality type has two constructors, *Modal* and *Neg-*

modal, which correspond to the positive operator [K] and negative operator [-K], respectively.

```
Inductive [Modality:Type(0)] ElimOver Type
Constructors [Modal:(list Label)->Modality]
             [Negmodal:(list Label)->Modality];
```

In this formalisation, we use lists to represent finite sets of labels. To prevent doing induction in proving the membership of the finite set, we define a computational function `Modal_check(`a`,`M`)` to check whether label a is in modality M and return a corresponding boolean value.

Modal_check(a, Modal K) = member(a, K)
Modal_check(a, Negmodal K) = not (member(a, K))

where function `member(a,K)` checks whether `a` is a member of list `K` and return a boolean value.

Therefore, we can define the transition relation $MTrans(K, s, s')$ which represents $s \stackrel{K}{\rightarrow} s'$ as follows.

$$MTrans(K, s, s') = \exists a : Label.\ \text{Eq}\ Modal\_check(a, K)\ \text{true}\ \wedge Trans(a, s, s')$$

where $Trans(a, s, s')$ represents $s \stackrel{a}{\rightarrow} s'$.

μ-calculus

In our previous paper [YL97], we formalise the syntax of μ-calculus by an inductive data type and use de Bruijn index to deal with variable binding Since the variables which are denoted by natural numbers will change when doing variable substitution, it is very complicated to do reasoning. We therefore re-define the syntax of μ-calculus by encoding the semantics in Lego's internal higher order logic. Using the notation of set defined in previous chapter, the set of tags `Tag` and the set of μ-calculus formulas `Form` are defined as subsets of states `State.Pred`. The μ-calculus operators are defined as follows.

$$
\begin{aligned}
&Or(A, B : Form) = Union(A, B)\\
&And(A, B : Form) = Meet(A, B)\\
&Dia(K : Modality, P : Form)\\
&\qquad\qquad = \lambda s : State.\exists s' : State.and(MTrans(K, s, s'))s'.P\\
&Box(K : Modality, P : Form)\\
&\qquad\qquad = \lambda s : State.\forall s' : State(MTransKss') \to s'.P\\
&Tnu(T : Tag, F : Form \to Form)\\
&\qquad\qquad = \lambda s : State.\exists P : Form.and(P.Subset((FP).UnionT))s.P\\
&Tmu(T : Tag, F : Form \to Form)\\
&\qquad\qquad = \lambda s : State.\forall P : Form.(((FP).MinusT).SubsetP) \to s.P
\end{aligned}
$$

Using the above formalisation of syntax and semantics, we have proved in Lego the inference rules and the lemmas, nu_base, nu_unfold, mu_unfold, lemma_box and lemma_dia as introduced in the next section. Furthermore, it is easier to extend with more operators simply by encoding them as Lego propositions as well.

3.3 Inference Rules

Finite-State Systems

This section presents a sound and complete proof system for the μ-calculus to reason about finite-state systems. The judgements take the form

$$s \vdash \Phi,$$

which means that property Φ is satisfied at state s. We have formally proved the soundness in Lego and we believe the completeness can be proved in a way similar to [BS92]. The rules are presented in natural deduction style as follows.

$$\frac{s \in \rho(A)}{s \vdash A} \quad \text{(A is an atomic assertion)}$$

$$\frac{s \vdash \Phi}{s \vdash \Phi \vee \Psi} \qquad \frac{s \vdash \Psi}{s \vdash \Phi \vee \Psi}$$

$$\frac{s \vdash \Phi \quad s \vdash \Psi}{s \vdash \Phi \wedge \Psi}$$

$$\frac{s' \vdash \Phi}{s \vdash \langle K \rangle \Phi}(a \in K \text{ and } s' \in \{s' | s \xrightarrow{a} s'\})$$

$$\frac{s_1 \vdash \Phi, \ldots, s_n \vdash \Phi}{s \vdash [K]\Phi}(a \in K \text{ and } \{s_1, \ldots, s_n\} = \{s' | s \xrightarrow{a} s'\})$$

$$\frac{s \in U}{s \vdash \nu Z.U\Phi} \qquad \frac{s \vdash \Phi[\nu Z.(U \cup \{s\})\Phi/Z]}{s \vdash \nu Z.U\Phi}(s \notin U)$$

$$\frac{s \in U}{s \not\vdash \mu Z.U\Phi} \qquad \frac{s \vdash \Phi[\mu Z.(U \cup \{s\})\Phi/Z]}{s \vdash \mu Z.U\Phi}(s \notin U)$$

For [ ] and $\langle\ \rangle$ operators, to simplify the reasoning, we defined two functions `Succ` and `Filter`. `Succ(`s`)` generates a list of successor (label-state) pairs of a state s. `Filter(`K`, slist)` filters the states with corresponding labels in the Modality K from `slist` which is the output of `Succ(`s`)`. We can then prove lemma_dia and lemma_box as follows.

lemma_dia

$$\frac{s' \vdash \Phi}{s \vdash \langle K \rangle \Phi}(s' \in \text{Filter } K \text{ (Succ } s))$$

lemma_box

$$\frac{s_1 \vdash \Phi, \ldots, s_n \vdash \Phi}{s \vdash [K]\Phi}(\{s_1, \ldots, s_n\} = \text{Filter } K \text{ (Succ } s))$$

Because `Succ` function is used to get a finite list of successor states, these two lemmas can only be used to systems with a finite-branching structure.

We have also proved the following useful lemmas.

lemma_True

$$\forall s.s \vdash tt$$

lemma_False

$$\forall s.s \not\vdash ff$$

Infinite State Systems

We have also formally proved in Lego a sound proof system for reasoning about infinite-state systems which is adapted from [BS92] for tagged μ-calculus. The judgement is defined as

$$\varepsilon \vdash \Phi \quad \text{iff} \quad \forall s \in \varepsilon . s \vdash \Phi$$

where ε is a set of states and s is a state. The inference rules are as follows.

$$\frac{\varepsilon \subseteq \rho(A)}{\varepsilon \vdash A} \quad \text{A is an atomic assertion}$$

$$\frac{\varepsilon \vdash \Phi \quad \varepsilon \vdash \Psi}{\varepsilon \vdash \Phi \wedge \Psi}$$

$$\frac{\varepsilon_1 \vdash \Phi \quad \varepsilon_2 \vdash \Psi}{\varepsilon \vdash \Phi \vee \Psi}(\varepsilon = \varepsilon_1 \cup \varepsilon_2)$$

$$\frac{\varepsilon' \vdash \Phi}{\varepsilon \vdash \langle K\rangle\Phi}(\varepsilon \subseteq \{s \in S | \exists s' \in \varepsilon' \exists a \in K . s \xrightarrow{a} s'\})$$

$$\frac{(\varepsilon \xrightarrow{K}) \vdash \Phi}{\varepsilon \vdash [K]\Phi}((\varepsilon \xrightarrow{K}) = \{s' | \exists s \in \varepsilon \exists a \in K . s \xrightarrow{a} s'\})$$

$$\frac{\varepsilon \subseteq U}{\varepsilon \vdash \nu Z.U\Phi} \quad \frac{\varepsilon \vdash \Phi[\nu Z.(U \cup \varepsilon)\Phi/Z]}{\varepsilon \vdash \nu Z.U\Phi}(\varepsilon \not\subseteq U)$$

$$\frac{\varepsilon \subseteq U}{\varepsilon \not\vdash \mu Z.U\Phi} \quad \frac{\varepsilon \vdash \Phi[\mu Z.(U \cup \varepsilon)\Phi/Z]}{\varepsilon \vdash \mu Z.U\Phi}(\varepsilon \not\subseteq U)$$

$$\frac{\varepsilon' \cup \varepsilon \vdash \Phi}{\varepsilon \vdash \Phi}$$

where $(\varepsilon \xrightarrow{K})$ denotes the subset of states that can be reached through an action in K from a state in ε.

Monotonicity

The above inference rules for ν and μ constructors have the implicit monotonicity premise that all the functors which are defined by means of the μ-calculus operators are monotonic. The monotonicity of functors for predicates over A is defined is defined as follows:

$$Mono(F : A.Pred \to A.Pred) = \forall C, D : A.Pred(C \subseteq D) \to (F(C) \subseteq F(D))$$

To complete the rigorous proof in Lego, we cannot simply ignore that premise. Fortunately, the proof can be automated by proving the following rules for individual constructor of μ-calculus and developing an algorithm which can apply those rules to prove the monotonicity automatically.

- $Mono\_triv : \forall F.Mono\ \lambda Z.F$(Z is not bound in F)
- $Mono\_Var : Mono\ \lambda Z.Z$
- $Mono\_And1 : \forall F \forall Q (Mono\ F) \to Mono\ (\lambda Z.(F Z) \wedge Q)$
- $Mono\_And2 : \forall F \forall Q (Mono\ F) \to Mono\ (\lambda Z.Q \wedge (F Z))$

- $Mono\_And : \forall F \forall G (Mono\ F) \to (Mono\ G) \to Mono(\lambda Z.(FZ) \wedge (GZ))$

- $Mono\_Or : \forall F \forall G (Mono\ F) \to (Mono\ G) \to Mono(\lambda Z.(FZ) \vee (GZ))$

- $Mono\_Box : \forall F \forall K (Mono\ F) \to Mono\ (\lambda Z.[K](FZ))$

- $Mono\_Dia : \forall F \forall K (Mono\ F) \to Mono\ (\lambda Z.\langle M \rangle (FZ))$

- $Mono\_Nu : \forall F \forall T (\forall X.Mono\ FX) \to Mono\ (\lambda Z.\nu Y.T(FYZ))$

- $Mono\_Mu : \forall F \forall T (\forall X.Mono\ FX) \to Mono\ (\lambda Z.\mu Y.T(FYZ))$

3.4 Embedding of Temporal Logics

The above formalisation is expressive enough for us to reason about various temporal properties. However, the μ-calculus is not natural to capture people's understanding of properties. Therefore, it is better to define other temporal logics as abbreviations of μ-calculus.

Propositional Linear Temporal Logic

The *Propositional Linear Temporal Logic* (PLTL) is one of linear temporal logics advocated by Manna and Pneuli [MP92]. The basic temporal operators in PLTL are $\Diamond\Phi$ ("eventually Φ"), $\Box\Phi$ ("always Φ"), $\bigcirc\Phi$ ("next Φ") and $\Phi U \Psi$ ("Φ until Ψ"). The syntax of PLTL formulas can be defined by the following grammar.

$$\Phi ::= A | \top | \Phi \wedge \Phi | \neg \Phi | \bigcirc \Phi | \Phi U \Phi$$

where A ranges over atomic formulas. The other formulas can then be introduced as abbreviations as follows.

$$\begin{aligned} \Diamond\Phi &= \top U \Phi \\ \Box\Phi &= \neg\Diamond\neg\Phi \end{aligned}$$

The semantics of a formula Φ of PLTL can be defined with respect to a linear-time structure $M = (S, x, \rho)$ where S is a set of states, $x = s_0, s_1, s_2, \ldots = x(0), x(1), x(2), \ldots$ is an infinite sequence of states and ρ is an evaluation function from atomic formulas A to a subset of S. We write $x \models \Phi$ to mean that Φ is true in terms of timeline x. x^i denotes the suffix path $s_i, s_{i+1}, s_{i+2}, \ldots$. The semantics is defined as follows.

$$
\begin{array}{rcl}
x \models A & \text{iff} & x(0) \in \rho(A) \\
x \models \Phi \wedge \Psi & \text{iff} & x \models \Phi \text{ and } x \models \Psi \\
x \models \neg\Phi & \text{iff} & x \not\models \Phi \\
x \models \Phi U \Psi & \text{iff} & \exists i . x^i \models \Psi \text{ and } \forall j . j < i \text{ implies } x^j \models \Phi \\
x \models \bigcirc\Phi & \text{iff} & x^1 \models \Phi
\end{array}
$$

Computation Tree Logic

CTL [CE81, CES86] is a branching time temporal logic which allows basic temporal operators of a path quantifier, either **A**("for all") or **E**("exist"), followed by a single one of the usual linear temporal operators **G**("always"), **F**("sometime"), **X**("next"), or **U**("until"). Giving one set of atomic formulas A, the formulas Φ of the language CTL can be defined by the following grammar:

$$\Phi ::= A | \top | \neg\Phi | \Phi \wedge \Phi | \mathbf{AX}\Phi | \mathbf{EX}\Phi | \mathbf{A}(\Phi\mathbf{U}\Phi) | \mathbf{E}(\Phi\mathbf{U}\Phi)$$

We also introduce the following syntactic abbreviations:

$$
\begin{array}{rcl}
\mathbf{AF}\Phi & = & \mathbf{A}\top\mathbf{U}\Phi \\
\mathbf{EF}\Phi & = & \mathbf{E}\top\mathbf{U}\Phi \\
\mathbf{AG}\Phi & = & \neg\mathbf{EF}\neg\Phi \\
\mathbf{EG}\Phi & = & \neg\mathbf{AF}\neg\Phi
\end{array}
$$

Giving a transition system $M = (S, \rightarrow, \rho)$, a *satisfaction* relation $\models$ between states of M and formulas of CTL can be defined by induction over the structure of the formulas. The notation $s \models \Phi$ means that state s satisfies formula Φ. Here S is the set of states, $\rightarrow$ is a total binary relation $\subseteq S \times S$, and ρ is an evaluation function taking all atomic formulae A to subset of S.

$$
\begin{array}{rcl}
s \models A & \text{iff} & s \in \rho(A) \\
s \models \top & \text{iff} & s \in S \\
s \models \neg\Phi & \text{iff} & s \not\models \Phi \\
s \models \Phi \wedge \Psi & \text{iff} & s \models \Phi \text{ and } s \models \Psi \\
s \models \mathbf{AX}\Phi & \text{iff} & \forall s'.s \rightarrow s' \text{ implies } s' \models \Phi \\
s \models \mathbf{EX}\Phi & \text{iff} & \exists s'.s \rightarrow s' \text{ and } s' \models \Phi \\
s_0 \models \mathbf{A}(\Phi\mathbf{U}\Psi) & \text{iff} & \forall \text{ paths } s_0, s_1, \ldots, \exists i \geq 0.s_i \models \Psi \text{ and } \forall j < i.s_i \models \Phi \\
s_0 \models \mathbf{E}(\Phi\mathbf{U}\Psi) & \text{iff} & \exists \text{a path } s_0, s_1, \ldots, \exists i \geq 0.s_i \models \Psi \text{ and } \forall j < i.s_i \models \Phi
\end{array}
$$

CTL^*

CTL severely restricts the type of formula that can appear after a path quantifier, i.e. only single linear time operator, **F**, **G**, **X**, or **U** can follow a path quantifier. By distinguishing two types of formulas: state formulas and path formulas, CTL^* permits an arbitrary formula of linear time logic to follow a path quantifier. It was proposed as an unifying framework subsuming both CTL and PLTL [EH86].

$$\text{state–formula}(\Phi) ::= A|\Phi \wedge \Phi|\neg\Phi|\mathbf{E}\Psi$$

$$\text{path–formula}(\Psi) ::= \Phi|\top|\Psi \wedge \Psi|\neg\Psi|\mathbf{X}\Psi|\Psi\mathbf{U}\Psi$$

where A ranges over atomic assertions. The other operators are defined as syntax abbreviation as follows.

$$\mathbf{A}\Phi \quad = \quad \neg\mathbf{E}\neg\Phi$$

$$
\begin{aligned}
\mathbf{F}\Psi &= \top\mathbf{U}\Psi \\
\mathbf{G}\Psi &= \neg\mathbf{F}\neg\Psi
\end{aligned}
$$

Note that if we define path formulas as $\mathbf{X}\Phi$ and $\Phi\mathbf{U}\Psi$ only, the set of state formulas forms CTL. Also note that the set of path formulas yields PLTL.

Given a transition system $M = (S, \rightarrow, \rho)$ as defined above, a full path of it is an infinite sequence $s_0, s_1, s_2, \ldots$ of states such that $\forall i. s_i \rightarrow s_{i+1}$. We use $x = (s_0, s_1, s_2, \ldots)$ denotes a full path, $x(i)$ denotes s_i, and that x^i denotes the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. The notation $s \models \Phi$ means that state s satisfies formula Φ, $x \models \Psi$ means that full path x satisfies formula Ψ.

$$
\begin{array}{rll}
s \models A & \text{iff} & s \in \rho(A) \\
s \models \Phi_1 \wedge \Phi_2 & \text{iff} & s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s \models \neg\Phi & \text{iff} & s \not\models \Phi \\
s \models \mathbf{E}\Psi & \text{iff} & \exists x. x(0) = s \text{ and } x \models \Psi \\
x \models \Phi & \text{iff} & x(0) \models \Phi \\
x \models \Psi_1 \wedge \Psi_2 & \text{iff} & x \models \Psi_1 \text{ and } x \models \Psi_2 \\
x \models \neg\Psi & \text{iff} & x \not\models \Psi \\
x \models \Psi_1\mathbf{U}\Psi_2 & \text{iff} & \exists i. x^i \models \Psi_2 \text{ and } \forall j. j < i \text{ implies } x^j \models \Psi_1 \\
x \models \mathbf{X}\Psi & \text{iff} & x^1 \models \Psi
\end{array}
$$

Embedding

Since CTL* subsumes PLTL and CTL, it should be enough to embed CTL* in μ-calculus. To describe the embedding we need a weak version of diamond-operator $\langle - \rangle' \Phi = \langle - \rangle \Phi \vee [-]False$ and a strong version of box-operator $[-]' \Phi = [-]\Phi \wedge \langle - \rangle True$ which originate from [And93]. The embedding is presented in Table 3.1 where e.p.o.w should be read as "exists a path on which" and o.a.p

CTL*	μ-calculus	Meaning
$\mathbf{EX}\Phi$	$\langle - \rangle \Phi$	e.p.o.w Φ at next state
$\mathbf{AX}\Phi$	$[-]'\Phi$	o.a.p. Φ at next state
$\mathbf{EG}\Phi$	$\nu Z.\langle - \rangle' Z \wedge \Phi$	e.p.o.w always Φ
$\mathbf{AG}\Phi$	$\nu Z.[-] Z \wedge \Phi$	o.a.p. always Φ
$\mathbf{EF}\Phi$	$\mu Z.\langle - \rangle Z \vee \Phi$	e.p.o.w. eventually Φ
$\mathbf{AF}\Phi$	$\mu Z.[-]' Z \vee \Phi$	o.a.p. eventually Φ
$\mathbf{EGF}\Phi$	$\nu X.\mu Y.\langle - \rangle Y \vee (\langle - \rangle' X \wedge \Phi)$	e.p.o.w. infinitely often Φ
$\mathbf{AGF}\Phi$	$\mathbf{AG}(\mathbf{AF}\Phi)$	o.a.p. infinitely often Φ
$\mathbf{EFG}\Phi$	$\mathbf{EF}(\mathbf{EG}\Phi)$	e.p.o.w eventually always Φ
$\mathbf{AFG}\Phi$	$\mu X.\nu Y.[-] Y \wedge ([-]' X \vee \Phi)$	o.a.p. eventually always Φ

Table 3.1: Embedding of CTL* in μ-calculus

should be read as "on all paths". For the proof of such embedding the reader is referred to [EC80, Koz83, EL85, Dam90, Sti92].

In later discussion of this thesis, I shall use "always" to denote **AG**, "eventually" to denote **EF** and "next" to denote **X**.

Chapter 4

System Modeling and CCS

The usual way of modeling a system in most of model-checkers and interactive theorem provers is to use a labeled state-transition graph (explicit or implicit). The transition relation of a system can be defined as an inductive relation in theorem provers. Although the mechanism to define inductive data types in many theorem provers helps to reduce human effort significantly, it is very time-consuming and error-prone for a large system.

Another alternative approach is to formalise the syntax and semantics of a system description language and then use this system description language to describe systems. It is believed that this approach is more natural and easier to model systems and therefore gives a better user interface. Moreover, an interface with exactly the syntax of description language and specification language will further simplify the verification job.

We have formalised two description languages: CCS and an imperative and concurrent language (ICPL). This chapter gives a general introduction about concurrent systems and then presents CCS and our formalisation of CCS in Lego. ICPL is presented in next chapter.

4.1 Concurrent Systems

Concurrency can be represented by *interleaving* [MP92]. Therefore, a concurrent system can be regarded as a system in which there are several entities (called *agents*) in progress at the same time by *interleaved* execution sequences of the atomic instructions of sequential agents. Concurrent systems are different from sequential systems in at least two ways: agents compete for access to *shared resources* and they exchange *messages*. Therefore there are two general points of view of modeling concurrency: *shared variable model* and *message passing model*.

Shared variable model considers parallel agents of the form $P_1|P_2|\ldots|P_n$ consisting of a finite set of sequential agents $P_1, P_2, \ldots, P_n$ running together in parallel. There is an underlying set of *variables* $v_1, \ldots, v_m$ that are *shared* among the processes in order to provide for inter-process communication and co-ordination. We define a simple imperative and concurrent programming language to model shared variable systems.

Message passing model, on the other hand, has its own set of local variables $y_1, \ldots, y_n$ for each process that cannot be accessed by other processes. All inter-process communication is performed by *message passing primitives*. CSP [Hoa85] and CCS [Mil89] are best examples of this model. We use CCS to model message passing systems.

4.2 CCS: Calculus of Communicating System

In this thesis, we consider *pure* CCS, which does not involve value passing. The expressions of CCS, which are called *agents*, are used to model systems of communicating processes. A process uses *actions* to communicate with other processes where each action is associated with a *label*. There is a special action τ which models *idling* or *invisible* or *internal* actions. Let *Act* be a set of actions defined as follows.

1. τ : *internal* or *idling* action

2. a : base action

3. $\overline{a}$: complement action

where a ranges over the names of actions. The complement action has the property that $\overline{\overline{a}} = a$.

The expressions of CCS are defined as follows.

- Nil: empty agent, a process which cannot perform any actions

- X: agent variable

- α.E : prefix, a process which can only perform action α and thereafter behave as the process described by E.

