Implementing a Model Checker for LEGO

Shenwei Yu and Zhaohui Luo**

Department of Computer Science, University of Durham,
South Road, Durham, DH1 3LE, UK

Abstract. Interactive theorem proving provides a general approach to modelling and verification of both hardware and software systems but requires significant human efforts to deal with many tedious proofs. To be effectively used in practice, we need some automatic tools such as model checkers to deal with those tedious proofs. In this paper, 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 and infinite 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 models as well.

1 Introduction

Interactive theorem proving gives a general approach to modelling 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-processes mutual exclusion problem is fairly complicated to verify. On the other hand, model checking is automatic but limited to certain problems - i.e., simple finite state processes, although this limitation can be partially overcome to deal with more complicated problems by improving the efficiency through BDD techniques [4]. Since theorem proving and model checking are complementary techniques, both schools have been trying to combine the strength of these two approaches by using theorem provers to reduce or divide the problems to ones which can be checked by model checkers.

Wolper and Lovinfosse [26] and Kurshan and McMillan [14] extended model checking for inductive proofs by using an invariant to capture the induction hypothesis in the inductive step. Joyce and Seger [11] used HOL theorem prover to verify formulas which contain uninterpreted constants as lemmas which are verified by Voss's model checker. Kurshan and Lamport [12] proved a multiplier where the 8-bit multiplier can be verified by COSPAN model checker [13] and the n-bit multiplier composed from 8-bit multipliers can be verified by TLP theorem prover [10]. In principle, these approaches are to divide the whole problem to

* Email address:{Shenwei.Yu, Zhaohui.Luo}@durham.ac.uk

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 [20] 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 [21] 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 [24]. However, the correctness of model checkers is still a big concern since they themselves are computer softwares 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 [16], ALF [1, 17], Coq [8] and Nuprl [6], 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. Our work is to implement a model checker for LEGO by producing proof terms. One of the major contributions of our model checker is the automatic generation of proof terms so that we can enhance the efficiency of verification in a general theorem prover, LEGO.

We use the Calculus of Communicating Systems (CCS) [19], a message-passing concurrent language, to model the systems and propositional μ-calculus to express the system properties. Both CCS and propositional μ-calculus are formalised in LEGO for both finite and infinite state systems. Our model checker (LegoMC) is an independent program which takes the syntax of CCS and propositional μ-calculus in LEGO and then returns a string which is a proof term in the syntax of LEGO. We can therefore integrate this proof term with other proof terms to complete a larger proof. This system can also deal with other temporal logics by giving their abbreviations in μ-calculus. Furthermore, the domain model can be changed to imperative languages as well. The system structure is shown in Fig. 1.

Using this system, we have successfully verified some finite state processes automatically such as the ticking clock, the vending machine, 2-process mutual exclusion .... For infinite state problems, we have verified a n-process mutual exclusion problem by reducing the model to a finite state abstract model. LEGO is used to prove that the abstract model preserves the property of the original model, and LegoMC is used to verify the abstract model. We have also verified some finite examples in an imperative language such as Peterson's algorithm and the dining philosophers problem.

In the following section, a brief introduction to CCS and propositional μ-calculus is given. Their formalisation in LEGO is presented in section 3. The implementation of the model checker is discussed in section 4. Section 5 presents an example of a n-process token ring network. In section 6, we discuss the exten-