- E_1+E_2 : summation, a process which behaves as either the process described by E_1 or as the process described by E_2.

- $E_1|E_2$: parallel composition, a process which consists of two process described by E_1 and E_2, which can have independent behaviours or communication through complement actions.

- E\L : restriction, a process behaves likes E but cannot perform any actions in L or their complement actions.

- E[f] : relabeling, a process behaves like E with its labels of actions renamed by function f

- rec X.E : recursion, a recursive process which behaves like the agent E with X substituted by rec X.E where α ranges over actions, E, E_1, E_2 range over agents, L is a subset of base actions, f is a relabeling function from Act to Act with $f(\overline{a}) = \overline{f(a)}$ and $f(\tau) = \tau$. The syntax can be summarized as the following grammar:

$$E ::= Nil \mid X \mid \alpha.E \mid E_1 + E_2 \mid E \backslash L \mid E[f] \mid E_1 | E_2 \mid recX.E$$

Formalisation

We use natural numbers to represent the base names of actions: `Base = nat` and then define the types of actions as follows.

```
Inductive [ActB : TYPE(0)] ElimOver Type
Constructors [base : Base->ActB][comp : Base->ActB];

Inductive [Act : TYPE(0)] ElimOver Type
Constructors [tau:Act][act : ActB->Act];
```

The notion of complement can then be defined as a function from $ActB$ to $ActB$ as follows.

```
Comp (base a) = comp a
Comp (comp a) = base a
```

We can then prove the property $\overline{\overline{a}} = a$ as $\forall a.Comp(Comp\ a) = a$ by doing inductive reasoning over the type $ActB$.

We use lists to represent sets and natural numbers to represent the process variables: `Var = nat` and then define the type of processes as follows.

```
Inductive [Process : TYPE(0)] ElimOver Type
Constructors
```

```
[Nil : Process]
[dot : Act->Process->Process]
[cho : Process->Process->Process]
[par : Process->Process->Process]
[hide: Process->(list ActB)->Process]
[ren : Process->(Base->Base)->Process]
[var : Var->Process]
[rec : Process->Process];
```

In the above, the natural way to express `rec` constructor should be

$$[\mathtt{rec} : (\mathtt{Process} -> \mathtt{Process}) -> \mathtt{Process}].$$

However, Lego does not allow this sort of expressions since in general they could introduce paradox [Luo94]. Instead, we use de Bruijn's index [dB72] to deal with variable binding. Since de Bruijn's index is complicated and difficult for general users to use and understand, we have implemented an interface in LegoMC where the user does not use de Bruijn's index, while the machine translates the user notation into de Bruijn's notation. LegoMC is discussed in chapter 6.

4.3 de Bruijn's Index

Instead of using names to express variable, the method of de Bruijn's index uses natural numbers which denote their reference depth (the number of λ between variables and their binders plus one). This representation avoids all the renaming problems associated with actual names (α conversion). For example, the following λ term,

$$\lambda x(x\lambda y(y\ x))$$

can be represented by de Bruijn's index as

$$\lambda(1\ \lambda(1\ 2)).$$

The distance of the first x with its binder is 1, whereas the second x is 2 because there is one λ between the second x with its binder λx. Although this method is very convenient for implementation, it is easy to confuse people since the same variables are represented by different numbers.

The *substitution* operation of λ terms expressed in de Bruijn's indexes needs the *weakening* operation. They are defined as follows.

- $weaken(n, x)$ adds one to the variables in term x which are bigger than n.
- $subst(x, n, A)$ replace the variables in term x which are equal to n with A. If x has the form λy, then A should be weaken by the depth of A, which is the maximum depth of variables in A.

`depth`, `weaken` and `subst` are defined as functions which are then used for the substitution operation in the formalisation of transitional relations of CCS described in the next section.

4.4 Semantics of CCS

The operational semantics of CCS agents is given via a *labeled transition system*

$$(S, L, \{\stackrel{t}{\rightarrow}: t \in L\})$$

which consists of a set S of *states*, a set L of *transition labels*, and a *transition relation* $\stackrel{t}{\rightarrow} \subseteq S \times S$ for each $t \in L$. In CCS, we shall take S to be E, the agent expressions, and L to be *Act*, the actions. The transition relations are given by the following transition rules in terms of the structure of agent expressions.

$$Prefix \quad \alpha.E \stackrel{\alpha}{\rightarrow} E$$

$$Sum \quad \frac{E_1 \stackrel{\alpha}{\rightarrow} E'}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E'} \quad \frac{E_2 \stackrel{\alpha}{\rightarrow} E'}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E'}$$

$$Par \quad \frac{E_1 \stackrel{\alpha}{\to} E'}{E_1|E_2 \stackrel{\alpha}{\to} E'|E_2} \quad \frac{E_2 \stackrel{\alpha}{\to} E'}{E_1|E_2 \stackrel{\alpha}{\to} E_1|E'}$$

$$Par(Com) \quad \frac{E_1 \stackrel{a}{\to} E_1' \quad E_2 \stackrel{\overline{a}}{\to} E_2'}{E_1|E_2 \stackrel{\tau}{\to} E_1'|E_2'}$$

$$Res \quad \frac{E \stackrel{a}{\to} E'}{E\backslash L \stackrel{a}{\to} E'\backslash L}(a, \overline{a} \notin L) \quad \frac{E \stackrel{\tau}{\to} E'}{E\backslash L \stackrel{\tau}{\to} E'\backslash L}$$

$$Rel \quad \frac{E \stackrel{\alpha}{\to} E'}{E[f] \stackrel{f(\alpha)}{\to} E'[f]}$$

$$Rec \quad \frac{E[(Rec\ X.E)/X] \stackrel{\alpha}{\to} E'}{Rec\ X.E \stackrel{\alpha}{\to} E'}$$

Whenever $E \stackrel{\alpha}{\to} E'$, we call the pair (α, E') an *immediate derivative* of E, α an *action* of E, and E' an α*-derivative* of E.

Formalisation

The transition relation can be defined as an inductive relation with each of the constructors in the definition corresponding to one or two rules as follows.

```
Inductive [TRANS : Act->Process->Process->Prop] Relation
Constructors
[Dot  : {a:Act}{p:Process}
(*-------------------------------------------------*)
TRANS a (dot a p) p
]
[ChoL : {a:Act}{p1,p2,p:Process}
(TRANS a p1 p)->
(*-------------------------------------------------*)
(TRANS a (cho p1 p2) p)
]
[ChoR : {a:Act}{p1,p2,p:Process}
```

```
(TRANS a p2 p)->
(*----------------------------------------------------*)
(TRANS a (cho p1 p2) p)
]
[ParL : {a:Act}{p1,p2,p:Process}
(TRANS a p1 p)->
(*----------------------------------------------------*)
(TRANS a (par p1 p2) (par p p2))
]
[ParR : {a:Act}{p1,p2,p:Process}
(TRANS a p2 p)->
(*----------------------------------------------------*)
(TRANS a (par p1 p2) (par p1 p))
]
[Tau1 : {n:Base}{p1,p2,q1,q2:Process}
(TRANS n.base.act p1 q1)->(TRANS n.comp.act p2 q2)->
(*----------------------------------------------------*)
(TRANS tau (par p1 p2) (par q1 q2))
]
[Tau2 : {n:Base}{p1,p2,q1,q2:Process}
(TRANS n.comp.act p1 q1)->(TRANS n.base.act p2 q2)->
(*----------------------------------------------------*)
(TRANS tau (par p1 p2) (par q1 q2))
]
[Hide : {a:ActB}{p,q:Process}{R:list ActB}
(TRANS a.act p q)->
(is_false (orelse(member a R)(member a.comple R)))->
(*----------------------------------------------------*)
(TRANS a.act (hide p R) (hide q R))
```

```
]
[Ren  : {a:Act}{p,q:Process}{f:Base->Base}
(TRANS a p q)->
(*--------------------------------------------------*)
(TRANS (rename f a) (ren p f) (ren q f))
]
[TauH : {p,q:Process}{R:list ActB}
(TRANS tau p q)->
(*--------------------------------------------------*)
(TRANS tau (hide p R) (hide q R))
]
[Rec  : {a:Act}{p,p':Process}
(TRANS a (subst p one p.rec) p')->
(*--------------------------------------------------*)
(TRANS a p.rec p')];
```

where `Relation` is one of options of inductive data type which is used to define inductive relation. `rename(f,a)` is a higher order function that will rename `a` by the mapping function `f`. `(subst p one p.rec)` is $p[(rec\ x.p)/x]$ represented by de Bruijn's index, which is described in previous section.

For instance, the constructor of rule $Dot : \alpha.P \stackrel{\alpha}{\to} P$ is

```
[Dot  : {a:Act}{p:Process}
(*--------------------------------------------------*)
TRANS a (dot a p) p
]
```

which means $\forall a \in Act \forall p \in Process$ (p is an a-derivative of $a.p$). The constructor of rule $ChoL : \frac{P_1 \stackrel{\alpha}{\to} P}{P_1 + P_2 \stackrel{\alpha}{\to} P}$ is

```
[ChoL : {a:Act}{p1,p2,p:Process}
(TRANS a p1 p)->
(*--------------------------------------------------*)
(TRANS a (cho p1 p2) p)
]
```

which means $\forall a \in Act\ \forall p, p1, p2 \in Process$ (if p is an a-derivative of $p1$, then p is an a-derivative of $p1 + p2$).

4.5 Lemmas and Theorems

Based on the above formalisation of CCS, we have formally proved the following *inversion lemmas* which are useful to reason about the transition relation. The inversion lemmas can be used to infer the premise from the conclusion under the inductive relation definition. For example, for the following inference rule

$$\frac{A}{B} \quad ,$$

the inversion lemma could be "if B is provable, then A should be provable".

lemma_nil	$Nil \not\rightarrow E$
lemma_nil'	$E \rightarrow E'$ implies $E \neq Nil$
lemma_dot	$a.E \xrightarrow{b} F$ implies $a = b$ and $E = F$
lemma_cho	$P_1 + P_2 \xrightarrow{a} P$ implies $P_1 \xrightarrow{a} P$ or $P_2 \xrightarrow{a} P$
lemma_par	$P_1 \vert P_2 \xrightarrow{a} P$ implies
	$(\exists P'.P_1 \xrightarrow{a} P' \vee P_2 \xrightarrow{a} P') \vee$
	(if $a = \tau$ then $\exists P_1', P_2', b.(P_1 \xrightarrow{b} P_1' \wedge P_2 \xrightarrow{\bar{b}} P_2') \vee (P_1 \xrightarrow{\bar{b}} P_1' \wedge P_2 \xrightarrow{b} P_2')$)
lemma_hide	$P\backslash K \xrightarrow{a} P'$ implies $\exists P_1'.(P' = P_1'\backslash K) \wedge$
	(if $a = act\ b$ then $(b \notin K) \wedge (P \xrightarrow{a} P_1')) \wedge$ (if $a = \tau$ then $P \xrightarrow{a} P_1'$)
lemma_rec	$rec\ X.P \xrightarrow{a} P'$ implies $P[rec\ X.P/X] \xrightarrow{a} P'$

4.6 Example: a Ticking Clock

I shall use an example to explain how the formalisation of CCS and temporal logics presented in previous chapter are used to do verification. This is the example of a ticking clock taken from [Sti92].

$$Cl = tick.Cl$$

The only action this process will perform is `tick` and it will `tick` forever. The CCS model of the clock in our formalisation is as follows.

```
[tick = act (base zero)];
[Cl = rec (tick.dot one.var)];
```

Here we define `tick` as action zero and define the clock by the expression with de Bruijn's index as `rec tick.1` which equals to CCS syntax `rec x tick.x`. We first try a simple property: the clock is able to `tick`.

Proof

$$Cl \vdash \langle tick \rangle tt$$

Refine by lemma_dia rule and instantiated by `Cl`, the goal becomes

$$Cl \vdash tt \tag{4.1}$$

$$Cl \in Filter(tick, Succ\, Cl) \tag{4.2}$$

Sub-goal 4.1 can be proved by lemma_True and sub-goal 4.2 can be proved by Member_head rule.

The second property is deadlock freedom which is $\nu Z.([-]Z \wedge \langle - \rangle tt)$.

Proof

$$Cl \vdash \nu Z.([-]Z \wedge \langle - \rangle tt)$$

Refine by nu_unfold and pair rules, we get the following two sub goals

$$Cl \vdash [-](\nu Z.\{Cl\}[-]Z \wedge \langle - \rangle tt) \tag{4.3}$$

$$Cl \vdash \langle - \rangle tt \tag{4.4}$$

Sub-goal 4.3 can be proved lemma_box and nu_base rules. Sub-goal 4.4 can be refined by lemma_dia and instantiated by `Cl` and get the following two sub-goals.

$$Cl \vdash tt \tag{4.5}$$

$$Cl \in Filter(-, Succ\, Cl) \tag{4.6}$$

Sub-goal 4.5 can be proved by lemma_True. Sub-goal 4.6 can be proved by Member_head rule.

Chapter 5

An Imperative and Concurrent Programming Language (ICPL)

It is believed that the modeling job will become pretty easy in the syntax of imperative programming languages compared with CCS or transitional graphs. Using ICPL to model a system has at least four advantages.

1. It can simplify the modeling job.
2. It can reduce the mistakes during modeling.
3. It is easier for programmers to use verification tools.
4. It can be used to verify real programs.

In the following section, we describe the syntax of ICPL and its formalisaion in Lego. We then define the transitional semantics of ICPL and present its formalisation in section 5.2. An example is given in section 5.3.

5.1 The Syntax

We consider a concurrent program as several sequential *processes* in progress at the same time by *interleaved* execution sequences of the primitive statements. There is an underlying set of global variables that are shared among the processes for inter-process communication and synchronization. Each set of global variables ranges over a data type and has an initial value before the program starts to execute. The syntax of our language can be described as follows, where $\mathcal{N}$ ranges over natural numbers, $\mathcal{V}$ ranges over natural number variables, $\mathcal{BE}$ ranges over boolean expressions and $\mathcal{NE}$ ranges over natural number expressions, **wait** and **signal** are semaphore statements.

1. natural number expressions

$$\mathcal{N} ::= 0 \mid 1 \mid 2 \mid \ldots.$$
$$\mathcal{NE} ::= \mathcal{N} \mid \mathcal{V} \mid \mathcal{NE} + \mathcal{NE} \mid \mathcal{NE} - \mathcal{NE} \mid \mathcal{NE} \times \mathcal{NE}$$

2. boolean expressions

$$\mathcal{BE} ::= \mathbf{true} \mid \mathbf{false} \mid \mathcal{BE}\&\&\mathcal{BE} \mid \mathcal{BE} \mid\mid \mathcal{BE} \mid \neg\, \mathcal{BE} \mid \mathcal{NE} == \mathcal{NE} \mid \mathcal{NE} < \mathcal{NE}$$

3. semaphores

$$Sem ::= \mathcal{V}$$

4. primitive statements

$$Primitive ::= \mathcal{V} := \mathcal{NE} \mid \mathbf{skip} \mid \mathbf{await}\ \mathcal{BE} \mid \mathbf{wait}\ Sem \mid \mathbf{signal}\ Sem$$

5. processes

$$\begin{aligned} Process ::= & Primitive \mid \textbf{Empty} \mid \textbf{If } \mathcal{BE} \textbf{ then } Process \textbf{ else } Process \\ & \mid \textbf{While } \mathcal{BE} \textbf{ do } Process \mid Process; Process \\ Program ::= & \; Process \quad \mid \quad Process | Program \end{aligned}$$

Formalisation

Each of the expression types is defined as an inductive data type, consisting of constants, variables and operators to construct expressions. The type of natural number expressions may be represented as the following inductive data type.

```
Inductive [NatExp : Type(0)] ElimOver Type
Constructors
[natConst : nat->NatExp]
[natVar : Var->NatExp]
[natAdd : NatExp->NatExp->NatExp]
[natMinus : NatExp->NatExp->NatExp]
[natTimes : NatExp->NatExp->NatExp];
```

where `nat` is the type of natural numbers, `Var` is a type of variables represented by `nat`. We shall use the following abbreviations.

```
[ONE = natConst zero.suc]
[ZERO = natConst zero]
```

The type of boolean expressions may be represented as the following inductive data type.

```
Inductive [BoolExp : Type(0)] ElimOver Type
```

```
Constructors
[boolConst : bool->BoolExp]
[boolAnd   : BoolExp->BoolExp->BoolExp]
[boolOr    : BoolExp->BoolExp->BoolExp]
[boolNot   : BoolExp->BoolExp]
[boolEq    : NatExp->NatExp->BoolExp]
[natLess   : NatExp->NatExp->BoolExp];
```

where `bool` is the type of boolean values (i.e. true and false).

Primitives can be defined as an inductive data type as well. The assignment is represented by a pair of variables and natural number expressions. We only allow assignment statements of natural number expressions at the moment. The semaphore is represented by variable type. The formalistion is as follows.

```
[Assignment= Var#NatExp];
[Semaphore = Var];

Inductive [Primitive : Type(0)]
Constructors
[assign : Assignment->Primitive]
[skip   : Primitive]
[await : BoolExp->Primitive]
[wait   : Semaphore->Primitive]
[signal : Semaphore->Primitive];
```

Labeled Processes

To provide a unique and convenient identification and reference to the positions of processes, we label processes with line numbers similar to [MP92]. We label all statements except *Comp* statement which does not need labels obviously. A *labeled program* is the program of which processes are all labeled. We use natural numbers to represent the line numbers: lno = nat. The formalisation of processes and programs are as follows.

```
Inductive [Process : Type(0)] ElimOver Type
Constructors [Prim : Primitive->Process]
             [Empty : Process]
             [If    : BoolExp->Process->Process->Process]
             [While: BoolExp->Process->Process]
             [Comp : Process->Process->Process];

Inductive [Program : Type(0)] ElimOver Type Double Theorems
Constructors [PROC : Process->Program]
             [PAR  : Program->Program->Program];
```

We shall use the following abbreviations.

```
(* abbreviations for processes *)
[Assign [x:Var][e:NatExp] = (x,e).assign.Prim];
[Skip = skip.Prim];
[Await [b:BoolExp] = b.await.Prim];
[Wait [S:Semaphore] = S.wait.Prim];
[Signal [S:Semaphore] = S.signal.Prim];

(* abbreviations for programs *)
[ASSIGN [x:Var][e:NatExp] = (x,e).assign.Prim.PROC];
[SKIP = skip.Prim.PROC];
[AWAIT [b:BoolExp] = b.await.Prim.PROC];
```

```
[WAIT [S:Semaphore] = S.wait.Prim.PROC];
[SIGNAL [S:Semaphore] = S.signal.Prim.PROC];
[IF [b:BoolExp][p1,p2:Process] = (If b p1 p2).PROC];
[WHILE [b:BoolExp][p:Process] = (While b p).PROC];
[COMP [p1,p2:Process] = (Comp p1 p2).PROC];
[EMPTY = Empty.PROC];
```

A function `process_label(p)` is defined to get the line number of a process p in a state. We label all the top statements first and then the statements under top statements. For example, here is a labeled program with two labeled processes.

```
p1 = 1: While true do
        2:   (While s==1 do
          5:      skip);
        3:   skip;   (* critical section *)
        4:   s := 1
p2 = 1: While true do
        2:   (While s==0 do
          5:      skip);
        3:   skip;   (* critical section *)
        4:   s := 0
```

Therefore we can express the mutual exclusion property as "There is not a state in which p1 is at position 3 and p2 is at position 3".

A function `th_process(k,P)` is defined to get the kth process of a program P.

5.2 Shared Memory and Transitional Semantics

We define a state as a pair (P, M), consisting of a labeled program P, which represents the labeled program text to be further executed, and a memory M. The memory is a table containing the current values of variables denoted as a list of (Variable, Value) pair. We shall use $M(e)$ to denote the value of e under evaluation in memory M and M_e^x to denote changing the value of x to $M(e)$ in memory M. Therefore, we can define the operational semantics of ICPL via a labeled transition system as follows, where the transition labels are primitive statements or boolean expressions and ϵ is the Empty statement and $\epsilon|p = p|\epsilon = p$.

$$\frac{}{(x := e, M) \stackrel{x:=e}{\longrightarrow} (\epsilon, M_e^x)} \quad \frac{}{(skip, M) \stackrel{skip}{\longrightarrow} (\epsilon, M)} \quad \frac{M(b) = true}{(await(b), M) \stackrel{await(b)}{\longrightarrow} (\epsilon, M)}$$

$$\frac{M(s) > 0}{(wait(s), M) \stackrel{wait(s)}{\longrightarrow} (\epsilon, M_{s-1}^s)} \quad \frac{}{(signal(s), M) \stackrel{signal(s)}{\longrightarrow} (\epsilon, M_{s+1}^s)}$$

$$\frac{M(b) = true}{(\text{if } b \text{ then } p_1 \text{ else } p_2, M) \stackrel{b}{\longrightarrow} (p_1, M)} \quad \frac{M(b) = false}{(\text{if } b \text{ then } p_1 \text{ else } p_2, M) \stackrel{\neg b}{\longrightarrow} (p_2, M)}$$

$$\frac{M(b) = true}{(\text{while } b \text{ do } p, M) \stackrel{b}{\longrightarrow} (p; \text{while } b \text{ do } p, M)} \quad \frac{M(b) = false}{(\text{while } b \text{ do } p, M) \stackrel{\neg b}{\longrightarrow} (\epsilon, M)}$$

$$\frac{(p_1, M) \stackrel{l}{\longrightarrow} (\epsilon, M')}{(p_1; p_2, M) \stackrel{l}{\longrightarrow} (p_2, M')} \quad \frac{(p_1, M) \stackrel{l}{\longrightarrow} (p, M')}{(p_1; p_2, M) \stackrel{l}{\longrightarrow} (p; p_2, M')} (p \neq \epsilon)$$

$$\frac{(p_1, M) \stackrel{l}{\longrightarrow} (p, M')}{(p_1|p_2, M) \stackrel{l}{\longrightarrow} (p|p_2, M')} \quad \frac{(p_2, M) \stackrel{l}{\longrightarrow} (p, M')}{(p_1|p_2, M) \stackrel{l}{\longrightarrow} (p_1|p, M')}$$

The Formalisation of Semantics

Memories can be defined as a function of type `Var`$\rightarrow$ `nat`. However, since the memory we will consider is always a finite set, we can use lists to represent memories to simplify the memory manipulation. Also it is easier to define the equivalence by lists. Therefore, memories are represented as a list of pairs of variables and values as follows.

```
[Memory = list (Var#nat)]
```

The evaluation of `NatExp` and `BoolExp` in memory, $M(e)$, are defined as function `natsEval:Memory→NatExp→nat` and `boolEval:Memory→BoolExp→bool` respectively. Another function `Change:Memory→Var→nat→Memory` is defined for changing the current value of variables, M_e^x.