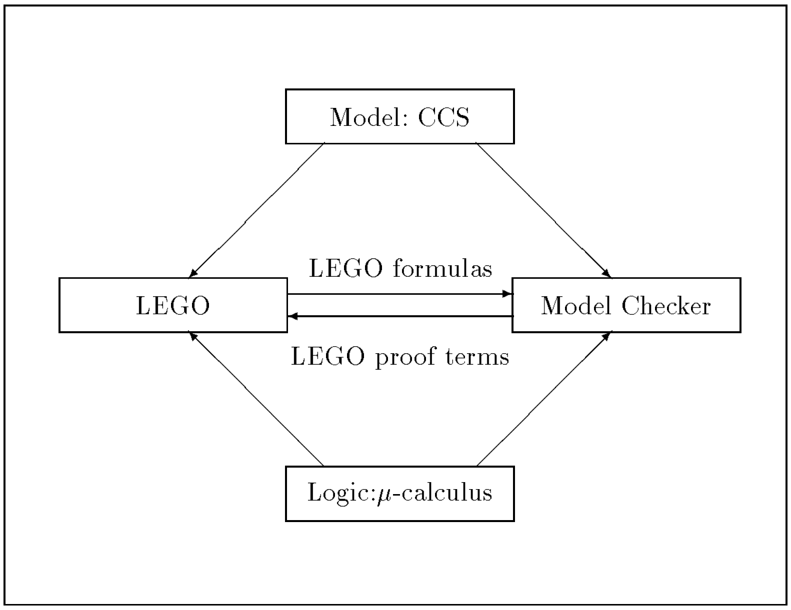

Fig. 1. The system structure of LegoMC

sion to an imperative programming language. Conclusions are given in section 7.

2 Model and Logic

2.1 CCS: Calculus of Communicating Systems

We recall only the essential information about *pure* CCS, which does not involve value passing, and refer to [19] for more details.

Let *Act* be a set of *actions* consisting of *internal* action τ, *base* actions a and *complement* actions $\overline{a}$ with the property $\overline{\overline{a}} = a$. The *process* expressions are defined by the following grammar.

$$P \ ::= \ Nil \mid X \mid \alpha.P \mid P_1 + P_2 \mid P_1|P_2 \mid P\backslash L \mid P[f] \mid rec\ X.P$$

where α ranges over actions, P, P_1, P_2 range over processes, L is a subset of base actions or complement actions, f is a relabelling function from Act to Act with $f(\overline{a}) = \overline{f(a)}$ and $f(\tau) = \tau$. The operational semantics is given via a *labelled transition system* with processes as states and actions as labels. The transition relations are given by the following transition rules in terms of the structure of process expressions.

$$\alpha.P \stackrel{\alpha}{\to} P \qquad \frac{P_1 \stackrel{\alpha}{\to} P}{P_1 + P_2 \stackrel{\alpha}{\to} P} \quad \frac{P_2 \stackrel{\alpha}{\to} P}{P_1 + P_2 \stackrel{\alpha}{\to} P}$$

$$\frac{P_1 \xrightarrow{\alpha} P}{P_1|P_2 \xrightarrow{\alpha} P|P_2} \quad \frac{P_2 \xrightarrow{\alpha} P}{P_1|P_2 \xrightarrow{\alpha} P_1|P} \quad \frac{P_1 \xrightarrow{a} P_1 \quad P_2 \xrightarrow{\overline{a}} P_2}{P_1|P_2 \xrightarrow{\tau} P_1|P_2}$$

$$\frac{P \xrightarrow{a} P'}{P\backslash L \xrightarrow{a} P'\backslash L}(a, \overline{a} \notin L) \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad \frac{P[(rec\ X.P)/X] \xrightarrow{\alpha} P'}{rec\ X.P \xrightarrow{\alpha} P'}$$

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

2.2 μ-calculus

Kozen's (propositional) modal μ-calculus(μK) has expressive power subsuming many modal and temporal logics such as LTL and CTL [4, 5, 9]. We take a negation-free version of the modal μ-calculus and use Winskel's construction of *tagging* fixed points with sets of states [25]. The assertions are constructed from the following grammar:

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

where U is called a *tag* which is a subset of states, X ranges over a set of assertion variables, and K ranges over subsets of labels. We will use $-K$ to abbreviate the universal set of labels except K. The tag-free fixed points $\mu Z.\Phi$ and $\nu Z.\Phi$ are special cases with empty tags.

Let S be the set of states in a labelled transition system. The semantics of assertions $[\![\Phi]\!]_\rho \subseteq S$ is given by induction on the structure of Φ as follows.

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

where the map ρ is an evaluation function which assigns to each assertion variable X a subset of S, and $\rho[\Phi/Z]$ is the evaluation ρ' which agrees with ρ everywhere except on Z when $\rho'(Z) = [\![\Phi]\!]_\rho$. Satisfaction between a state s and an assertion Φ is now defined by: $s \models \Phi$ iff $s \in [\![\Phi]\!]_\rho$ for all ρ.