The type of states, `state`, can be defined as a record type as follows.

```
Record [State : Type(0)]
Fields [program : Program]
       [memory : Memory]
```

Lego will generate a function `make_State:Program→Memory→State` which forms a state by taking a program and a memory. The equivalence of states then involves program equivalence and memory equivalence.

The labels are defined as follows.

```
Inductive [Label : Type(0)]
Constructors [bool_label : BoolExp->Label]
             [prim_label : Primitive->Label];
```

The transition relation `TRANS` can be defined as an inductive relation with each of the constructors in the definition corresponding to one or two rules as follows.

```
Inductive [TRANS : Label->State->State->Prop] Relation
Constructors
```

```
[ruleASSIGN :
 {M:Memory}{x:Var}{e:NatExp}
  TRANS (x,e).assign.prim_label
(make_State (ASSIGN x e) M)
(make_State EMPTY (Change M x (natsEval M e)))
]
[ruleSKIP :
 {M:Memory}
  TRANS skip.prim_label
            (make_State SKIP M)
           (make_State EMPTY M)
]
[ruleIF1 :
 {M:Memory}{b:BoolExp}{p1,p2:Process}
          (Eq (boolEval M b) true) ->
  (**************************************************)
    TRANS b.bool_label
            (make_State (IF b p1 p2) M)
            (make_State p1.PROC M)
]
[ruleIF2 :
 {M:Memory}{b:BoolExp}{p1,p2:Process}
          (Eq (boolEval M b) false) ->
  (**************************************************)
    TRANS b.bool_label
            (make_State (IF b p1 p2) M)
            (make_State p2.PROC  M)
]
[ruleWHILE1 :
```

```
 {M:Memory}{b:BoolExp}{p:Process}
          (Eq (boolEval M b) true) ->
  (************************************************)
    TRANS b.bool_label
            (make_State (WHILE b p) M)
            (make_State (COMP p (While b p)) M)
]
[ruleWHILE2 :
 {M:Memory}{b:BoolExp}{p:Process}
          (Eq (boolEval M b) false) ->
  (************************************************)
    TRANS b.bool_label
            (make_State (WHILE b p) M)
            (make_State EMPTY M)
]
[ruleAWAIT :
 {M:Memory}{b:BoolExp}
          (Eq (boolEval M b) true) ->
  (************************************************)
    TRANS b.await.prim_label
            (make_State (AWAIT b) M)
            (make_State EMPTY M)
]
[ruleCOMP1 :
 {M,M':Memory}{p1,p2:Process}{L:Label}
    (TRANS L
            (make_State p1.PROC M)
            (make_State EMPTY M')) ->
   (***********************************************)
```

```
     TRANS L
            (make_State (COMP p1 p2) M)
            (make_State p2.PROC  M')
]
[ruleCOMP2 :
 {M,M':Memory}{p,p1,p2:Process}{L:Label}
    (TRANS L
            (make_State p1.PROC M)
            (make_State p.PROC  M')) ->
   (**********************************************)
    TRANS L
            (make_State (COMP p1 p2) M)
            (make_State (COMP p  p2) M')
]
[ruleWAIT :
 {M:Memory}{S:Semaphore}
    (Lt zero (natsEval M S.natVar)) ->
  (*************************************************)
    TRANS S.wait.prim_label
            (make_State (WAIT S) M)
            (make_State EMPTY (Change M S (natsEval M S.natVar).pred))
]
[ruleSIGNAL :
 {M:Memory}{S:Semaphore}
  (*************************************************)
    TRANS S.signal.prim_label
            (make_State (SIGNAL S) M)
            (make_State EMPTY (Change M S (natsEval M S.natVar).suc))
]
```

```
[rulePAR1 :
 {M,M':Memory}{P,P1,P2:Program}{L:Label}
    (TRANS L
            (make_State P1 M)
            (make_State P  M')) ->
   (*******************************************)
     TRANS L
            (make_State (PAR P1 P2) M)
            (make_State (PAR P  P2) M')
]
[rulePAR2 :
 {M,M':Memory}{P,P1,P2:Program}{L:Label}
    (TRANS L
            (make_State P2 M)
            (make_State P  M')) ->
   (*******************************************)
     TRANS L
            (make_State (PAR P1 P2) M)
            (make_State (PAR P1 P ) M')
];
```

For instance, the constructor of rule $Assign : (x := e, M) \stackrel{x:=e}{\rightarrow} (\epsilon, M_e^x)$ is

```
[ruleASSIGN :
 {M:Memory}{x:Var}{e:NatExp}
  TRANS (x,e).assign.prim_label
        (make_State (ASSIGN x e) M)
        (make_State EMPTY (Change M x (natsEval M e)))
]
```

The constructor of rule $IF1 : \frac{M(b)=true}{(\text{if } b \text{ then } p_1 \text{ else } p_2, M) \xrightarrow{b} (p_1, M)}$ is

```
[ruleIF1 :
 {M:Memory}{b:BoolExp}{p1,p2:Process}
          (Eq (boolEval M b) true) ->
  (**********************************************)
    TRANS b.bool_label
          (make_State (IF b p1 p2) M)
          (make_State      p1.PROC M)
]
```

Atomic formulas

To reason about properties related to the position of programs and data-dependent properties, we introduce the atomic formulas and then give the following as semantics.

At(pno,lno) = λ s:State.

Eq process_label(th_process(pno, s.program)) lno

NotAt(pno,lno) = λ s:State.

not (Eq process_label (th_process(pno, s.program)) lno)

Bool(b) = λ s:State. Eq (boolEval(s.memory, b)) true

where *pno* is the process number and *lno* is the line number and s is the state which is a pair of memory and program. At(2, 3) means "process 2 at line 3" and NotAt(1, 4) means "process 1 not at line 4". *Bool* is used to specify the boolean properties such as "$a > 3$".

p1=	p2=
1: while true do	1: while true do
2: await turn == 0;	2: await turn == 1;
3: critical section;	3: critical section;
4: turn := 1	4: turn := 0
5: done;;	5: done;;

Figure 5.1: A solution for the mutual exclusion problem

5.3 Example - A Mutual Exclusion Algorithm

One of important properties of concurrent programs is *mutual exclusion* of critical sections. Typical critical sections involve access to non-sharable resources such as printers; only one process is allowed to access at every a single moment. Since there are many processes executing at the same time in a concurrent system which compete for resources, it is essential to have certain strategy to control the access of critical sections for processes. To ensure the correctness of those strategies, it is important to have those strategies formally verified. Mutual exclusion states that at most one process is allowed to be in its critical section at any time, that is, for all reachable states only one process in the critical section.

The above algorithm is a simple solution for mutual exclusion of a two-process system. There is a variable `turn` to control the access of the critical section. The statements in the above algorithm have their usual meanings except `await` which means "wait at that position until the boolean expression becomes `true`". The mutual exclusion property for this algorithm would be

> "In every reachable state, it is not true that p1 is at position 3 and p2 is at position 3."

Using the syntax presented in this section, the algorithm can then be described as follows.

```
[turn = one]; [turn' = natVar turn];
[p1 = While TRUE
             (((Await (boolEq turn' ZERO)).Comp
             Skip).Comp
            (Assign turn ONE))];
[p2 = While TRUE
            (((Await (boolEq turn' ONE)).Comp
            Skip).Comp
            (Assign turn ZERO))];


[Pro = (Proc p1).Par (Proc p2)];
[init = (one,one).singleton:Memory];
```

The program of 2-process is $Pro = [p1, p2]$. The initial value of variable $turn$ is 1, $init = [(turn, 1)]$. The mutual exclusion property is $me = AG((NotAt(1,3) \vee NotAt(2,3))$.

Proof

$$(init, Pro) \vdash \nu Z.[-]Z \wedge (NotAt(1,3) \vee NotAt(2,3))$$

The proof is represented as the following proof tree with P_{ij} represents process Pi at line j and `me` represents $(NotAt(1,3) \vee NotAt(2,3))$.

$$\dfrac{\dfrac{(init, Pro) \vdash \nu Z.[-]Z \wedge me}{(init, Pro) \vdash [-](\nu Z.\{(init, Pro)\}[-]Z \wedge me) \wedge me}}{\underline{(init, Pro) \vdash [-](\nu Z.\{(init, Pro)\}[-]Z \wedge me)} \quad (init, Pro) \vdash me}$$

$$(1) \qquad\qquad (2)$$

(1)

$$\cfrac{\cfrac{\overline{(init, P_{12}|P_{21}) \vdash (\nu Z.\{(init, Pro)\}[-]Z \wedge me)}}{(init, P_{12}|P_{21}) \vdash [-](\nu Z.\{(init, Pro), (init, P_{12}|P_{21})\}[-]Z \wedge me) \wedge me}}{\underline{(init, P_{12}|P_{21}) \vdash [-](\nu Z.\{(init, Pro), (init, P_{12}|P_{21})\}[-]Z \wedge me)} \quad (init, P_{12}|P_{21}) \vdash me}$$

$$(3) \qquad\qquad\qquad (4)$$

The node $(init, Pro) \vdash me$ is true because process 1 is at line 1 and process 2 is at line 2 in program Pro. The node $(init, P_{12}|P_{21}) \vdash me$ is true as well.

Most of the proof tree are similar to the above and therefore we can simply denote them as the following traveling tree with (i, j, k) representing the value of $turn$ is i, process one is at j and process two is at k.

$$\begin{array}{ccccccccccc}
(1,1,1) & \to & (1,1,2) & \to & (1,1,3) & \to & (1,1,4) & \to & (0,1,1) & \to & (0,1,2) \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
(1,2,1) & \to & (1,2,2) & \to & (1,2,3) & \to & (1,2,4) & \to & (0,2,1) & \to & (0,2,2) \\
 & & & & & & & & \downarrow & & \downarrow \\
 & & & & & & & & (0,3,1) & \to & (0,3,2) \\
 & & & & & & & & \downarrow & & \downarrow \\
 & & & & & & & & (0,4,1) & \to & (0,4,2) \\
 & & & & & & & & \downarrow & & \downarrow \\
 & & & & & & & & (1,1,1) & & (1,1,2)
\end{array}$$

The leaves in the proof tree are (1,1,1), (1,1,2) and nodes of me assertions. Nodes (1,1,1) and (1,1,2) have appeared before in the proof tree. Therefore, the states should be in the tag of assertions. We can then use nu_base rule to prove them. We can find $(0, 3, 3)$ and $(1, 3, 3)$ do not appear in the tree and therefore all the me assertion node are true.

Part III

LegoMC

Chapter 6

The Model-Checker, LegoMC

We can verify finite and infinite problems using the formalisation in Part II already. However, there are so many tedious and trivial proof steps; we expect to use model-checking to develop parts of the proofs automatically. We have implemented a model-checker called LegoMC in ML language. A domain specific interface is created so that the user can define their model and specification in the syntax that they are familiar with and then use simple commands to verify properties and generate proof terms. After a brief introduction to model-checking in the next section, the structure of LegoMC is described in the subsequent section. The implementation is then discussed in section 6.3. Section 6.4 presents the user interface of LegoMC. Two examples for CCS and ICPL respectively are then used to demonstrate the verification process of LegoMC.

6.1 Model-Checking

Over the last decade *model-checking* has emerged as a powerful technique for automatically verifying concurrent systems [CES86, VW86, Cle90, And92]. The basic idea is to determine whether or not a system satisfies a property typically expressed as a temporal logic formula by searching the state space of the system thoroughly. When systems have finite-state space, model-checking algorithms can be used to verify the system completely automatically.

Two major categories of model-checking algorithms have been developed: *global* and *local* model-checking. Global model-checking requires the *a priori* construction of the entire state space of the system being analyzed and then a subsequent pass over the state space determines the truth or falsity of the formula. Although exhibiting good worst-case behavior, in practice the overhead of computing the whole state space is unnecessary, as the answer can often be deduced from a small part of it. Local model-checking remedies this shortcoming by exploring the state space in *demand-drived* fashion but has poor worst-case behavior compared with global model-checking. We use local model-checking because it is easier to be formalised in theorem proving settings.

In contrast to model-checking, interactive theorem proving gives a general approach to modeling and verification of both hardware and software systems but requires significant human efforts to deal with many tedious proofs. Even a simple model like the 2-process mutual exclusion problem can be fairly complicated to verify. If we can adapt model-checking techniques into theorem proving settings, we should be able to simplify the verification dramatically.

The idea is to adapt a model-checking algorithm to generate proof terms for finite-state system verification. This model-checker should be able to verify finite-state systems completely automatically. We can also use Lego to decompose a large system (could be infinite) to several smaller sub-systems. Among those smaller sub-systems, the model-checker can be used to generate proof terms for them if they have finite-state spaces. The proof terms from each part of sub-systems can then be integrated to complete the whole proof. LegoMC is implemented in functional language ML with two versions, one for CCS and another one for imperative and concurrent programming language (ICPL). The Overall system structure of LegoMC is described in the next section.

6.2 System Structure and Inference Rules

LegoMC is an independent program with the user interface in the syntax of CCS or the imperative language and propositional μ-calculus. The system can be modeled using CCS or the imperative language and the properties can be expressed using μ-calculus. LegoMC will compute answers of whether a system satisfies certain properties and return proof terms in the syntax of Lego if the system does satisfy the properties. The proof terms can then be integrated with other proof terms to complete a larger proof. Other temporal logics such as LTL and CTL are defined as the abbreviations of μ-calculus. The system structure is shown in Fig. 6.1.

Given as input the definition of a finite model and a specification (formula) in the syntax described in section 6.4, LegoMC will produce the proof term in Lego syntax which could be put into Lego to complete a larger proof if the model satisfies the specification. If the model does not satisfy the specification, LegoMC simply produces an error message. CCS, ICPL and μ-calculus are formalised in Lego and defined in ML programs.

Inference Rules

One difficulty for adapting model-checking algorithms into type theory based theorem proving settings is that almost all the model-checking techniques are based on classic logic and therefore we have to change the inference rules and algorithms to positive forms (without negation operators). The model-checking algorithm is based on several inference rules for finite-state systems which are formally proved in Lego as chapter 3 and repeated as follows.

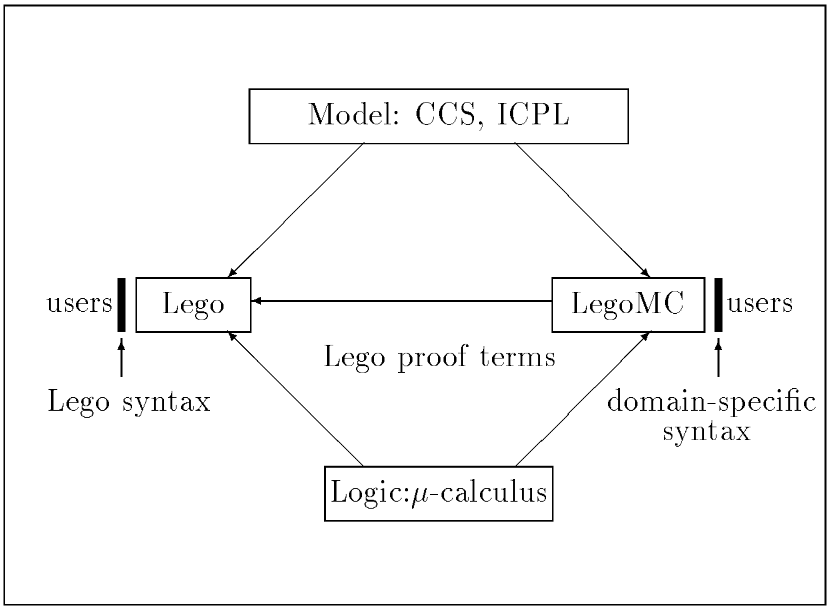

Figure 6.1: The system structure of LegoMC

$$\frac{s \in \rho(A)}{s \vdash A} \quad \text{(A is an atomic assertion)}$$

$$\frac{s \vdash \Phi}{s \vdash \Phi \vee \Psi} \qquad \frac{s \vdash \Psi}{s \vdash \Phi \vee \Psi}$$

$$\frac{s \vdash \Phi \quad s \vdash \Psi}{s \vdash \Phi \wedge \Psi}$$

$$\frac{s' \vdash \Phi}{s \vdash \langle K \rangle \Phi}(a \in K \text{ and } s' \in \{s' | s \xrightarrow{a} s'\})$$

$$\frac{s_1 \vdash \Phi, \ldots, s_n \vdash \Phi}{s \vdash [K]\Phi}(a \in K \text{ and } \{s_1, \ldots, s_n\} = \{s' | s \xrightarrow{a} s'\})$$

$$nu\_base : \frac{s \in U}{s \vdash \nu Z.U\Phi} \quad nu\_unfold : \frac{s \vdash \Phi[\nu Z.(U \cup \{s\})\Phi/Z]}{s \vdash \nu Z.U\Phi}(s \notin U)$$

$$mu\_base : \frac{s \in U}{s \not\vdash \mu Z.U\Phi} \quad mu\_unfold : \frac{s \vdash \Phi[\mu Z.(U \cup \{s\})\Phi/Z]}{s \vdash \mu Z.U\Phi}(s \notin U)$$

For [ ] and $\langle\ \rangle$ operators, to simplify the reasoning, we defined two functions `Succ` and `Filter`. `Succ`(s) generates a list of successor (label-state) pairs of a state s. `Filter`(K, `slist`) filters the states with corresponding labels in the

Modality K from `slist` which is the output of `Succ(`s`)`. We can then prove lemma_dia and lemma_box as follows.

lemma_dia

$$\frac{s' \vdash \Phi}{s \vdash \langle K \rangle \Phi}(s' \in \text{Filter } K \text{ (Succ } s))$$

lemma_box

$$\frac{s_1 \vdash \Phi, \ldots, s_n \vdash \Phi}{s \vdash [K]\Phi}(\{s_1, \ldots, s_n\} = \text{Filter } K \text{ (Succ } s))$$

Proof Terms

The rules for generating proof terms, which use the above inference rules, are described as follows, where $p : P(s)$ means p is a proof term of predicate P on state s and

$$\frac{p : P(s)}{q : Q(s)}$$

means if p is a proof term of $P(s)$ then q is a proof term of $Q(s)$.

OR

$$\frac{p : P}{\text{inl } p\ q : P \vee Q} \quad \frac{q : Q}{\text{inr } p\ q : P \vee Q}$$

AND

$$\frac{p : P \qquad q : Q}{\text{pair } p\ q : P \wedge Q}$$

BOX

$$\frac{p_1 : \Phi(s_1), \ldots, p_n : \Phi(s_n)}{\text{lemma\_box prove\_state\_list} : [K]\Phi(s)}(\{s_1, \ldots, s_n\} = \text{Filter K (Succ s))}$$

where prove_state_list=$[s'$:State]mem_ind $p_1 \ldots$ mem_ind p_n(not_mem_nil s')

DIA

$$\frac{p' : P(s')}{\text{lemma\_dia (pair prove\_member } p') : \langle K \rangle P(s)} (s' \in \text{Filter K (Succ s)})$$

where prove_member= (Member_tail ... Member_tail Member_head)

NU

$$\frac{}{\text{nu\_base} : \nu Z.U\Phi(s)} (s \in U) \quad \frac{p : \Phi[\nu Z.U \cup \{s\}\Phi/Z](s)}{\text{nu\_unfold } p : \nu Z.U\Phi(s)} (s \notin U)$$

MU

$$\frac{p : \Phi[\mu Z.U \cup \{s\}\Phi/Z](s)}{\text{mu\_unfold } p : \mu Z.U\Phi(s)} (s \notin U)$$

In the above rules, *inr* and *inl* are the or-introduction proof operators, *pair* is for and-introduction, *ExIntro* for exists-introduction, *mem_ind* for the membership induction rule and *not_mem_nil* for the rule that no element is the member of an empty set, *Member_tail* for the rule that an element is a member of the tail of a list, *Member_head* for the rule that an element is the head of a list.

Atomic formulas

Besides the above regular μ-calculus properties, there are also properties which are specific to the description languages such as *At* in ICPL for describing the location counters of processes. The proof terms for those properties vary for different properties and different description languages. One way of simplifying proof term generation for atomic formulas is using computational functions.

For example, At property is defined as

$$At(pno, lno) = \lambda s : State.Eq\ (process\_label\ (th\_process\ pno\ s.program))\ lno$$

where pno is process number, lno is line number and $process\_label$ and $th\_process$ are computational functions. $(process\_label\ (th\_process\ pno\ s.program))$ will compute to a natural number, the same type as lno. Therefore the proof term is simply "$Eq\_refl\ lno$". This example also demonstrates how computational functions can simplify an external program as described in chapter 9.

6.3 The Implementation

We have implemented LegoMC as a separate program in ML. The entities of CCS, ICPL and μ-calculus are defined as inductive data types which are similar to the formalisation of Lego presented in part II. It is obvious very inconvenient and error prone for users to use de Bruijn's indexes. Therefore, an interface with CCS syntax is defined and translation mechanism is implemented to translate terms in CCS syntax into the internal terms with de Bruijn's indexes.

Proof Term Generation

The proof term generation of LegoMC uses "Separating search from justification" technique [Bou93]. The basic idea is dividing proof construction into two passes, one for proof search and another one for proof term generation. The motivation for this technique is to avoid contract proof terms for unsuccessful branches of a search. The first pass is implemented as a function `Check` which only returns a boolean value "true" or "false". The second pass is another function `Prove` which then performs the task of proof term generation. **OR** and **AND** operators are quite straightforward. The other operators are described as follows. Note: the exact syntax of the proof term has been changed to make them more understandable.

DIA

Assume we want to find a proof term p of $\langle K\rangle P(s)$. We check individual state in `Filter K (Succ s)` until we find a state s' with the proof term p' of $P(s')$. If we can find s', then p is "lemma_dia (pair (Member_tail ... Member_tail Member_head) p'".

BOX

Assume we want to find a proof term p of $[K]P(s)$. We check all the states in `Filter K (Succ s)`. If all the states $s_1, \ldots, s_n$ with the corresponding proof term $p_1, \ldots, p_n$ of $P(s_1), \ldots, P(s_n)$ then p is "lemma_box [s':State]mem_ind p_1... mem_ind p_n(not_mem_nil s')".

NU

Assume we want to find a proof term p of $\nu Z.U\Phi(s)$, we check whether $s \in U$ first. If $s \in U$, the proof term p is "nu_base". If $s \notin U$, we try to find the proof term p' of $\Phi[\nu Z.U \cup \{s\}\Phi/Z(s)]$. If we can find p', then p is "nu_unfold p'".

MU

Assume we want to find a proof term p of $\mu Z.U\Phi(s)$, we check whether $s \in U$ first. If $s \in U$, the proof fails and we return "false". If $s \notin U$, we try to find the proof term p' of $\Phi[\mu Z.U \cup \{s\}\Phi/Z(s)]$. If we can find p', then p is "mu_unfold p'".

6.4 User Interface

This section presents the syntax of user interface for users to model their systems in CCS or ICPL and define the temporal properties they want to verify.

6.4.1 CCS

LegoMC uses the following syntactic conventions for CCS processes.

- Identifiers: consist of sequences of printable characters, where the first character must be a letter. There are some characters which cannot be used as part of an identifier. These are: `.,=()[]{}\/+|'` and space, tab, carriage return.
- Actions: consist of sequences of printable characters (excluding those excluded for identifiers above), The *internal* or *idling* action τ is represented as `Tau`. The complement actions, which are indicated by an overbar (e.g. $\overline{a}$), are formed by single quotation mark postfixing, (e.g. `a'`).

The CCS process constructors are as follows.

1. Nil: The constant `Nil` is the CCS process **Nil**.
2. Action prefixing: If `a` is an action and `P` is a process, then `a.P` is a process.
3. Summation: If `P1,...,Pn` are processes, then `P1+...+Pn` is a process.
4. Parallel composition: If `P1,...,Pn` are processes, then `P1|...|Pn` is a process.
5. Restriction: If `P` is a process, `a1,...,an` are actions except τ, then `P`$\backslash\{$`a1,...,an`$\}$ is a process.

6. Relabelling: If P is a process, and a1,...,an,b1,...,bn are actions, then P[a1/b1,...,an/bn] is a process.

7. Recursion: If x is an identifier, P(x) is a process with x as free variables, then Rec x P(x) is a process.

Here is a process example of three cell buffer in the syntax of LegoMC.

```
cell = Rec x a.b'.x;;
c0 = cell[c/b];;
c1 = cell[c/a,d/b];;
c2 = cell[d/a];;
buffer3 = (C0 | C1 | C2)\{c,d};;
```

6.4.2 ICPL

LegoMC uses the following syntactic conventions for ICPL programs.

- Identifiers: consist of sequences of printable characters, where the first character must be a letter. There are some characters which cannot be used as part of an identifier. These are: .,=()[]{}+-*|\ and space, tab, carriage return.

The ICPL natural number expressions are as follows.

1. Constants: general natural numbers such as 0,1,2,3,....

2. Variables: unused identifiers.

3. Plus: If a and b are natural number expressions, then a+b is a natural number expression.

4. Minus: If `a` and `b` are natural number expressions, then `a-b` is a natural number expression.

5. Times: If `a` and `b` are natural number expressions, then `a*b` is a natural number expression.

The ICPL boolean expressions are as follows.

1. Constants: `true, false`.

2. Negation: If `a` is a boolean expressions, then `-a` is a boolean expression.

3. Conjunction: If `a` and `b` are boolean expressions, then `a&&b` is a boolean expression.

4. Disjunction: If `a` and `b` are boolean expressions, then `a||b` is a boolean expression.

5. Natural number Equality: If `a` and `b` are natural number expressions, then `a==b` is a boolean expression.

6. Natural number Less: If `a` and `b` are natural number expressions, then `a<b` is a boolean expression.

The ICPL primitive statements are as follows

1. Skip: `skip` is a primitive statement.

2. Assignment: if `a` is an identifier and `b` is a natural number expression, then `a:=b` is a primitive statement.

3. Boolean Await: if `a` is a boolean expression, then `await a` is a primitive statement.

4. Semaphore Wait: if `a` is a identifier, then `wait a` is a primitive statement.

5. Semaphore Signal: if `a` is an identifier, then `signal a` is a primitive statement.

The ICPL sequential process and program constructors are as follows.

1. Empty process: The constant `Empty` is the empty process ϵ.
2. Primitive statement: If `a` is a primitive statement, then `a` is a process.
3. If statement: If `a` is a boolean expression, `P1` and `P2` are processes, then `if a then P1 else P2 endif` is a process.
4. While statement: If `a` is a boolean expression, `P` is a process, then `while a do P done` is a process.
5. Sequential composition: If `P1` and `P2` are processes, then `P1;P2` is a process.
6. Single process program: If `P` is a process, then `P` is a program.
7. Parallel composition: If `P1` is a process and `P2` is a program, then `P1|P2` is a program.

The memory and states are constructed as follows.

- Memory: If `V1 ...Vn` are program variables and `N1 ...Nn` are natural numbers, then {`(V1,N1),...,(Vn,Nn)`} is the memory.
- State: If `M` is memory and `P` is a program, then `(M,P)` is a state.

6.4.3 Temporal Logics

LegoMC uses the following syntactic conventions for temporal logic formulas.

1. Identifiers: consist of sequences of printable characters, where the first character must be a letter. There are some characters which cannot be used as part of an identifier. These are: .,=()[]{}+-*|\/' and space, tab, carriage return.

2. Or: If P and Q are formulas, so is P\/Q.

3. And: If P and Q are formulas, so is P/\Q.

4. Dia: If P is a formula and a,a1,a2,...,an are actions, then <a>P, <->P and <a1,a2,...,an>P are formulas.

5. Box: If P is a formula and a,a1,a2,...,an are formulas, then [a]P, [-]P and [a1,a2,...,an]P are formulas.

6. Nu: If P is a formula and Z is an unused identifier, then nu Z P is a formula.

7. Mu: If P is a formula and Z is an unused identifier, then mu Z P is a formula.

8. True: tt is a formula.

9. False: ff is a formula.

10. Free from deadlock: deadlockfree is a formula.

11. Always: If P is a formula, so is AG P.

12. Eventually: If P is a formula, so is EF P.

The following two syntax of formula is for ICPL only.

1. At: at(pno,lno) is a formula where pno and lno are natural numbers.

2. Not At: notat(pno,lno) is a formula where pno and lno are natural numbers.

6.4.4 Commands, Comments and Abbreviations

At present, only two commands are defined as follows.

1. Check: If `s` is a state and `P` is a formula, then `Check s P` will give an answer of "true" or "false".

2. Prove: If `s` is a state and `P` is a formula, then `Prove s P` will return a proof term if `Check s p` returns "true".

Comments can be made by `(* ...*)`

The Identifiers can also be used to abbreviate a process or formula. If A is an identifier, P is a process or formula, then A can be an abbreviation of P defined as:

Abbreviation: `A = P`.

The end of a command and abbreviation is represented by double semi-colon: `;;`.

6.5 Examples

This section presents two simple verification examples of using LegoMC. More complicated case studies and the integration with other methods to verify infinite state cases are presented in the next two chapters.

The first example is a ticking clock modeled in CCS. The second example is a solution for 2-process mutual exclusion problem by using semaphores. Beside the automation, the user interfaces with domain-specific syntax make the verification easier and more readable.

Example 1: Ticking Clock Modeled in CCS

This example is taking from [Sti92]. There are three versions of ticking clocks. The first one $Cl1$ shows a clock can only tick and tick forever. The second clock $Cl2$ can tick and also tock rotately. The third one $Cl3$ can tick but can go dead as well. We can verify those properties and generate the proof terms completely automatically in LegoMC.

Below is a sample input to LegoMC. We use `Check` to check the property first and then use `Prove` to generate proof terms.

```
cl1 = Rec x tick.x;;


Check cl1 deadlockfree;;  (* free from deadlock => true *)
Check cl1 (<tick>tt);;    (* able to tick => true *)
Check cl1 (EF [tick]ff);;  (* eventually inable to tick => false *)
(* perpetually ticks and can do nothing else => true*)
Check cl1 (AG (<->tt/\[-(tick)]ff));;

Prove cl1 deadlockfree;;
Prove cl1 (<tick>tt);;
Prove cl1 (AG (<->tt/\[-(tick)]ff));;



cl2 = Rec x tick.tock.x;;


(* eventually inable to tock *)
Check cl2 (EF [tock]ff);;
Prove cl2 (EF [tock]ff);;
```

```
cl3 = Rec x tick.x+tick.Nil;;

(* It is possible that the clock tick forever. *)
Check cl3 (nu Z (<tick>Z));;
Prove cl3 (nu Z (<tick>Z));;
```

Example 2: Semaphore Solution for 2-process Mutual Exclusion Problem

Even this is a simple example, the presentation and indeed the verification process of a similar algorithm in chapter 5 are very complicated. Here we can easily check many properties and generate proof terms by LegoMC.

```
(* semaphore based on busy-wait *)
p = while true do
      wait s;
      skip;   (* critical section *)
      signal s
    done;;
pro = p|p;;
init = {(s,1)};;  (* initial memory *)

(* mutual exclusion property, critical section is at position 3 *)
me = AG (notat(1,3)\/notat(2,3));;
(* another representation of mutual exclusion property *)
alter = AG([skip] nu Z ([skip]ff/\[-(signal s)]Z));;

Check (init,pro) me;;     (* true *)
```

```
Check (init,pro) alter;;     (* true *)
Check (init,pro) deadlockfree;;     (* true *)
(* whenever process one wants to enter its critical section,
it can eventually do *)
Check (init,pro) AG (notat(1,2)\/(evn at(1,3)));;  (* true *)

Prove (init,pro) me;;
Prove (init,pro) alter;;
Prove (init,pro) deadlockfree;;
Prove (init,pro) AG (notat(1,2)\/(evn at(1,3)));;
```

Chapter 7

Finite-State Examples

This chapter demonstrates how to use LegoMC to verify finite state systems. Since model checking can be used to verify finite-state systems completely automatically, the examples in this chapter are verified automatically by LegoMC.

A system analysis process includes the following steps.

1. System Modeling.
2. System Specification.
3. Verification and Analysis
4. System Improvement
5. Re-Analysis

In LegoMC, ICPL and CCS are used to model systems and temporal logics are used to specify system properties. The automation in LegoMC makes verification only by a command `Check`. System improvement and re-analysis are easy as well by simply modifying the model and executing `Check` again. Once users are satisfied with the result, they can then use command `Prove` to generate the proof terms for the final model to be type checked by Lego to further ensure their confidence in the verification result. Therefore, LegoMC has the advantage of early debugging and also final rigorous proofs. I will show how LegoMC is used to verify systems through several examples.

The first example is a simple communicating protocol. We model it in both CCS and ICPL and then use LegoMC to verify the desired properties. This example shows the comparison of verification on CCS and ICPL and demonstrates the process of improving a system design by our tool. The second example is a class of mutual exclusion algorithms. Since we use ICPL as the description language and use LegoMC to do verification automatically, we can easily formally verify all of those mutual exclusion algorithms. This example shows how easily to use LegoMC to analyse and compare a group of similar algorithms.

7.1 A Simple Communicating Protocol

This example, which is taken from [Wal87], is an extremely simple communication protocol with sender entity *Sender* and receiver entity *Receiver*, interconnected with a medium. It takes into account the possibility that a message may be lost during transmission. *Sender* transmits the message through *Medium* to *Receiver*. On receiving a message, *Receiver* will send an acknowledgement through *Medium* to *Sender*. After receiving such an acknowledgement, *Sender* may send another message. The *Medium* is not a reliable medium which may lose the message.

This example was used by Walker to explain its divergent behaviour. In our analysis, we divide this into 3 phases and explain the improvement process of this protocol design. First, we assume the medium is a reliable medium which will not lose messages. Then, we release the assumption and analyse this protocol with losing medium. Finally, we add a timer to enable the sender to re-send the message once the message is lost. We assume the acknowledge channel is safe and omit the data flow in this analysis.

We use both ICPL and CCS to model this protocol and compare them.

7.1.1 Modeling in ICPL

System Modeling

The system can be modeled in ICPL as Fig. 7.1. We use a semaphore variable *med* to model the transport medium and a shared variable *ack*, which has two states, *empty* and *ok*, to model the acknowledge channel. The difference of semaphore variables and shared variables is that semaphore variables execute variable access and variable updating in a single primitive statement whereas shared variables execute them by two separated primitive statements. If variable access and variable updating are executed in two separated primitive statements, other processes may access the variable.

System Specification

There are three important properties about this protocol that we want to prove as follows.

Property 1

This protocol is free from deadlock.

$$deadlockfree$$

```
(* msg : Sem => message channel is a semaphore *)
(* ack : Var => acknowledge channel is a variable*)
empty = 0;;
ok = 1;; (* two possible states of acknowledge channel *)
(* time : Var => time is a variable *)
timeout = 3;; (* timeout is a constant,  we set it to 3 here *)
sender = While true do
                await ack==ok;
                ack := empty;
                signal msg   (* send the message *)
         done;;
sender1= While true do
                await ack==ok  timeout < time;
                ack := empty;
                time := 0;
                signal msg  (* send the message *)
         done;;
medium = While true do
                wait msg    (* the message is lost *)
         done;;
receiver = While true do
                  wait msg;  (* get the message *)
                  ack := ok
           done;;
timer = While true do
          await time < timeout;
          time := time + 1
        done;;
protocol1 = sender|receiver
protocol2 = sender|receiver|medium
protocol3 = sender1|receiver|medium|timer
init = {(msg,0),(ack,empty),(time,0)}  (* initial state *)
```

Figure 7.1: A simple transport protocol

Property 2

After putting the message in the medium, the program can not put messages in unless the ack channel becomes ok.

$$AG([signal\ med]\nu Z(inable[signal\ med] \wedge ([-(ack := ok)]Z)))$$

Property 3

After sending, the receiver eventually receives, which means the process 2, receiver, is at line number 3.

$$AG([send]EF(\text{At } 2\ 3)))$$

Verification and Analysis

We can prove $protocol1$ satisfies all of the above three properties by LegoMC automatically.

$(init, protocol1) \vdash deadlockfree$
$(init, protocol1) \vdash AG([signal\ med]\nu Z(inable[signal\ med] \wedge ([-(ack := ok)]Z)))$
$(init, protocol1) \vdash AG([send]EF(\text{At } 2\ 3)))$

The transport protocol is however not safe and therefore the message can be lost during the transport. We use another entity *medium* to model the losing medium which will consume the message in *med*. The protocol is $protocol2$ in Fig. 7.1. Although $protocol2$ can still satisfy property 2 and 3, it fails in property 1 because neither receiver nor sender can proceed anymore once the message in the medium is lost.

$(init, protocol2) \nvdash deadlockfree$

$(init, protocol2) \vdash AG([signal\ med]\nu Z(inable[signal\ med] \wedge ([-(ack := ok)]Z)))$

$(init, protocol2) \vdash AG([send]EF(\mathrm{At}\ 2\ 3)))$

Improvement and Re-analysis

We add another entity, timer, to allow sender to re-send the message after time-out. The protocol is *protocol*3. The *sender* becomes *sender*1 which will set timer to zero before sending a message and will re-send the message once the timer reaches timeout. After this improvement, the protocol can satisfy all of three properties.

$(init, protocol3) \vdash deadlockfree$

$(init, protocol3) \vdash AG([signal\ med]\nu Z(inable[signal\ med] \wedge ([-(ack := ok)]Z)))$

$(init, protocol3) \vdash AG([send]EF(\mathrm{At}\ 2\ 3)))$

7.1.2 Modeling in CCS

Walker modeled this protocol in CCS as follows.

```
Sending = rec x (ms.sm'.x + rs.rece.sm'.x)
Sender  = rece.sm'.Sending
Medium1 = rec x (mr'.sm.x + tau.ms'.sm.x)
Medium = sm.Medium1
Receiver = rec x mr.send.x

protocol = (Sender | Medium | Receiver)\{sm,ms,mr,rs};;
```

The `Sender` receive a data to transmit, it then send it to medium by action `mr'` and then become state `Sending`. State `Sending` can either get an acknowledge `rs` from receiver and then wait for next data or get a timeout message `ms` from medium and re-transmit the data. After receiving a data from Sender `sm`, the medium can either send this data to receiver and then wait for next data or pass the time `tau` and then send a timeout `ms'` to sender. Receiver can receive a data from medium and then send an acknowledgement `rs'` to sender.

The three properties now become

Property 1

This protocol is free from deadlock.

$$deadlockfree$$

Property 2

After putting the message in the medium, the program can not put messages in unless the ack channel becomes OK.

$$AG([sm']\nu Z(inable\{sm'\} \wedge ([-rs]Z)))$$

Property 3

After sending, the receiver eventually receives.

$$AG([sm']EF(able\{mr\})))$$

We can also verify this protocol in CCS version by LegoMC automatically.

$protocol \vdash deadlockfree$

$protocol \vdash AG([sm']\nu Z(inable\{sm'\} \wedge ([-rs]Z)))$
$protocol \vdash AG([sm']EF(able\{mr\})))$

7.1.3 Comparison

We consider three aspects, modeling, specification and verification, to compare CCS and ICPL in analysing systems in our verification environment.

Modeling

ICPL should be easier for programmers to model their systems. In the future, it has the potential that programmers can use their programs directly rather than translate their programs to other description languages. However, CCS can be more concise for certain small systems.

Specification

The position property of ICPL provides an easy way to specify position. For example, the mutual exclusion property for two processes should be "There is no state in which more than one process at the critical section." Suppose the critical section is at line four of programs, this property can be expressed in ICPL as follows:

$$AG\ Not(At(1,4) \wedge At(2.4))$$

which means "There is no state in which process one is at line four and process two is at line four."

The property in CCS should look like

$$protocol \vdash AG([sm']\nu Z(inable\{sm'\} \wedge ([-rs]Z)))$$

which means "After sending a message, the protocol cannot send again unless sender get an acknowledgement from receiver" which doesn't quite catch the original meaning of mutual exclusion property.

Verification

Since we are using automatic tools, the verifications in both CCS and ICPL are similar by using the commands `Check` and `Prove`.

7.2 Mutual Exclusion Algorithms

Mutual exclusion is an essential property for concurrent systems. The difficulty in reasoning reliably about concurrent algorithms has long been recognized. There are many algorithms in the literatures to solve mutual exclusion problems. There are basically three properties we expect for these algorithms, mutual exclusion, deadlock freedom and fairness (non-starvation). There are two versions of fairness property, *week fairness* and *strong fairness*.

Week fairness, which is also referred to as *justice* [MP92], is based on the assumption of hardware fairness which means the hardware is a fair device so that all concurrent processes have the same possibility to access processors, i.e. no single process is consistently neglected. Week fairness can be expressed in temporal logic as "Whenever a process attempts to enter its critical section, there exists a path on which a process can eventually enter critical section." Since hardware is a fair device, it should be possible to choose the path which a process can reach its critical section. For a process, e.g. process one, if its critical section is at position 3, the week fairness property can be denoted as follows:

$$AG(notat(1,3) \vee (EF\ at(1,4)))$$

where we use "$notat(1,3) \vee (EF\ at(1,4))$" to replace "$at(1,3)$ implies $(EF\ at(1,4))$" because we do not have **implies** operator in positive version of μ-calculus.

Strong fairness, which is also referred to as *compassion* [MP92], is based on the assumption that the hardware could be unfair and it is possible that hardware always grants access to some processes and completely ignores the requests of other processes. Strong fairness can be expressed in temporal logic as "Whenever a process attempts to enter its critical section, for all paths on which a process can eventually enter critical section." Since the hardware is unfair, to satisfy non-starvation requirement a process should reach its critical section on all paths. For the same process as above, the strong fairness property can be denoted as follows:

$$AG(notat(1,3) \vee (AF\ at(1,4))).$$

This section presents the verification of several larger algorithms for two-process mutual exclusion. LegoMC is a very suitable tool for analysing and comparing several similar algorithms, implementations or systems since the modeling and verification in LegoMC are easy and therefore help people to focus on the algorithms and their properties. Most of the formulations of the algorithms are taken from Raynal's book [Ray86].

7.2.1 Dekker's Algorithm

The first algorithm to solve two-process mutual exclusion problem was designed by T. Dekker. There are two processes `p1` and `p2`, two boolean variables `flag1` and `flag2` whose initial values are `false`, and a variable `turn` whose value can be 1 or 2.

```
(* Dekker's Algorithm for 2-process mutual exclusion *)
(* The formulation here is taken from Raynal's book *)
```

```
(* flag1,flag2 : boolean with false is 0, true is 1 *)
(* turn can be 1 or 2 *)
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
            flag1 := 1;
            while flag2==1 do
                 if turn==2 then
                     flag1 := 0;
                     await turn==1;
                     flag1 := 1
                 else
                     skip
                 endif
            done;
            cri1;
            turn := 2;
            flag1 := 0
       done;;

p2 = while true do
             flag2 := 1;
             while flag1==0 do
                  if turn==1 then
                     flag2 := 0;
                     await turn==2;
                     flag2 := 1
                  else
```

```
                skip
             endif
         done;
         cri2;
         turn := 1;
         flag2 := 0
     done;;

pro = p1|p2;;
init1 = {(flag1,0),(flag2,0),(turn,1)};; (* initial memory *)
init2 = {(flag1,0),(flag2,0),(turn,2)};; (* initial memory *)

me = notat(1,4) \/ notat(2,4);;

Check (init1,pro) AG me;; (* true *)
Check (init2,pro) AG me;; (* true *)
Check (init1,pro) deadlockfree;; (* true *)
Check (init2,pro) deadlockfree;; (* true *)

(* check weak fairness => true *)
Check (init1,pro) AG (notat(1,4) \/ (EF at(1,5)));;
(* check weak fairness => true *)
Check (init2,pro) AG (notat(2,4) \/ (EF at(2,5)));;
(* check strong fairness => false *)
Check (init1,pro) AG (notat(1,4) \/ (AF at(1,5)));;
(* check strong fairness => false *)
Check (init2,pro) AG (notat(2,4) \/ (AF at(2,5)));;
```

The critical section is position 4. The mutual exclusion property is therefore either process 1 not at position 4 or process 2 not at position 4. The processes attempt to enter their critical sections at position 4 and leave their critical section at position 5. We can prove that under both initial conditions (turn=1 or turn=2), Dekker's algorithm satisfies mutual exclusion and deadlockfree. Dekker's algorithm can satisfy weak fairness but not strong fairness.

7.2.2 Dijkstra's Algorithm

Dijkstra [Dij65] generalized Dekker's solution to the case of n processes. We adapted the algorithm for two processes from Raynal's book [Ray86]. Variable `turn` is the same as Dekker's algorithm but `flag1` and `flag2` take 3 values (`passive`, `requesting` and `in_cs`) with initial values as `passive`.

```
(* Dijkstra's Algorithm for 2 process mutual exclusion *)
(* The formulation here is taken from Raynal's book *)

passive = 1;;
requesting = 2;;
in_cs = 3;;
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
                  flag1 := requesting;
                  while turn==2 do
                        if flag2==passive then
                            turn := 1
                        else
```

```
                    skip
                endif
          done;
          flag1 := in_cs;
          while flag2==in_cs do
                flag1 := requesting;
                while turn==2 do
                      if flag2==passive then
                          turn := 1
                      else
                          skip
                      endif
                done;
                flag1 := in_cs
          done;
          cri1;
          flag1 := passive
    done;;

p2 = while true do
          flag2 := requesting;
          while turn==1 do
                if flag1==passive then
                    turn := 2
                else
                    skip
                endif
          done;
          flag2 := in_cs;
```

```
                    while flag1==in_cs do
                          flag2 := requesting;
                          while turn==1 do
                                if flag1==passive then
                                    turn := 2
                                else
                                    skip
                                endif
                          done;
                          flag2 := in_cs
                    done;
                    cri2;
                    flag2 := passive
              done;;

pro = p1|p2;;
init1 = {(flag1,0),(flag2,0),(turn,1)};; (* initial memory *)
init2 = {(flag1,0),(flag2,0),(turn,2)};; (* initial memory *)

me = notat(1,6) \/ notat(2,6);;

Check (init1,pro) AG me;; (* true *)
Check (init2,pro) AG me;; (* true *)
Check (init1,pro) deadlockfree;; (* true *)
Check (init2,pro) deadlockfree;; (* true *)

(* check weak fairness => true *)
Check (init1,pro) AG (notat(1,6) \/ (EF at(1,7)));;
(* check weak fairness => true *)
```

```
Check (init2,pro) AG (notat(1,6) \/ (EF at(1,7)));;
(* check strong fairness => false *)
Check (init1,pro) AG (notat(1,6) \/ (AF at(1,7)));;
(* check strong fairness => false *)
Check (init2,pro) AG (notat(1,6) \/ (AF at(1,7)));;
```

The critical section is position 6. The mutual exclusion property is therefore either process 1 not at position 6 or process 2 not at position 6. The processes attempt to enter their critical sections at position 6 and leave at position 7. We can prove that under both initial conditions (turn=1 or turn=2), Dijkstra's algorithm satisfies mutual exclusion and deadlockfree. Dijkstra's algorithm can satisfy weak fairness but not strong fairness.

7.2.3 Hyman's Algorithm

Hyman's algorithm [Hym66] tried to simplify Dijkstra's algorithm in the case of two processes. However, Hyman's simplification is not entirely satisfactory. The variable `flag1`, `flag2` and `turn` are the same as Dekker's algorithm.

```
(* Hyman's Algorithm for 2 process mutual exclusion *)
(* Raynal's book *)

(* flag1,flag2 : boolean with false is 0, true is 1 *)
(* turn can be 1 or 2 *)
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
       flag1 := 1;
       while - turn==1 do
```

```
          await flag2==0;
          turn := 1
       done;
       cri1;
       flag1 := 0
     done;;

p2 = while true do
       flag2 := 1;
       while - turn==2 do
          await flag1==0;
          turn := 2
       done;
       cri2;
       flag2 := 0
     done;;

pro = p1|p2;;
init1 = {(flag1,1),(flag2,1),(turn,1)};; (* initial memory *)
init2 = {(flag1,1),(flag2,1),(turn,2)};; (* initial memory *)

me = notat(1,4) \/ notat(2,4);;

Check (init1,pro) AG me;; (* false *)
Check (init2,pro) AG me;; (* false *)
Check (init1,pro) deadlockfree;; (* true *)
Check (init2,pro) deadlockfree;; (* true *)

(* check weak fairness => true *)
```

```
Check (init1,pro) AG (notat(1,4) \/ (EF at(1,5)));;
(* check weak fairness => true *)
Check (init2,pro) AG (notat(2,4) \/ (EF at(2,5)));;
(* check strong fairness => false *)
Check (init1,pro) AG (notat(1,4) \/ (AF at(1,5)));;
(* check strong fairness => false *)
Check (init2,pro) AG (notat(2,4) \/ (AF at(2,5)));;
```

The critical section is position 4. The mutual exclusion property is therefore either process 1 not at position 4 or process 2 not at position 4. The processes attempt to enter their critical sections at position 4 and leave their critical sections at position 5. We can prove that under both initial conditions (turn=1 or turn=2), Hyman's algorithm does not satisfy mutual exclusion but does satisfy deadlockfree. Hyman's algorithm can satisfy weak fairness but not strong fairness.

7.2.4 Knuth's Algorithm

Knuth's protocol [Knu66] was the first strong fair solution. The variable `flag1`, `flag2` and `turn` are the same as Dijkstra's algorithm.

```
(* Knuth's Algorithm for 2 process mutual exclusion *)
(* The formulation here is taken from Raynal's book *)

(* flag1 : passive, requesting, in_cs *)
(* turn can be 1 or 2 *)
passive = 1;;
requesting = 2;;
in_cs = 3;;
```

```
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
       flag1 := requesting;
       await turn==1 || flag2==passive;
       flag1 := in_cs;
       while flag2==in_cs do
         flag1 := requesting;
         await turn==1 || flag2==passive;
         flag1 := in_cs
       done;
       turn := 1;
       cri1;
       turn := 2;
       flag1 := passive
     done;;

p2 = while true do
       flag2 := requesting;
       await turn==2 || flag1==passive;
       flag2 := in_cs;
       while flag1==in_cs do
         flag2 := requesting;
         await turn==2 || flag1==passive;
         flag2 := in_cs
       done;
       turn := 2;
       cri1;
```

```
    turn := 1;
    flag2 := passive
  done;;

pro = p1|p2;;
init1 = {(flag1,1),(flag2,1),(turn,1)};; (* initial memory *)
init2 = {(flag1,1),(flag2,1),(turn,2)};; (* initial memory *)

me = notat(1,7) \/ notat(2,7);;

Check (init1,pro) AG me;; (* true *)
Check (init2,pro) AG(me);; (* true *)
Check (init1,pro) deadlockfree;; (* true *)
Check (init2,pro) deadlockfree;; (* true *)

(* check weak fairness => true *)
Check (init1,pro) AG (notat(1,7) \/ (EF at(1,8)));;
(* check weak fairness => true *)
Check (init2,pro) AG (notat(2,7) \/ (EF at(2,8)));;
(* check strong fairness => true *)
Check (init1,pro) AG (notat(1,7) \/ (AF at(1,8)));;
(* check strong fairness => true *)
Check (init2,pro) AG (notat(2,7) \/ (AF at(2,8)));;
```

The critical section is position 7. The mutual exclusion property is therefore either process 1 not at position 7 or process 2 not at position 7. The processes attempt to enter their critical sections at position 7 and leave their critical section at position 8. We can prove that under both initial conditions (turn=1 or turn=2), Knuth's algorithm satisfies mutual exclusion, deadlockfree, weak fairness and strong fairness.

7.2.5 Peterson's Algorithm

Peterson [Pet81] gave an elegant and simple solution to mutual exclusion problem. The variables `flag1`, `flag2` and `turn` are the same as Dekker's algorithm.

```
(* Peterson's Algorithm for 2 process mutual exclusion *)
(* The formulation here is taken from Raynal's book *)

(* flag1,flag2 : boolean with false is 0, true is 1 *)
(* turn can be 1 or 2 *)
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
       flag1 := 1;
       turn := 1;
       await flag2==0 || turn==2;
       cri1;
       flag1 := 0
     done;;

p2 = while true do
       flag2 := 1;
       turn := 2;
       await flag1==0 || turn==1;
       cri2;
       flag2 := 0
```

```
    done;;

pro = p1|p2;;
init1 = {(flag1,0),(flag2,0),(turn,1)};; (* initial memory *)
init2 = {(flag1,0),(flag2,0),(turn,2)};; (* initial memory *)

me = notat(1,5) \/ notat(2,5);;

Check (init1,pro) AG me;; (* true *)
Check (init2,pro) AG me;; (* true *)
Check (init1,pro) deadlockfree;;  (* true *)
Check (init2,pro) deadlockfree;;  (* true *)

(* check weak fairness => true *)
Check (init1,pro) AG(notat(1,5) \/ EF at(1,6));;
(* check weak fairness => true *)
Check (init2,pro) AG(notat(2,5) \/ EF at(2,6));;
(* check strong fairness => true *)
Check (init1,pro) AG(notat(1,5) \/ AF at(1,6));;
 (* check strong fairness => true *)
Check (init2,pro) AG(notat(2,5) \/ AF at(2,6));;
```

The critical section is position 5. The mutual exclusion property is therefore either process 1 not at position 5 or process 2 not at position 5. The processes attempt to enter their critical sections at position 5 and leave their critical sections at position 6. We can prove that under both initial conditions (turn=1 or turn=2), Peterson's algorithm satisfies mutual exclusion, deadlockfree, weak fairness and strong fairness.

7.2.6 Lamport's Algorithm

Lamport's one-bit algorithm [Lam86] uses only one variable for each process. The variables **flag1** and **flag2** are the same as Dekker' algorithm.

```
(* Lamport's Algorithm for 2 process mutual exclusion *)


(* flag1,flag2 : boolean with false is 0, true is 1 *)
cri1 = skip;;   (* critical section *)
cri2 = skip;;   (* critical section *)

p1 = while true do
              flag1 := 1;
              await flag2==0;
              cri1;
              flag1 := 0
        done;;

p2 = while true do
              flag2 := 1;
              while flag1==1 do
                 flag2 := 0;
                 await flag1==0;
                 flag2 := 1
              done;
              cri2;
              flag2 := 0
         done;;

pro = p1|p2;;
```

```
init = {(flag1,0),(flag2,0)};; (* initial memory *)

me = notat(1,4) \/ notat(2,4);;

Check (init,pro) AG me;; (* true *)
Check (init,pro) deadlockfree;; (* true *)

(* check weak fairness => true *)
Check (init,pro) AG (notat(1,4)\/(EF at(1,5)));;
(* check strong fairness => true *)
Check (init,pro) AG (notat(1,4)\/(AF at(1,5)));;
(* check weak fairness => true *)
Check (init,pro) AG (notat(2,4)\/(EF at(1,5)));;
(* check strong fairness => false *)
Check (init,pro) AG (notat(2,4)\/(AF at(1,5)));;
```

The critical section is position 4. The mutual exclusion property is therefore either process 1 not at position 4 or process 2 not at position 4. Lamport's algorithm is not symmetric. The processes attempt to enter their critical sections at position 4 and leave at position 5. We can easily prove that under initial conditions, Lamport's algorithm satisfies mutual exclusion, deadlockfree and weak fairness. The process one satisfies strong fairness, whereas the process two does not.

7.2.7 Results and Comments

The table below summarizes the results obtained from previous sub-sections.

* "Yes" for the first process and "No" for the second process.

Algorithm	Mutual Exclusion	Deadlock Freedom	Weak Fairness	Strong Fairness
Dekker	Yes	Yes	Yes	No
Dijkstra	Yes	Yes	Yes	No
Hyman	No	Yes	Yes	No
Knuth	Yes	Yes	Yes	Yes
Peterson	Yes	Yes	Yes	Yes
Lamport	Yes	Yes	Yes	Yes/No*

Table 7.1: Verification results of mutual exclusion algorithms

Compared with Walker's analysis [Wal89] of mutual exclusion algorithms using CCS and Concurrency Workbench [CPS93], the modeling in ICPL is much simpler and the ICPL presentation is clearer than CCS presentation. Further simplification can be made by creating syntax abbreviation for `repeat ...until` and `for` loop statements.

Chapter 8

Infinite-State Case Studies

As shown in previous chapter, LegoMC provides a convenient way to do verification for systems with finite state space. One feature of our verification environment is the possibility of integrating model-checker with other verification methods to verify infinite state systems. This chapter shows how LegoMC is used upon Lego to analyse systems with infinite state space.

The most basic verification method is through the semantics of description languages and specification languages by doing inductive reasoning over the transitional structure of systems. This method sometimes can have very elegant and straightforward proofs. It is especially useful for parameterized processes, which have many identical processes executing concurrently.

Compositional method provides another alternative for verification. By decomposing a system into several sub-systems, we can then verify the whole system by verifying individual sub-systems. The specification for such a system can also be decomposed into properties that concern only those sub-systems. We have to prove that the conjunction of local properties implies the overall specification.

Another method is trying to reduce the complexity of a verification task by abstraction technique. By proving that an abstraction mapping preserves the properties to be verified, one can work on the simpler abstract system instead of the original more complicated system. However, sometimes it is not so easy to find a suitable abstract system and prove the property is preserved.

The inference rules of the above verification techniques can be formally proved in Lego as libraries. The verification of infinite state systems is carried on Lego. Users are free to choose different methods by evoking their inference rules in the libraries. They can use LegoMC to generate some parts of proofs and then insert those proofs into Lego to complete the whole proof. In this way, we can then combine various verification techniques with LegoMC to form a general framework which can be used to verify more complicated or infinite state systems in a more efficient way.

In the next section, two examples are given to demonstrate the verification by semantics and induction. The first example is an infinite counter, which have an evolving structure. The second example is a token-ring network which has many identical workstations connected in a network. The introduction of compositional method follows in section 8.2. The counter example is re-verified by compositional method. The abstraction technique is presented in section 8.3. We also re-do the verification of token-ring example by abstraction. Finally, some discussion is given in section 8.4.

8.1 Proving by Semantics and Induction

This section presents two examples of proving by semantics and induction. The same examples will be verified by compositional method and abstraction in subsequent two sections.

8.1.1 Example : an Infinite Counter

This example is taken from [Dam95], a counter can count forever. We want to prove that the counter is always able to "up" and is free from deadlock.

$$Counter = rec\ x.up.(x|down.Nil)$$

Since this is an evolving system with infinite state space, we cannot use merely LegoMC to prove it. This infinite counter can be verified purely by semantics. It can also be verified by compositional method which is presented in next section.

Always Able to Up

We want to prove the property $\Phi = AG(able\{up\})$ which is $\nu Z.\langle up\rangle True \wedge [-]Z$.

$$Counter \vdash \Phi \tag{8.1}$$

Expand by the semantics of ν operator, that means $\exists P.P \subseteq (\langle up\rangle True \wedge [-]P)$ and $Counter \in P$. We take this P as the infinite set $\{cnt(i)|i \in nat\}$, i.e.

$$P = \lambda p : Process \exists n : nat.\mathrm{Eq}\ p\ cnt(n)$$

where $cnt(0) = Counter, cnt(1) = Counter|(down.Nil), \ldots, cnt(i{+}1) = cnt(i)|(down.Nil)$. Therefore, the original goal is reduced to the following three sub-goals.

$$P \subseteq \langle up\rangle True \tag{8.2}$$

$$P \subseteq [-]P \tag{8.3}$$

$$Counter \in P \tag{8.4}$$

Sub-goal (8.4) is true by the membership of P. By the semantics of $\langle\ \rangle$ operator, sub-goal (8.2) is $\forall p \in P \exists p'.p \stackrel{up}{\to} p'$ and $p' \in True$. Take p' as $p|down.Nil$, sub-goal (8.2) arrives

$$\forall p \in P.p \xrightarrow{up} p|down.Nil \tag{8.5}$$

$$p|down.Nil \in True \tag{8.6}$$

Sub-goal (8.6) is proved by LegoMC. Sub-goal (8.5) can be proved by inductive reasoning over natural number i. The only sub-goal left now is (8.3). By the semantics of [ ] operator, sub-goal (8.3) is $\forall p \in P \forall p' \exists \alpha, p \xrightarrow{\alpha} p'$ implies $p' \in P$ which can be proved by inductive reasoning over natural number i.

Therefore, we finish the proof by means of the semantics of μ-calculus formulas and induction.

Deadlock Freedom

Another property we want to prove is deadlock freedom which is $\nu Z.\langle - \rangle True \wedge [-]Z$.

By the syntax of formula we can find that this property is similar to the previous one and therefore we only have to prove the following formula

$$\forall p \in P \exists \alpha, p'.p \xrightarrow{\alpha} p'$$

Take α as up , the formula becomes $\forall p \in P \exists p'.p \xrightarrow{up} p'$ which we have proved in proving the previous property.

8.1.2 Example: a Token Ring Network

Assume there are n workstations in a ring network as Fig 8.1. Every workstation which wants to enter its critical section should hold a token which passes around the ring. The workstation which holds the token can also merely do nothing and pass out the token. If the workstation enters its critical section, it can only exit the critical section but still keep the token. The whole model can be expressed in CCS as follows:

$$I = pass.IT$$

$$IT = enter.exit.IT + \overline{pass}.I$$

$$Ring(n) = (IT|I|\ldots|I)\backslash\{pass\} \text{ with } n+1 \ Is\text{— at least two processes}$$

where I is the idle workstation and IT is the workstation which holds the token.

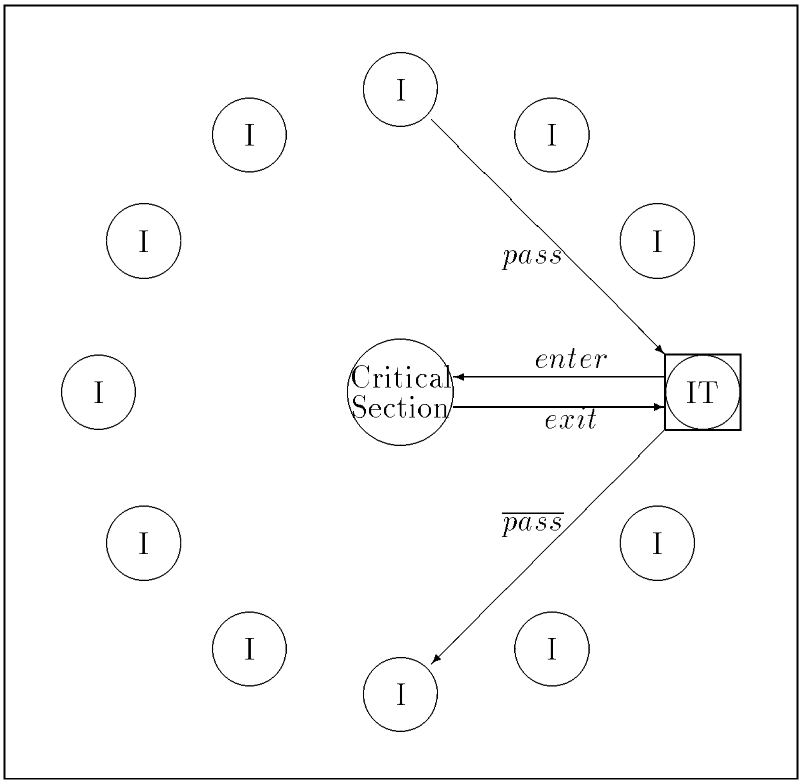

Figure 8.1: A token ring network with 12 workstations

We first prove some transition relations as lemmas. We will need some inversion lemmas which are presented in chapter 4.

Lemma 8.1.1 $\forall n.Ring(n) \xrightarrow{a} P$ *implies* $(a = \tau \wedge P = Ring(n)) \vee (a = enter \wedge P = Ring_{enter}(n))$, *where* $Ring_{enter}(n) = (exit.IT|I|\ldots|I)\backslash\{pass\}$

Proof Using lemma_hide and the premise, we can have two sub-goals depending on whether $a = \tau$.

Case 1. $a \neq \tau$

We are going to prove $a = enter \wedge P = Ring_{enter}(n)$. This can be proved by induction over n.

Base case: $Ring(0) \overset{enter}{\rightarrow} P$ implies $P = Ring_{enter}(0)$

That is $IT|I \overset{enter}{\rightarrow} P$ implies $P = exit.IT|I$. Since I cannot perform $enter$, only IT can $enter$ and by lemma_cho and lemma_dot the $enter$-derivative state of IT is $exit.IT$. Therefore $P = exit.IT|I$

Induction case: $(Ring(n) \overset{enter}{\rightarrow} P$ implies $P = Ring(n)_{enter})$ implies $(Ring(n+1) \overset{enter}{\rightarrow} P$ implies $P = Ring(n+1)_{enter})$

We refine the goal by lemma_par. Since I cannot $enter$, P depends on the $enter-$derivative of $Ring(n)$ which, by induction assumption, is $Ring(n)_{enter}$. Therefore $P = Ring(n+1)_{enter}$

Case 2. $a = \tau$

We still do induction over n .

Base case: $Ring(0) \overset{\tau}{\rightarrow} P$ implies $P = Ring(0)$

The only τ transition of $Ring(0)$ is $IT \overset{\overline{pass}}{\rightarrow} I$ and $I \overset{pass}{\rightarrow} IT$. Therefore P is $I|IT$ which equals to $IT|I$.

Induction case: $(Ring(n) \xrightarrow{\tau} P$ implies $P = Ring(n))$ implies $(Ring(n+1) \xrightarrow{\tau} P$ implies $P = Ring(n+1))$

We refine the goal by lemma_par. There are only one possible τ-transitions of $Ring(n+1)$ which is $Ring(n+1) \xrightarrow{\tau} Ring(n+1)$. Therefore P is $Ring(n+1)$.

$\square$

Lemma 8.1.2 $\forall n. Ring_{enter}(n) \xrightarrow{a} P$ *implies* $a = exit \wedge P = Ring(n)$, *where* $Ring_{enter}(n) = (exit.IT|I|\ldots|I)\backslash\{pass\}$

Proof Using lemma_hide and the premise, we can have two sub-goals depending on whether $a = \tau$.

Case 1. $a \neq \tau$

We are going to prove $a = exit \wedge P = Ring(n)$. We do induction over n.

Base case: $Ring_{enter}(0) \xrightarrow{exit} P$ implies $P = Ring(0)$

That is $exit.IT|I \xrightarrow{exit} P$ implies $P = IT|I$. Since I cannot perform $exit$, only $exit.IT$ can $exit$ and by **lemma_dot** the $exit$-derivative of $exit.IT$ is IT. Therefore $P = IT|I$

Induction case: $(Ring_{enter}(n) \xrightarrow{exit} P$ implies $P = Ring(n))$ implies $(Ring_{enter}(n+1) \xrightarrow{exit} P$ implies $P = Ring(n+1))$

We refine the goal by lemma_par. Since I cannot $exit$, P depends on the $exit$-derivative of $Ring_{enter}(n)$ which, by induction assumption, is $Ring(n)$. Hence $P = Ring(n+1)$.

Case 2. $a = \tau$

When $a = \tau$, we are going to prove that the promise $Ring_{enter}(n) \xrightarrow{\tau} P$ is $false$. Since $exit.It$ can only perform $exit$ and I can only perform $pass$ and $exit$ and $pass$ are not complement, $Ring_{enter}(n)$ can not perform τ.

□

Lemma 8.1.3 $Ring(n) \neq Ring(n)_{enter}$

Proof If $Ring(n) = Ring(n)_{enter}$, then $IT = exit.IT$, which is $false$. Therefore $Ring(n) \neq Ring(n)_{enter}$

□

Theorem 8.1.1 (Mutual exclusion property) *If the process perform enter, it can not enter again except it perform exit.*

$$Ring(n) \vdash \nu Z_1.([enter](\nu Z_2.([enter]false \wedge [-exit]Z_2)) \wedge [-]Z_1)$$

Proof We shall use Φ to abbreviate $[enter](\nu Z_2.([enter]false \wedge [-exit]Z_2))$.

By ν-unfold, $\wedge$-rule and lemma 8.1.1, we can get the following two sub-goals.

$$Ring_{enter}(n) \vdash \nu Z_2([enter]false \wedge [-exit]Z_2) \tag{8.7}$$

$$Ring(n) \vdash [-]\nu Z_1\{Ring(n)\}(\Phi \wedge [-]Z_1) \tag{8.8}$$

Using ν-unfold and $\wedge$-rule, sub-goal 8.7 can arrive two sub-goals as follows

$$Ring_{enter}(n) \vdash [enter]false \tag{8.9}$$

$$Ring_{enter}(n) \vdash [-exit]\nu Z_2\{Ring_{enter}(n)\}([enter]false \wedge [-exit]Z_2 \tag{8.10}$$

By lemma 8.1.2, the only action $Ring(n)_{enter}$ can perform is $exit$. Therefore, the above two sub-goals are true by the semantics of Box operator.

By the semantics of Box operator and lemma 8.1.1, sub-goal 8.8 can be reduced to

$$Ring(n) \vdash \nu Z_1.\{Ring(n)\}(\Phi \wedge [-]Z_1) \tag{8.11}$$

$$Ring_{enter}(n) \vdash \nu Z_1.\{Ring(n)\}(\Phi \wedge [-]Z_1) \tag{8.12}$$

Sub-goal 8.11 can be proved by ν-base rule. Using ν-unfold and $\wedge$-rule, sub-goal 8.12 can be reduced to

$$Ring_{enter}(n) \vdash \Phi \tag{8.13}$$

$$Ring_{enter}(n) \vdash [-]\nu Z_1\{Ring(n), Ring_{enter}(n)\}(\Phi \wedge [-]Z_1) \tag{8.14}$$

By lemma 8.1.2, the only action that $Ring_{enter}(n)$ can perform is $exit$. Therefore sub-goal 8.13 is true because $Ring_{enter}(n)$ cannot perform $enter$.

By the semantics of Box operator and lemma 8.1.2, sub-goal 8.14 can be reduced to

$$Ring(n) \vdash \nu Z_1\{Ring(n), Ring_{enter}(n)\}(\Phi \wedge [-]Z_1) \tag{8.15}$$

which can be proved by ν-base rule.

□

Theorem 8.1.2 *The deadlock free property:*

$$Ring(n) \vdash \nu Z.(\langle - \rangle true \wedge [-]Z)$$

Proof By ν-unfold and $\wedge$-rule, the goal can be reduced to the following two sub-goals.

$$Ring(n) \vdash \langle - \rangle true \tag{8.16}$$

$$Ring(n) \vdash [-](\nu Z.\{Ring(n)\}(\langle - \rangle true \wedge [-]Z)) \tag{8.17}$$

By lemma 8.1.1, $Ring(n)$ can perform $enter$ and therefore $\langle - \rangle true$ is true.

From lemma 8.1.1, the successor state of $Ring(n)$ is either $Ring(n)$ or $Ring_{enter}(n)$. Therefore sub-goal 8.17 can be reduced to

$$Ring(n) \vdash \nu Z.\{Ring(n)\}(\langle - \rangle true \wedge [-]Z) \tag{8.18}$$

$$Ring_{enter}(n) \vdash \nu Z.\{Ring(n)\}(\langle - \rangle true \wedge [-]Z) \tag{8.19}$$

Sub-goal 8.18 can be proved by ν-base rule. Using ν-unfold and $\wedge$ rule again, sub-goal 8.19 arrives

$$Ring_{enter}(n) \vdash \langle - \rangle true \tag{8.20}$$

$$Ring_{enter}(n) \vdash [-](\nu Z.\{Ring(n), Ring_{enter}(n)\}(\langle - \rangle true \wedge [-]Z)) \tag{8.21}$$

By lemma 8.1.2, $Ring_{enter}(n)$ can *exit* and therefore $\langle - \rangle true$ is true. By lemma 8.1.2, $Ring_{enter}(n)$ has only one successor state which is performing *exit* to become $Ring(n)$. The sub-goal becomes

$$Ring(n) \vdash \nu Z.\{Ring(n), Ring_{enter}(n)\}(\langle - \rangle true \wedge [-]Z) \tag{8.22}$$

which can be proved by ν-base rule.
□

8.2 Composition

A concurrent system usually consists of many processes running in parallel. The interleaved execution between individual processes causes the state space to grow exponentially. Therefore a natural solution would be to decompose a system into several sub-systems. The specification for such a system can also be decomposed into properties that concern only those sub-systems. If we know that the conjunction of the local properties implies the overall specification, we can then verify the whole system by verifying individual sub-systems.

By exploring the modular structure of a complex system, *compositional* techniques [Sti85, Win85, AW92, Lon93, And93, ASW94] use the divide-and-conquer approach to decompose the system and the properties into simpler ones. Compositional approach reasons in the structure of the states and works purely on the syntax of states. By a sequence of reductions on the top-level operator of the state, compositional method decomposes a problem into equivalent sub-problems for the immediate sub-components.

For instance, suppose we want to verify a communication protocol consisting of three processes: a sender, a communication media and a receiver. One of the properties about the communication protocol is the deliveribility: data is eventually transmitted correctly from the sender to the receiver. Deliveribility property can be decomposed into two local properties. First, the data should be eventually transferred correctly from the sender to the media. Second, the data should eventually be transferred correctly from the media to the receiver. Since the first property involves only the sender and the media. We should be able to verify the first property using only the sender and the media. In the same way, we should be able to verify the second property using only the media and the receiver. By decomposition, we transform a three-process verification problem into a two-process verification problem. The state space of the latter should normally be smaller.

The local properties are usually only true under certain conditions. Therefore, we have to make some assumptions about the environment of the components to be verified. Those assumptions, which represent requirements on other components, should be verified as well. Finally, we have to show that the conjunction of those local properties implies the original specification.

Compositionality provides many useful features. First, it allows better structuring and decomposition of verification task so that only changed parts have to be re-done when modifying a system. Secondly, the decomposition deduces a complicated verification task to several simpler tasks and therefore overcomes part of the state-exploration problem. Thirdly, the verified components can be reused when they are used to build a larger system and therefore it is possible to build a library of verified standard components for others to use. Fourthly, it is possible to design a system with some undefined parts and still to be able to reason about some properties of it. Finally, by assuming some properties of individual components, we can design and reason individual parts of a system and therefore support group work.

Since the compositional techniques concern mainly with the syntax of system description languages, there should be different inference rules for different description language (e.g. automata, process algebra, imperative programming language, petri net, etc.). We take CCS as an example to experiment the application of our framework to compositional techniques. Due to the characters of composition, we can extract parts of verification tasks to be proved automatically by LegoMC. The other parts of verification tasks and those composition rules are proved in Lego.

8.2.1 Compositional Rules for CCS

To have a sound theorem, we formally prove the compositional rules in Lego by the semantics of μ-calculus and CCS. The following rules for CCS are adapted from Dam's work [Dam95] and Anderson's work [And93].

The rules for boolean connectives and fixed points operators are the same as the model checking rules in section 3.3. The rules for modality operators with idling action τ are as follows.

$$([\tau])\frac{s \vdash \Phi \quad s \vdash [K]\Phi}{s \vdash [\tau, K]\Phi} \quad (\langle\tau\rangle - 1)\frac{s \vdash \Phi}{s \vdash \langle\tau, K\rangle\Phi} \quad (\langle\tau\rangle - 2)\frac{s \vdash \langle K\rangle\Phi}{s \vdash \langle\tau, K\rangle\Phi}$$

By the above rules, we can always extract τ action from modality and therefore we assume there is no τ action in the modality in the following rules. For the modality operators without τ action, the rules can be classified into two categories as follows.

Dynamical Rules The decomposition for dynamical rules can always remove the top operators of CCS.

$$Nil - Box\frac{}{Nil \vdash [K]\Phi} \quad Dot - Dia\frac{x \vdash \Phi}{a.x \vdash \langle K\rangle\Phi}(a \in K)$$

$$Dot - Box1\frac{x \vdash \Phi}{a.x \vdash [K]\Phi}(a \in K) \quad Dot - Box2\frac{}{a.x \vdash [K]\Phi}(a \notin K)$$

$$Cho - Dia1\frac{x \vdash \langle K\rangle\Phi}{x + y \vdash \langle K\rangle\Phi} \quad Cho - Dia2\frac{y \vdash \langle K\rangle\Phi}{x + y \vdash \langle K\rangle\Phi}$$

$$Cho - Box\frac{x \vdash [K]\Phi \quad y \vdash [K]\Phi}{x + y \vdash [K]\Phi}$$

$$Rec\frac{x[rec\ x.s/x] \vdash \Phi}{rec\ x.s \vdash \Phi}$$

Static rules For the rules of static operators, we take Anderson's extension [And93] of μ-calculus for expressing the "pre-images" of the corresponding process operators.

$$\Phi := \ldots|\Phi[f]|\Phi\backslash L|\Phi/t$$

Their semantics is as follows.

$$[\![\Phi[f]]\!]_\rho = \{s|s[f] \in [\![\Phi]\!]_\rho\}$$

$$[\![\Phi\backslash L]\!]_\rho = \{s|s\backslash L \in [\![\Phi]\!]_\rho\}$$

$$[\![\Phi/t]\!]_\rho = \{s|s|t \in [\![\Phi]\!]_\rho\}$$

Therefore, the static rules can be proved as follows.

$$Rel - Dia\frac{s \vdash \langle f^{-1}(K)\rangle\Phi[f]}{s[f] \vdash \langle K\rangle\Phi} \quad Rel - Box\frac{s \vdash [f^{-1}(K)]\Phi[f]}{s[f] \vdash [K]\Phi} \quad Rel\frac{s[f] \vdash \Phi}{s \vdash \Phi[f]}$$

$$Res-Dia\frac{s\vdash\langle K\backslash L\rangle\Phi\backslash L}{s\backslash L\vdash\langle K\rangle\Phi}\quad Res-Box\frac{s\vdash[K\backslash L]\Phi\backslash L}{s\backslash L\vdash[K]\Phi}\quad Res\frac{s\backslash L\vdash\Phi}{s\vdash\Phi\backslash L}$$

$$Par-Dia1\frac{s_1\vdash\langle K\rangle(\Phi/s_2)}{s_1|s_2\vdash\langle K\rangle\Phi}\quad Par-Dia2\frac{s_2\vdash\langle K\rangle(\Phi/s_1)}{s_1|s_2\vdash\langle K\rangle\Phi}$$
$$Par-Box\frac{s_1\vdash[K](\Phi/s_2)\quad s_2\vdash[K](\Phi/s_1)}{s_1|s_2\vdash[K]\Phi}\quad Par\frac{s_1|s_2\vdash\Phi}{s_1\vdash\Phi/s_2}$$

8.2.2 Example

By composition, we can get a more concise proof for the counter example presented in section 8.1.

$$Counter = rec\ x.up.(x|down.Nil)$$

The property of "Always able to up" is proved as follows

$$Counter\vdash AG(able\{up\}) \tag{8.23}$$

Using ν-unfold and $\wedge$-rule, we arrive at two sub-goals

$$Counter\vdash able\{up\} \tag{8.24}$$
$$Counter\vdash[up]\nu Z\{Counter\}(able\{up\}) \tag{8.25}$$

Sub-goal 8.24 can be proved by LegoMC. Using Dot-Box1, sub-goal 8.25 become

$$Counter|down.Nil\vdash\nu Z\{Counter\}(able\{up\}) \tag{8.26}$$

which can then be reduced by Par rule to

$$Counter\vdash(\nu Z\{Counter\}(able\{up\}))/down.Nil \tag{8.27}$$

We can move in the pre-image operator of Par by lemma

$$s \vdash (\nu Z.U\ \phi)/t \leftrightarrow s \vdash \nu Z.U\ (\Phi/t).$$

The goal now becomes

$$Counter \vdash \nu Z\{Counter\}((able\{up\})/down.Nil) \tag{8.28}$$

which can then be proved using LegoMC.

8.3 Abstraction

One way to tackle with the state explosion problem is trying to replace a large system by a smaller abstract system which either is equivalent to original system and preserves the same properties or preserves the properties to be verified. The abstraction can be used to reduce the complexity of systems, and as a result, it is much simpler to verify properties of the abstract system. Lego can be used to prove that the abstraction reserves the properties to be verified, and then LegoMC can be used to prove the abstract finite-state model.

8.3.1 Strong Bisimulation

One of the popular equivalence relation used in verification is bisimularity [Par81, Mil83] that is an abstract equivalence on processes to state that two processes have the same operational behaviour. It can be used to transform a complicated model to its equivalent abstract model. There are several bisimulation relations and I am going to discuss only strong bisimulation in this thesis. It is believed the treatment of other bisimulation relations is similar.

The equivalence based on strong bisimulation is: P and Q are equivalent iff, for every action a, every a-derivative of P is equivalent to some a-derivative of Q, and conversely. This can be defined formally as follows.

Definition 8.3.1 [Mil89] A *strong bisimulation* S is a set of pairs of processes, such that whenever $(P, Q) \in \mathsf{S}$ implies, $\forall a \in Act$,
(i) Whenever $P \xrightarrow{a} P'$ then, $\exists Q', Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathsf{S}$
(ii) Whenever $Q \xrightarrow{a} Q'$ then, $\exists P', P \xrightarrow{a} P'$ and $(P', Q') \in \mathsf{S}$

The processes P and Q are *bisimular* if $(P, Q) \in$ some bisimulation.

Definition 8.3.2 [Mil89] P and Q are *strong equivalent*, written as $P \sim Q$, iff, $\forall a \in Act$,
(i) Whenever $P \xrightarrow{a} P'$ then, $\exists Q'.Q \xrightarrow{a} Q'$ and $P' \sim Q'$
(ii) Whenever $Q \xrightarrow{a} Q'$ then, $\exists P'.P \xrightarrow{a} P'$ and $P' \sim Q'$

Theorem 8.3.1 *If P and Q are strong equivalent, they will preserve the same properties [Mil89].*
If $P \sim Q$ then $\forall \Phi . P \models \Phi$ iff $Q \models \Phi$

Under this theorem, we can then use the abstract model instead of original model if we can prove they are strong equivalent.

8.3.2 Abstraction Mapping

Sometimes equivalence does not result in a significant reduction in the number of states. For some properties, abstract mapping should be enough if the abstraction preserves the properties. This approach is based on the observation that the specifications of systems usually involve only some components of the systems or simple relationships among the system components. It is particularly essential for verifying programs with properties related to data flow. If the properties we want to verify are data-independent, we can create an abstract model with only control flow. If the properties are data-dependant, we can still try to find an abstract data domain with fewer values which is enough to describe the relationship between original data values.

For a system with infinite data values, the specification could only involve simple relation among those data values. Therefore, instead of examining every data value, we can give a mapping between the actual data values and a smaller set of abstract data values (e.g. boolean) and then creates an abstract model of the original system. For example, assume we are only interested in if a natural number $x \in \mathcal{N}$ is 0, we can create a domain A_x with values $\{true, false\}$ and define a mapping h_x from $\mathcal{N}$ to A_x as follows:

$$h_x(d) = \begin{cases} true, & \text{if } x = 0 \\ false, & \text{if } x > 0. \end{cases}$$

As an example of abstract technique, we shall verify the same token ring example which is verified in section 8.1 by creating an abstract model under bisimularity.

8.3.3 An Example

$$I = pass.IT$$

$$IT = enter.exit.IT + \overline{pass}.I$$

$$Ring(n) = (IT|I|\ldots|I)\backslash\{pass\} \text{ with } n+1 \text{ } Is\text{— at least two processes}$$

where I is the idle workstation, IT is the workstation which holds the token.

If we regard $Ring(n)$ as a whole, we have another proof based on bisimulation. The successor state of $Ring(n)$ is through τ to $Ring(n)$ or through $enter$ to $Ring_{enter}(n)$. The only successor state of $Ring_{enter}(n)$ is $Ring(n)$. We can then find that the abstract model

$$Ring_{abst} = \tau.Ring_{abst} + enter.exit.Ring_{abst}$$

is a bisimular of Ring(n). The Bisimulation is

$$\{(Ring_{abst}, Ring(n)), (exit.Ring_{abst}, Ring_{enter}(n))\}$$

which can be formally proved in Lego. $Ring_{abst}$ has a finite state space and therefore we can use LegoMC to verify $Ring_{abst}$. That is, we integrate Lego with LegoMC by proving the abstraction relation in Lego and verifying the abstract system by LegoMC.

8.4 Discussion

We have demonstrated how our system is used to verify infinite state systems. The infinite counter is an example of evolving systems. The token ring is an example of parameterized systems. Although these two examples have infinite state space, their structures are very simple and therefore can be handled by CCS. For more complicated systems, the modeling will become complicated and therefore it is better to use imperative languages.

Our demonstrations combine semantics reasoning, induction, abstraction and composition methods with LegoMC to verify infinite systems. All of the lemmas and inference rules behind individual verification technique are formally proved in Lego and therefore form a coherent system that firmly ensures the correctness of proofs. More case studies have to be carried for the verification of more complicated systems.

Part IV

Proof Generation and Future Research

Chapter 9

Automatic Generation of Proof Terms

The core technique in our automation is the automatic generation of proof terms. Proof terms in type theory have their intended types that can be checked by simple type checking algorithms to ensure the correctness of the proofs. We can therefore implement more efficient and possibly more complicated algorithms to generate the proof terms without worrying about the correctness since the final results can always be checked by computers.

Although our techniques are experimented and implemented in the Lego proof checker, it is believed that these techniques are general and therefore can be adapted to other type theory based theorem provers very easily. Therefore, "Lego" in my following discussion can be regarded as general type theory based theorem provers such as Coq and Alf.

A general introduction to proof terms has been given in chapter 2. This chapter focuses on automation related issues. A general presentation about the construction of proof terms for assertions is given in section 9.1. Section 9.2 will focus on the automatic methods to construct proof terms. Some efficiency issues are discussed in section 9.3. Finally, some remarks are given in section 9.4.

Part of this chapter has been published in [YL98].

9.1 Proof Term Construction

Proof terms are λ-terms which are the proof objects in type theory. Logical formulas or propositions and logical inference in type theory are based on the idea of *propositions–as–types*, discovered by Curry [CF58] and Howard [How80]. According to this idea, any proposition P corresponds to a type **Prf**(P), the type of its proofs, and a proof of P corresponds to an object of type **Prf**(P). To assert that a proposition is true, one has to find (construct) a proof object of the proposition. For example, in an impredicative type theory, the disjunction $A \vee B$ is defined as

$$\forall C : Prop.(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C.$$

If a and b are proof objects of type **Prf**(A) and **Prf**(B), respectively, then the following is a proof object of type **Prf**$(A \vee B)$:

$$\lambda C : Prop \lambda h_1 : A \rightarrow C \lambda h_2 : B \rightarrow C.h_1\ a$$

The inference rules of logical connectives can be classified as *introduction* rules and *elimination* rules. The introduction rules reflects how to prove a logical formula with the operator concerned as the main connective, and the elimination rules reflect how to use such a formula to deduce other logical consequences. The proof objects of inference rules are listed as follows.

- True: The proof object of **true** is

$$PropId =_{df} \lambda X : Prop \lambda x : X.x$$

- False: $(\frac{\mathbf{false}}{\forall X.X})$ **false** has no proof objects in the empty context, and it implies every formula. if f is a proof of **false**, then

$$absurd\_elim(f, R) =_{df} f(R)$$

is a proof of R.

- $\wedge$–introduction: $(\frac{P_1, P_2}{P_1 \wedge P_2})$ If p_1 is a proof of P_1 and p_2 is a proof of P_2, then

$$pair(p_1, p_2) =_{df} \lambda X : Prop \lambda h : P_1 \to P_2 \to X.h(p_1, p_2)$$

is a proof of $P_1 \wedge P_2$.

- $\wedge$–elimination: $(\frac{P_1 \wedge P_2}{P_1}, \frac{P_1 \wedge P_2}{P_2})$ If h is a proof of $P_1 \wedge P_2$, then

$$fst(h) =_{df} h(P_1, \lambda p_1 : P_1 \lambda p_2 : P_2.p_1)$$

is a proof of P_1, and similarly

$$snd(h) =_{df} h(P_2, \lambda p_1 : P_1 \lambda p_2 : P_2.p_2)$$

is a proof of P_2.

- $\vee$–introduction: $(\frac{P_1}{P_1 \vee P_2}, \frac{P_2}{P_1 \vee P_2})$ If p_1 is a proof of P_1, then

$$inl(p_1) =_{df} \lambda X : Prop \lambda h_1 : P_1 \to X \lambda h_2 : P_2 \to X.h_1(p_1)$$

is a proof of $P_1 \vee P_2$. Similarly, if p_2 is a proof of P_2, then

$$inr(p_2) =_{df} \lambda X : Prop \lambda h_1 : P_1 \to X \lambda h_2 : P_2 \to X.h_2(p_2)$$

is a proof of $P_1 \vee P_2$.

- $\vee$–elimination: $(\frac{P_1 \vee P_2, P_1 \to R, P_2 \to R}{R})$ If h is a proof of $P_1 \vee P_2$ and r_i is a proof of $P_i \to R (i = 1, 2)$, then

$$or\_elim(h, r_1, r_2) =_{df} h(R, r_1, r_2)$$

is a proof of R.

- $\forall$–introduction: $(\frac{\Gamma, x:A \vdash P(x)}{\Gamma \vdash \forall x:A.P(x)})$ If p is a proof of $P(x)$ in $\Gamma, x : A$, then

$$\lambda x : A.P(x)$$

  is a proof of $\forall x : A.P(x)$.

- $\forall$–elimination: $(\frac{\forall x:A.P(x)}{P(a)})$ If p is a proof of $\forall x : A.P(x)$ and a is an object of type A, then

$$AllElim(p, a) =_{df} p(a)$$

  is a proof of $P(a)$.

- $\exists$–introduction: $(\frac{P(a)}{\exists x:A.P(x)})$ If p is a proof of $P(a)$, where a is an object of type A, then

$$ExIntro(a, p) =_{df} \lambda X : Prop \lambda h : (\forall x : A.P(x) \rightarrow X).h(a, p)$$

  is a proof of $\exists x : A.P(x)$.

- $\exists$–elimination: $(\frac{\Gamma \vdash \exists x:A.P(x) \quad \Gamma, x:A, p:P(x) \vdash R}{\Gamma \vdash R})$ If h is a proof of $\exists x : A.P(x)$ in Γ and r is a proof of R in $\Gamma, x : A, p : P(x)$, then

$$ExElim(h, r) =_{df} h(R, \lambda x : A \lambda p : P(x).r)$$

  is a proof of R in Γ.

- $\neg$–introduction: $(\frac{P \rightarrow \mathbf{false}}{\neg P})$ If r is a proof $P \rightarrow$ **false**, then r is a proof of $\neg P$.

- $\neg$–elimination: $(\frac{\neg P}{P \rightarrow \mathbf{false}})$ If r is a proof $\neg P$, then r is a proof of $P \rightarrow$ **false**.

Although proof terms provide a more concrete media for people to accept a proof, it is difficult to construct proof terms directly. Therefore, the common way to construct a proof term for an assertion at present is to communicate interactively with a theorem prover to construct the proof terms. Interactive theorem provers provide some guides to help the reasoning and finally construct proof terms. Users can choose the best strategy by their experience and heuristic knowledge. The users just have to follow the reasoning in their mind. They are not even conscious of proof terms. Some theorem provers have also implemented some tactics to do parts of the proof automatically.

An Example

We use a proof of the commutativity of disjunction to explain the process of proof term construction.

$$\frac{P_1 \vee P_2}{P_2 \vee P_1}$$

can be constructed by the following procedures. From P_1, we can infer $P_2 \vee P_1$ by **inr** rule and get the proof object $\lambda p_1 : P_1.inr(p_1)$. Similarly, from P_2, we can infer $P_2 \vee P_1$ by **inl** rule and get the proof object $\lambda p_2 : P_2.inl(p_2)$. From the above two scripts and the premise $P_1 \vee P_2$, we can conclude $P_2 \vee P_1$ by **or_elim** rule. The final proof term is therefore applying those rules as follows.

$$\lambda h : \mathrm{or}\ P_1\ P_2.\, or\_elim(h, \lambda p_1 : P_1.\, inr(p_1), \lambda p_2 : P_2.\, inl(p_2))$$

Some Remarks

The rigorous proofs in theorem provers usually include many trivial and tedious proof tasks. Our work shows that some decidable algorithms can be adapted to generate proof terms automatically, which will reduce the human effort in the process of doing formal proof. Compared with pure automatic tools, which use those decidable algorithms as well, automatic proof term generation can give not only the answer of "true" or "false" but also proof terms which can further ensure the correctness of the answer.

Automatic theorem provers use depth-first or width-first search method to search all possible solutions. Therefore the proof terms constructed by automatic algorithms are not the best in general, although they are automatically done by computers. Some automatic theorem provers can also integrate the heuristics to the search strategy and therefore improve the efficiency. Some may use data structure techniques or tricks to improve their algorithms. However, it is quite difficult to adapt those more efficient algorithms to generate proof terms and therefore certain changes of the original algorithms are necessary.

Unlike special purpose automatic tools, in which the correctness of tools usually is proved on papers, in our settings the inference rules or theorems behind the decidable algorithms have to be formally proved in a theorem prover. An automatic tool then applies those proven rules and follows the algorithms to construct the whole proof term. There are several ways to generate proof terms automatically in Lego, which are described individually in next section.

9.2 Automatic Generation of Proof Terms

There are at least four ways to generate the proof terms automatically in a type theory based theorem prover such as Lego.

1. Define computational functions of Lego to encode proof generation algorithms.
2. Use a tactical language.
3. Create an external program to generate Lego proof scripts which can be fed into Lego to generate proof terms.
4. Develop an external program to generate proof terms directly.

In the following sub-sections, we shall discuss these alternatives individually and analyse their advantages and disadvantages.

9.2.1 Use of Internal Functions

The first method is the simplest one since we implement algorithms for proof generation completely in Lego. One of the major features of type theory is that it is itself a computational language which enables us to do computing and logical reasoning in the same platform. By mixing the computational function with logic terms, parts of a reasoning task can be transformed into computations to simplify the reasoning task. This technique is similar to Oostdijk's *Internalization* [Oos98] presented in Type'98 conference.

If we want to prove formulas of the form $P(a)$ where $P : T \to Prop$ is a predicate over type T, we can try to find a function $f : T \to \text{bool}$ and prove the following theorem which state the correctness of the function:

$$bool2prop : \quad \forall a : T.(\text{Eq } f(a) \text{ true}) \to P(a),$$

For an object a of type T, if $f(a)$ can compute into **true**, the proof term of $P(a)$ is then

$$bool2prop(a, \text{Eq\_refl true}).$$

Once we have proved the correctness of function f and $f(a)$ can computes to `true` for an object a of type T, a proof of the form $P\ a : Prop$ can be transformed to `Eq true true` which have proof term `Eq_refl true`.

There are two essential jobs in the process of the transformation:

1. Find a suitable function f.

2. Prove the correctness of function f.

In our implementation of LegoMC, we used many internal functions such as computing successor states to simplify the construction of proof terms. Although this approach can simplify the subsequent proof process, the above two jobs are not easy in general. The first job is limited by the limitation of function definition in type theory. Some sorts of functions and techniques such as partial functions, dynamically programming and non-terminated functions are not definable. The second job involves complicated and tedious inductive reasoning.

Another problem with this approach is the computational speed of type theory based theorem provers is not satisfactory at present. However, this problem may be overcome gradually with the progress of the techniques of theorem provers.

9.2.2 Use of Tacticals

Lego, like some theorem provers such as Coq, has a simple tactical language. The tacticals are used to define sequences of commands to be executed. The syntax is as follows.

```
EXPRSN1 Then EXPRSN2    evaluate EXPRSN1, if evaluation succeeds,
                        evaluate EXPRSN2
EXPRSN1 Else EXPRSN2   evaluate EXPRSN1, if evaluation fails,
                        backtrack and evaluate EXPRSN2
For n EXPRSN            evaluate EXPRSN Then EXPRSN Then ...
                        (n times)
Repeat EXPRSN           evaluate EXPRSN Then EXPRSN Then ...
                        and backtrack last failure
Succeed                 this tactical doesn't do anything
Fail                    this tactical always fails
Try EXPRSN              evaluate EXPRSN and backtrack
                        if evaluation fails
(EXPRSN)                evaluate EXPRSN
```

For example, if we want to construct a proof term of $(\lambda x.\mathrm{Eq}\ a\ x \vee \mathrm{Eq}\ b\ x \vee \mathrm{Eq}\ c\ x)\ a$, we can use tactical

```
(Repeat (Refine inl)) Then (Refine inr).
```

This method is very slow and obviously not powerful enough to deal with more complicated algorithms.

9.2.3 Use of External Programs to Generate Proof Scripts

This method is to develop an external program to generate the proof scripts of Lego. These scripts can then be used to guide Lego to generate the proof terms. In developing LegoMC, which has been discussed in chapter 6, we first used this technique. With the aid of proof term construction mechanism in Lego, this technique is much easier than external proof term generation presented in the next sub-section.

For example, to use `or_elim` rule, we only have to generate proof script `Refine or_elim` without worrying about the parameters after `or_elim`. The unification mechanism in Lego will generate corresponding parameters automatically. However this method would be limited by the syntax of Lego commands. If the definition of Lego commands change, we have to change the external programs. Furthermore, it also has the limitation that it is too sensitive to the subtlety of the implementation of the command language of Lego.

9.2.4 External Programs to Generate Proof Terms

This method is to develop an external program by adapting automatic algorithms to directly generate the proof terms of Lego based on the proof generation techniques. The task consists of two phases.

1. Formally prove the axioms and inference rules behind the algorithms.

2. Adapt the algorithms to generate proof terms by applying the inference rules.

Because many existing algorithms are based on classical logics, we have to change the axioms and inference rules to meet the intuitionistic logics of type theory. Obviously the algorithms have to be changed as well. Once their inference rules have been proved formally in Lego, the generation of proof terms is then applying those inference rules recursively. Compared with the other methods, direct proof term generation is most complicated and difficult. However, it has the best speed performance and transferability. We can change the syntax generation part to fit to other theorem provers. In practice, there are many efficiency issues to be dealt with. We discuss them in the next section.

9.3 Efficiency Issues

Many decidable algorithms have been improved using some heuristics, tricks or the techniques of programming such as dynamic programming, data structure, etc. It is not easy to adapt those more efficient algorithms to generate proof terms. Another problem is that the functionities of external programming language can be different from Lego's and therefore can have different formalisations between these two platforms. In our experience, the efficiency issues can be partly overcome by reducing the size of proof terms.

The Size of Proof Terms

The proof terms which are generated by automatic tools are usually very big. This would be a problem because type-checking these generated proof terms can be very slow. Therefore, we must develop methods and techniques to reduce the size. In our work of automatic proof generation, the following four ways of size reduction have been used:

1. Abbreviation

   In automatically generated proof terms, there are usually many repeated sub-terms. Instead of fully expansion of proof terms, making definitional abbreviations can usually reduce the size effectively.

2. Internal functions

   Another useful way is using internal functions as mentioned in previous section to transform a complicated sub-term to be a function name with parameters.

3. Pre-proved lemmas

   Pre-proved lemmas are also a good way to reduce the size but we need to find essential lemmas in advance.

4. On-the-fly lemma generation

   The last but very important technique is what we call *on-the-fly lemma generation*. It can not only reduce the size of proof terms but also increase the efficiency of algorithms. We discuss this method in an independent paragraph.

The idea of on-the-fly lemma generation is very simple. For example, Fig 9.1 is a part of a proof tree with nodes as formulas to prove during the proof search process. Node a is like $\vee$ or $\wedge$ formulas with two branches. After several intermediate nodes, it is possible that two branches meet again at node b. The proof after b is the same for two branches and therefore the second branch can re-use the results from the first branch.

More precisely, we can store the proofs of some critical sub-trees as lemmas and refer to those lemmas in the subsequent proof processes. On the one hand, we can save the time to repeat the identical proofs. On the other hand, by referring to on-the-fly generated lemmas, we do not have to insert the same poof sub-term and therefore reduce the size of the whole proof term.

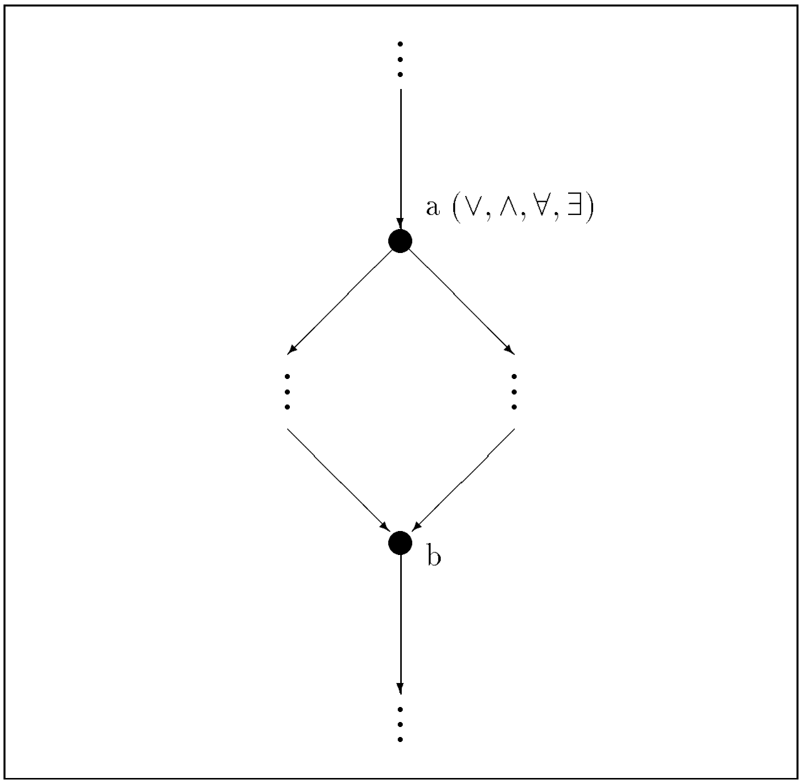

Figure 9.1: On-the-fly lemma generation

9.4 Discussion

In practice, it is better to combine the internal functions with external programs to get the benefits of both sides. The internal function can be used to define simple functions and external programs can be used to define more complicated functions or some functions which are undefinable in Lego. Moreover, further improvement of internal functions in type theory would be helpful to further simplify proof term generation. The implementation of computation in theorem provers is also essential for better speed performance.

The computational power of type theory has many advantages over other theorem provers. Because part of the proof can be transformed to computational function, the proof process can be simplified. It is also helpful for developing external programs to generate proof terms since computational functions simplify the proof terms.

One challenge for the correctness is that the computation mechanism and type checking is implemented by computer programs which may contain bugs as well. One way to further increase our confidence is to check the proof by different type theory based theorem provers. Because they all use λ-terms as proof objects, by syntax translation, we can put the same proofs to be checked by different theorem provers to make us more confident about the proofs.

Chapter 10

Other Automation Issues

Most model-checkers have difficulties to deal with properties which are highly data-dependent. One way to tackle such problems is to integrate other automatic techniques which are suitable for data manipulations, e.g. arithmetic decision procedures. Because most of the decision procedures for arithmetic manipulations are based on classical logic, those techniques are very difficult to be adapted to generate proof terms for general propositions of type theory, which is based on intuitionistic logic. However, it is no problem to use those procedures to deal with the boolean data type that is enough for us to verify properties related to data types.

The general form of boolean properties is like:

$$\Gamma \vdash \ \mathrm{Eq}\ p\ \mathrm{true}$$

where p is a boolean expression. The proof process is then trying to simplify p to boolean constant "true" by a sequence of inference rules. p can contain boolean operators and natural number operators and therefore the simplification process includes boolean expression simplification and natural number expression simplification. This chapter describes the possibility of applying proof term generation techniques described in previous chapter to do these simplifications. Some preliminary research and experiments are done but the implementation is beyond the scope of this thesis.

Many decision procedures for data domain are based on equational rewriting. Therefore, I will discuss the equality in type theory and equational rewriting techniques based on proof term generation in next section. In section 10.2, *Binary decision diagram* (BDD) technique is discussed which has been claimed as an efficient technique to manipulate boolean expressions or propositional formulas. In section 10.3, *arithmetic decision procedures* are discussed which can deal with arithmetic operation of natural numbers.

10.1 Equational Rewriting

There are two sorts of equalities: *intensional equality* and *extensional equality*. The computational equality is intensional and means that two objects are equal if and only if they can be computed to the same canonical objects. For the computational equality, rewriting can be done automatically through computation, and we do not have to develop an external program to do its rewriting. For instance, the following example is taken from Boulton's PhD thesis [Bou93]; there he implemented some conversion functions to rewrite the terms. Suppose we want to rewrite the term:

$$\lambda n.(n * 0) + (n * 4) \tag{10.1}$$

using the following equations:

$$x * y = y * x \tag{10.2}$$

$$0 * x = 0 \tag{10.3}$$

$$0 + x = x \tag{10.4}$$

In type theory, Eqn. 10.3 and Eqn. 10.4 are computational equalities which mean the terms in the left hand side of the equations and the terms in the right hand side can be computed into the same canonical objects. Therefore, we do not have to develop any external programs for computational equalities.

On the other hand, extensional equality concerns equality between functions. A notion of equality is extensional if for any functions f and g of the same type, f equals g provides that $f(a)$ and $g(a)$ are equal for every object a of the domain type. For example, Eqn. 10.2 corresponds to the following two functions:

$$times_1 =_{df} \lambda x : N \lambda y : N.x * y \quad \text{and}$$
$$times_2 =_{df} \lambda x : N \lambda y : N.y * x$$

which are extensionally equal since they return the same value for every two natural numbers but not intensionally equal because they are different canonical terms. For this sort of equality, we do need explicit term rewriting.

Propositional Equality

The propositional equality in type theory can be defined as the *Leibniz equality* which means that two objects of the same type are equal if and only if they cannot be distinguished by any property.

Definition 10.1.1 (Leibniz equality) Let A be a type. The Leibniz equality over A, notation $=_A$, is the binary relation over A defined as follows:

$$=_A \quad =_{df} \quad \lambda x : A \lambda y : A.\forall F : A \to \text{Prop}.F(x) \to F(y).$$

We shall write $a =_A b$ for $=_A (a, b)$.

Leibniz equality has the following properties.

- Eq_refl $= \lambda t : T \lambda P : T \to \text{Prop} \lambda h : Pt.\ h$
  $: \forall t : T.t =_T t$

- Eq_sym $= \lambda t, u : T \lambda g : t =_T u.g(\lambda x : T.x =_T t)(\text{Eq\_refl } t)$
  $: \forall t, u : T(t =_T u) \rightarrow u =_T t$

- Eq_trans $= \lambda t, u, v : T \lambda p : t =_T u \lambda q : u =_T v \lambda P : T \rightarrow \text{Prop}.\lambda x : T.(q\ P\ (p\ P\ x))$
  $: \forall t, u, v : T.(t =_T u) \rightarrow (u =_T v) \rightarrow t =_T v$

Explicit Rewriting

The rewriting is based on the substitution rule: for any predicate P over a type A, if formulas $P(a)$ and $a =_A b$ are provable, then so is $P(b)$.

$$\begin{aligned} \text{Eq\_subst} &= \lambda a, b : A \lambda h : \text{Eq } a\ b \lambda P : A \rightarrow \text{Prop} \lambda p : P(a).P(b) \\ &: \forall a, b : A(\text{Eq } a\ b) \rightarrow \forall P : A \rightarrow \text{Prop} P(a) \rightarrow P(b) \end{aligned}$$

In other word, for any proof p of $P(a)$ and proof h of $a =_A b$,

$$\text{Eq\_subst}(a, b, h, P, p)$$

, which can compute into $h(P, p)$, is a proof of $P(b)$.

In the above example, we only have to rewrite Eqn. 10.2 which can be achieved by apply

$$\text{Eq\_subst}(x * y, y * x, Eqn.10.2)$$

There is a command `Qrepl` in Lego to do this rewriting. However, since we want to generate the proof term externally, we have to use the above `Eq_subst` rule.

Although computational function can perform rewriting automatically, sometimes we would still use explicit term rewriting for computing speed concern. The computational function is defined by recursive function that takes a long time while computing a big term such as larger natural numbers. For instance, the following equation can be proved simply by `Eq_refl` rule.

$$1234 * 4321 = 4321 * 1234$$

However, the computer will need quite a long time to check this proof. Instead we can use the above rule to rewrite the equation to

$$4321 * 1234 = 4321 * 1234$$

and then apply `Eq_refl` rule.

10.2 BDD Propositional Simplifier

The boolean properties for imperative programs in chapter 5 can be defined as follows.

$$\text{Bool}(b, s) = \text{Eq (eval } b \text{ (memory } s)) \text{ true}$$

where b is a boolean expression, s is a state, `eval` and `memory` are defined in chapter 5. For any boolean expression b, `eval` b `(memory` s`)` will compute to a boolean value `true` or `false`. Therefore, to prove a boolean property which has a true value is just to apply the `Eq_refl` rule. However, for complicated boolean expressions, which are very common in real life systems, it takes a very long time for type theory based theorem prover to compute the value of a boolean expression.

The efficiency of *Binary Decision Diagrams (BDDs)* technique to deal with boolean computation draws the attention of many researchers in this decade. Therefore, BDD technique should be a considerable alternative to computational functions. This section describes the possibility to apply proof term generation to create a BDD propositional simplifier. Harrison has implemented BDD technique in HOL [Har95] that can be a good reference for the implementation issue.

BDD

The basic idea of BDDs is to build up a "binary decision diagram" with the variables at the nodes and either 1 (true) or 0 (false) at the leaves. Each node has two branches which represent the expressions formed by substituting the variable to be true or false, respectively. The increase of efficiency of BDDs actually comes from techniques of variable ordering [Lee59, Ake78] and variable reducing [Bry86] .

Variable reducing is using directed acyclic graphs to share common sub-expressions in the diagram. Variable ordering is choosing a better ordering of the variables to reduce the size of the BDDs produced. Some heuristics on variable ordering can be found on the paper by Bulter, Ross, Kapur and Mercer [BRKM91]. Such structures are formally called as *reduced ordered binary decision diagrams* (ROBDDs) but are usually called BDDs for short.

The construction of a BDD from a boolean expression proceeds as follows. The nodes of the BDD are constructed in the same order as they would be visited by a depth-first traversal of the boolean expression. Whenever a new node is to be created,

1. check whether it is equivalent to a node already created by looking among the existing nodes with the same variable as label and checking whether they have 0- and 1- branches identical to the branches of the node under consideration.

2. check whether the two branches are identical and the node therefore superfluous.

The purpose of the first step is to reduce the store space and the size of proof terms. Since it is equivalent to a node which has been traveled, we can use on-the-file lemma generation as described in previous chapter to use the stored lemma.

The second step is used to eliminate variables.

Following our approach of proof term generation, we have to find the inference rules behind the BDD reasoning and then implement a program to mimic the BDD algorithm to generate proof terms by applying those inference rules.

Equivalence Rules

Before introducing the equivalence rules, some notations have to be defined first.

Definition 10.2.1 (If Normal Form) $t \to t_0, t_1$ is an *if-then-else* operator defined by

$$t \to t_0, t_1 =_{df} (t \wedge t_0) \vee (\neg t \wedge t_1)$$

where t is called *test expression*. All operators can be expressed using only the if-then-else operator and the constants 0 and 1 that represent "false" and "true" correspondingly. An *If-then-else Normal Form* (INF) is a boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.

The basic BDD algorithm is based on the following two equivalence rules.

- Any boolean expression is equivalent to an INF $(t \to x1, x2)$ which can be done by transformation rule `bdd_trans`.

  $$\text{bdd\_trans} : \forall t : \mathit{bool} \to T \forall x : \mathit{bool}.\text{Eq } t(x) \text{ (if } x \text{ } t(\text{true}) \text{ } t(\text{false}))$$

- $(x \to y, y)$ is equivalent to y

  $$\text{if\_absorbs} : \forall b : \mathit{bool} \forall x : T.\text{Eq (if b x x) x}$$

Having the above equivalence rules, we can design a program to mimic BDD procedures that can automatically apply those equivalence rules to simplify propositional formulas. The procedure is to rewrite the variable using `bdd_trans` rule repeatedly to transform a boolean expression to its equivalent INF and then use *if_absorts* rule to reduce the variable until it reaches "true" or "false".

Example

This example shows $x \vee \neg x$ can be simplified to "true".

$$\forall x : bool.\ \text{Eq}\ (orelse(x, inv(x)))\ \text{true} \tag{10.5}$$

Proof Rewrite "Eq $(orelse(x, inv(x))$ true" by
bdd_trans $(\lambda x : bool.(orelse(x, inv(x)))x$, the goal becomes

$$\forall x : bool\ \text{Eq if}\ x(orelse(true, inv(true)))(orelse(false, inv(false))\ \text{true}$$

Because *orelse* and *inv* are both computational function, both `(true.orelse true.inv)` and `(false.orelse false.inv)` are computed to canonical object "ture". Therefore, the goal become

$$\forall x : bool\ \text{Eq (if}\ x\ \text{true true) true.}$$

By rule if_absorts, we can rewrite `(if x true true)` by `true`. The goal becomes

$$\forall x : bool\ \text{Eq true true}$$

which can be proved by `Eq_refl` rule.

10.3 Arithmetic Decision Procedures

These are some decision procedures to prove arithmetic relation, array and tuples automatically. We can use the same method to develop a program to mimic those decision procedures to generate proof terms.

The decision procedure discussed here is based on the variable elimination method for natural numbers, which has been implemented by Boulton in HOL

[Bou93]. Boulton's procedure operates in two phases. The first phase normalises the negation of the formula. The second phase is trying to simplify the normalised term until "true" or "false". The first phase proceeds as follows:

1. The logical implications and equivalences are replaced by conjunctions, disjunctions and negations.
2. The negations are then pushed as far down the term as possible.
3. The term is put into disjunction normal form (DNF).
4. Each disjunction is now the conjunction of inequalities which can then be normalised to $\leq$ inequalities.
5. The resulting inequalities are further normalised so that each variable appears at most once in each inequality, and on the appropriate side of the relation for its coefficient to be positive.

The second phase of the procedure tries to eliminate variables until all of the inequalities contain only constants which can then be evaluated to either true or false. A false inequality completes the proof of a conjunction. A true inequality can be discarded. If all of the disjunctions are proved to be false which means the negation of the original formula is false. We can then conclude that the original formula is true.

Inference Rules

The normalisation to DNF is based on the following equivalence rules.

$$x \Rightarrow y = \neg x \vee y$$
$$x \Leftrightarrow y = (\neg x \vee y) \wedge (\neg y \vee x)$$
$$\neg\neg x = x$$

$$\neg(x \wedge y) = \neg x \vee \neg y$$
$$\neg(x \vee y) = \neg x \wedge \neg y$$

This algorithm normalises each inequality to $\leq$ inequalities using the following rules.

$$(m < n) = (1 + m \leq n)$$
$$(m = n) = (m \leq n) \wedge (n \leq m)$$
$$(m > n) = \neg(m \leq n)$$

The second phase, variable elimination, is based on the following rules.

$$\forall a : nat \quad m \leq n = a * m \leq a * n \tag{10.6}$$
$$\forall a : nat \quad m \leq n = a + m \leq a + n \tag{10.7}$$
$$\forall p, q : bool \quad p \wedge q = p \wedge p \wedge q \tag{10.8}$$
$$\forall a, b : nat \quad a + b = b + a \tag{10.9}$$
$$\forall a, b, c, d, e, f, g : nat \quad (a \leq b + c) \wedge (c + d \leq e) = a + d \leq b + e \tag{10.10}$$

The above equivalence rules have been formally proved in Lego. The implementation is then following the algorithm and applying the above equivalence rules to generate proof terms. Here is an example to show the procedure for variable elimination.

An Example

Suppose we want to prove

$$\mathrm{Eq}\ (m + n \leq p) \wedge (2m \leq 1 + n) \wedge (3 + p \leq 3m)\ false,$$

the proof can be proceeded as follows.

Proof We want to rewrite

$$(m + n \leq p) \wedge (2m \leq 1 + n) \wedge (3 + p \leq 3m)$$

to be "false" so that we can use Eq_refl to finish the proof. Rewrite by the Eqn. 10.10, we can get the following goal:

$$(m + 2m \leq p + 1) \wedge (3 + p \leq 3m)$$

We rewrite the goal by Eqn. 10.10, we can get the following goal:

$$(m + 2m + 3 \leq 3m + 1)$$

which is

$$(3m + 3 \leq 3m + 1)$$

Rewrite by Eqn. 10.7, we can get

$$(3 \leq 1)$$

which can be computed to "false".

Chapter 11

Conclusion

In the conclusion, a summary of the thesis is presented. Finally, possible future researches are suggested.

11.1 Summary

We have described a formal development of a framework for the verification of concurrent programs in Lego system. Since we use CCS and an imperative language as the description language, the system modeling job is pretty easy. Additionally, LegoMC can do the verification of finite systems automatically which improve the efficiency dramatically. Infinite system verification can be carried on Lego with the aid of LegoMC for parts of proofs.

The second part of this thesis describes how we formalise description languages and specification languages in Lego. One essential feature of our framework is that all of the inference rules are formally proved and proof-checked in Lego. This feature ensures the correctness of individual verification methods and the consistency of their integration. The description languages are formalised by deep embedding, whereas specification language, i.e. μ-calculus, is formalised by shallow embedding which make the extension of specification languages and the integration of various verification methods easier.

The third part is focused on LegoMC. We have successfully developed a model checker, LegoMC, which can be used to verify finite state systems and generate proof terms automatically. The interface of LegoMC allows users to use the syntax which they are familiar with. We have used LegoMC to analysis a transport protocol. Several mutual exclusion algorithms have also been verified. We also use two infinite state examples to demonstrate the integration of various verification methods within this verification environment.

The final part summaries the techniques of automatic proof term generation that we got during the development of LegoMC. Besides model-checking, we have also studied the application of proof term generation to Binary Decision Diagram and arithmetic decision procedures.

In practice, we can use LegoMC as a debugging tool that can be used to find the counter example for a false system. After several modification and then getting a satisfactory system, we can then use LegoMC to generate proof terms which are then put into Lego to be type checked to further ensure the correctness of the results.

11.2 Future Research

This thesis has created a solid base for an effective and efficient domain-specific verification environment based on general-purpose theorem provers. The future work will further enrich this framework to be able to deal with wider classes of systems.

Extension with Real Time, Data Type and Functions

The current system for CCS is pure CCS (without value passing), it can be further extended to value pass CCS. The data type of ICPL is only natural number, we can extend it with more data types or abstract data types (e.g. array, list, etc.) and real time reasoning. Furthermore, to be able to incorporate future changes and extensions to other description languages, we can develop a tool similar to [Bou96] so that we can formalise languages in Lego from given specifications of the syntax and the operational semantics of the languages. Besides the formalisation, this tool should generate the relevant theorems and computational functions necessary to reason about the logical properties.

Implemention of Other Automatic Techniques and the Integration with Model Checking

In chapter 10, we have proved the inference rules of the BDD and arithmetic decision procedures. Further work would be implementing a suitable algorithm based on our proof term generation techniques in chapter 9. Certain adjustment would be necessary to fit into type theory setting. How to integrate BDD and arithmetic decision procedures with LegoMC to form a powerful tool is also a challenging research direction.

Efficiency and Background Proof Generation

Efficiency is a key issue in automated proof generation. This can be tackled in at least two aspects. First, more efficient algorithms can be investigated and implemented. Using the technique of on-the-fly lemma generation described in chapter 9, we have improved the efficiency of LegoMC dramatically: the state-space of a problem with several hundred thousand travel states was reduced to just several thousand. This is very promising, and we can continue this investigation and study its suitability in a more general context.

Secondly, we can explore another important idea – *background proof generation.* As some of the work experience shows (both in our work in type theory and the work in HOL [Bou93], for example), a fully expansive theorem prover could be less efficient in general. In the interactive environment of proof development, automated proof generation can be done in the background. In other words, we can use multi-thread techniques to hide the proof term generation in the background and keep the interactive user interface progressing fluently by means of fast classical methods. Theoretic justification of such a technique should be studied; for example, we can show that a decidable formula is true by a classical BDD method if and only if there is a proof of the formula in the type theory.

Composition Rules and Equivalence Rules of ICPL

Some of inference rules such as composition rules and equivalence rules of CCS have been proved and used to verify small infinite-state systems in chapter 9. Since ICPL is easier and clearer to describe more complicated systems, it would be interesting to develop composition rules and equivalence rules for infinite-state systems modeled in ICPL. We can also consider larger examples and case studies to demonstrate how different methods can be combined to solve problems efficiently.

Domain Specific Interface

The current human machine interface of LegoMC is a domain-specific interface with purely the syntax of description languages and specification languages. Therefore general programmers and system analysts can easily use this system to do verification by the syntax which they are familiar with instead of learning Lego. However, LegoMC can only handle finite state systems and therefore users have to use Lego if they want to verify infinite state systems. It is expected an integrated domain-specific interface for various verification techniques, both finite and infinite state, will further extend the usage of this framework.

Proof Explanation

Automatic theorem proving techniques have the advantage of efficiency. We can use them to verify a lot of complicated systems automatically. However, one drawback is no explanation about the proof. Some model checkers can generate counter examples when a system fail on the desired property but no clue about the proof once the system do satisfy the property. Our verification environment based on proof terms has the potential to tackle this problem by extract proof explanation from generated proof terms.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *proceedings of the 5th Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, June 1990.

[ACN90] L. Augustsson, Th. Coquand, and B. Nordström. A short description of another logical framework. In G. Huet and G. Plotkin, editors, *Preliminary Proc. of Logical Frameworks*, 1990.

[AH90] R. Alur and T. Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, June 1990.

[Ake78] S.B. Akers. Binary decision diagrams. *ACM Transations on Computers*, C-27:509–516, 1978.

[And92] H.R. Andersen. Model checking and boolean graphs. In B. Krieg-Bruckner, editor, *Proceedings of the Fourth European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer Verlag, February 1992.

[And93] H.R. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Aarhus University, Denmark, June 1993.

[Apt81] K. R. Apt. Ten years of Hoare's Logic. *ACM Transactions on Programming Language and System*, 3:431–483, 1981.

[ASW94] Henrik Reif Andersen, Colin Stirling, and Glynn Winskel. A compositional proof system for the modal μ-calculus. Technical Report RS-94-34, BRICS, 1994. Extended abstract appears in Proc. LICS'94.

[AW92] H.R. Andersen and G. Winskel. Compositional checking of satisfaction. *Formal Methods In System Design*, 1(4), 1992.

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st Design Automation Conference*, pages 596–602. Association for Computing Machinery, June 1994.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Bou93] R.J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report TR337, University of Cambridge Computer Laboratory, 1993. Author's PhD thesis.

[Bou96] R.J. Boulton. A tool to support formal reasoning about computer languages. Technical Report TR405, University of Cambridge Computer Laboratory, 1996.

[BRKM91] K.M. Butler, D.E. Ross, R. Kapur, and M.R. Mercer. Heuristics to compute variable orderings for efficient manipulation of Ordered Binary Decision Diagrams. In *Proceedings of 28th ACM/IEEE Design Automation Conference*, pages 417–420, 1991.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS92] Julian Bradfield and Colin Stirling. Local model checking for inifinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

[C+86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Pretice-Hall, 1986.

[CE81] E.M. Clarke and E.A. Emerson. *Design and synchronization skeletons using Branching Time Temporal Logic*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, Berlin, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transition on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CF58] Haskell B. Curry and Rebort Feys. *Combinatory Logic*. volume I. North-Holland, 1958.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D.L. Dill, editor, *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Stanford, CA, June 1994. Springer Verlag.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, 1992.

[CH88] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[Cle90] R. Cleaveland. tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1990.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision process and regular events. In *Proc. 17th Int. Coll. on Automata Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 336–349, Coventry, July 1990.

[D$^+$91] G. Dowek et al. *The Coq Proof Assistent: User's Guide (version 5.6).* INRIA-Rocquencourt and CNRS-ENS Lyon, 1991.

[Dam90] Mads Dam. Translating CTL* into the modal μ-calculus. Technical Report ECS-LFCS-90-123, LFCS, University of Edinburgh, November 1990.

[Dam95] Mads Dam. Compositional Proof System for Model Checking Infinite State Processes. In I. Lee and S.A. Smolka, editors, *Proc. CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26, Philadelphia, Pennsylvania, USA, August 1995. Springer-Verlag.

[dB72] Nicolaas G. de Bruijn. Lamda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.

[dB80] J.W. de Bakker. *The Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

[Dij65] E.W. Dijkstra. Cooperating sequrntial processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1965.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J.W. de Bakker and

J. van Leeuwen, editors, *Automata, Languages and Programming, ICALP'80*, volume 85 of *lncs*, pages 169–181. Springer-Verlag, 1980.

[EGL92] Urban Engberg, Peter Gronning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G.V. Bochmann and D.K. Probst, editors, *Computer-Aided Verification 92*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, 1992.

[EH86] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of ACM*, 33(1):151–178, 1986.

[EL85] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery. also in Proceedings of the First Annual Symposium om Logic in Computer Science, IEEE Computer Society Press, June, 1986.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In *Proceedings of Symposia in Applied Mathematics XIX*, pages 19–32. American Mathematical Society, 1967.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Computer Aided Verification, Proc. 5th Int. Workshop*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84, Elounda, Greece, June/July 1993. Springer Verlag.

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

[GW93] P. Godefroid and P. Wolper. Partial-order methods for temporal verification. In *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, August 1993. Springer-Verlag.

[Har95] John Harrison. Binary Decision Diagrams as a HOL Derived rule. *The Computer Journal*, 38(1), 1995.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *proceedings of the 5th Symposium on Logic in Computer Science*, pages 402–413, Philadelphia, June 1990.

[Hoa69] C.A.R. Hoare. A axiomatic basis for computer programming. *Communication of the ACM*, 12:576–583, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol81] G.J. Holzmann. Pan: a protocol specification analyzer. Technical Report TM81-11271-5, AT&T Bell Laboratories, March 1981.

[Hol85] G.J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64:2413–2434, December 1985.

[How80] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Es-*

says on combinatory logic, lambda calculus and formalism. Academic Press, 1980.

[Hym66] H. Hyman. Comments on a problem in concurrent programming control. *Comm. ACM*, 9(1):45, 1966.

[JS93] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking Bdd-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference.* Association for Computing Machinery, 1993.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification 93*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Elounda, Greece, June/July 1993. Springer Verlag.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *8th ACM Symposium on Principles of Distributed Computing*, pages 239–248, Edmonton, Albera, Canada, August 1989.

[Knu66] D.E. Knuth. Additional Comments on a Problem in Concurrent Programing Control. *Comm. ACM*, 9(5), 1966.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, Princeton, New Jersey, 1994.

[Lam80] L. Lamport. Sometimes is sometimes "not never" $\rightarrow$ on the temporal logic of programs. In *Proc. 7th Ann. ACM Symp. on Principles of Programming Languages*, pages 174–185, 1980.

[Lam86] L. Lamport. The Mutual Exclusion Problem Part II - Statement and Solutions. *J. ACM*, 33(2), 1986.

[Lar90] Kim G. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *Theoretical Computer Science*, 72:265–288, 1990.

[Lee59] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[Lon93] D.L. Long. *Model Checking, Abstraction, and Compositional Reasoning.* PhD thesis, Carnegie Mellon University, 1993.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Mag92] L. Magnusson. The new implementation of ALF. In *Informal Proceedings of Workshop on Logical Frameworks*, Bastad, 1992.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MN95] Olaf Müller and Tobias Nipkow. Combining Model Checking and Deduction for I/O-Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[Oos98] Martijn Oostdijk. Proof automation in type theory by internalization. In *Preliminary Programme of Type 98 workshop*, Kloster Irsee, Germany, 1998.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automted Deduction(CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Proc. of 5th GI Conf.*, volume 104 of *Lecture Notes in Computer Science*, 1981.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Proc. Lett.*, 12(3), 1981.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th International Coll. on Automata, Languages and Programming*, pages 15–32. Springer-Verlag, Berlin, 1985.

[Pol94] Randy Pollack. *Incremental Changes in LEGO: 1994*, May 1994. Available by ftp with LEGO distribution.

[Pol95] Robert Pollack. A Verified Typechecker. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, Edinburgh, 1995. Springer-Verlag.

[PZ86] A. Pnueli and L. Zuck. Probabilistic verification by tableaux. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proc. 5th Internat. Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer, Berlin, 1981.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof checking. In *Computer Aided Verification, Proc. 7th Int. Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liège, Belgium, July 1995. Springer-Verlag.

[Sti85] C. Stirling. A complete compositional modal proof system for a subset of CCS. In *volume 194 of LNCS*, pages 475–486. Springer-Verlag, 1985.

[Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[SW91] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 25(2):285–309, 1955.

[Var85] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 327–338, Portland, October 1985.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceed-*

ings, Symposium on Logic in Computer Science, pages 332–344, Cambridge, Massachusetts, June 1986. IEEE Computer Society.

[Wal87] David Walker. Introduction to a calculus of communicating systems. Technical Report ECS-LFCS-87-22, LFCS, University of Edinburgh, 1987.

[Wal89] David Walker. Automated analysis of mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-89-91, LFCS, University of Edinburgh, 1989.

[Wal95] Igor Walukiewicz. Notes on the propositional μ-calculus: Completeness and related results. Technical Report NS-95-1, BRICS, Denmark, 1995.

[Win85] Glynn Winskel. On the composition and decomposition of assertions. Technical Report TR-59, Computer Laboratory, University of Cambridge, 1985.

[Win89] Glynn Winskel. A note on model checking the modal ν-calculus. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 761–772. Springer-Verlag, 1989.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 1989. Springer-Verlag.

[YL97] Shenwei Yu and Zhaohui Luo. Implementing a Model Checker for LEGO. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 442–458, Graz, Austria, September 1997. Springer-Verlag.

[YL98] Shenwei Yu and Zhaohui Luo. Automated Proof Term Generation. In *Typr'98*, Kloster Irsee, Germany, 1998.