The inference rules for ν and μ operators can be expressed as follows, where $\vdash_s \Phi$ means that state s satisfies the property Φ.

nu_base

$$\frac{}{\vdash_s \nu Z.U\Phi}(s \in U)$$

nu_unfold

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

mu_base

$$\overline{\not\vdash_s \mu Z.U\Phi}(s \in U)$$

mu_unfold

$$\frac{\vdash_s \Phi[\mu Z.U \cup \{s\}\Phi/Z]}{\vdash_s \mu Z.U\Phi}(s \notin U)$$

To simplify the proof terms, we define two functions *Succ* and *Filter* for [ ] and $\langle\ \rangle$ operators. (*Succ s*) generates a list of successor (label-state) pairs of a state s. (*Filter K slist*) filters the states satisfying the Modality K from *slist* which is the output of *Succ*. Thus we can prove lemma_dia and lemma_box as follows.

lemma_dia

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

lemma_box

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

3 The Formalisation in LEGO

The syntax and semantics of both CCS and μ-calculus can be formalised by means of inductive data types of LEGO. Before describing the formalisation, we will give a brief introduction to LEGO. Further details of LEGO are referred to [16].

3.1 LEGO

LEGO is an interactive proof development system designed and implemented by Randy Pollack in Edinburgh [16]. It implements the type theory UTT [15]. 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 users 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 [23].

There is an `Inductive` command in LEGO [22] 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 datatype `T1 ... Tm` with the constructors `CONS1 ... CONSn` which have corresponding types `L1 ... Ln`.

3.2 CCS

We use lists to represent sets and natural numbers to introduce the base names of actions and variables of processes: Base = nat and Var = nat. Then we define the types of actions and processes as follows.

```
Inductive [ActB : SET] ElimOver Type
Constructors [base : Base->ActB][comp : Base->ActB];

Inductive [Act : SET] ElimOver Type
Constructors [tau:Act][act : ActB->Act];

Inductive [Process : SET] 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 `[rec:(Process->Process)->Process]`. However, LEGO does not allow this sort of expressions since in general they could introduce paradoxes [15]. Instead, we use de Bruijn's indexes [7] to deal with variable binding.

The transition relation can be defined as an inductive relation with each of the constructors in the definition corresponding to one or two rules. For instance, the constructor of rule $Dot : \alpha.P \xrightarrow{\alpha} P$ 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 $ChoL : \frac{P_1 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$ 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 $p1 + p2$).

The complete definition of the transition relation in LEGO syntax is given in Appendix 1.

3.3 μ-calculus

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 *Negmodal*, which correspond to the positive operator [K] and negative operator [-K], respectively. The precise LEGO definition is as follows, where we use de Bruijn's indexes to deal with the binding of ν and μ operators.

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

[Tag= list State];

Inductive [Form:SET] ElimOver Type
Constructors
[VarF:Var->Form]
[OrF:Form->Form->Form]
[AndF:Form->Form->Form]
[Dia:Modality->Form->Form]
[Box:Modality->Form->Form]
[Tnu:Tag->Form->Form]
[Tmu:Tag->Form->Form];
```

3.4 The Semantics and Inference Rules

The semantics is defined as a function which takes a μ-calculus formula and an evaluation mapping as arguments and returns a predicate over states. A function in LEGO can be defined by constructing a proof of the function type. Because LEGO proof scripts are not easy to read, we present the construction of the μ-calculus semantics in Appendix 2 in equational form to make it more understandable.

Using the above formalisation of syntax and semantics, we are able to prove the rules and the lemmas, nu_base, nu_unfold, mu_unfold, lemma_box and lemma_dia, in LEGO. We note that our embedding is *deep embedding*, not *shallow embedding*.

4 The Model Checker, LegoMC

We can verify finite and infinite problems using the above formalisation already. However, there are so many tedious and trivial proof steps; we expect to use model checking to develop parts of the proofs automatically. In the following subsection, we describe the structure of our LegoMC. We then discuss the implementation in subsection 4.2.

4.1 The Structure of LegoMC

Given a file which contains the definition of a finite model and a specification (formula), LegoMC will produce the proof term of LEGO which could be put into LEGO proof processes if the model satisfies the specification. If the model does not satisfy the specification, LegoMC simply produces an error message. The rules are as follows, where $p : (\vdash_s P)$ means p is a proof term of $\vdash_s P$.

OR

$$\frac{p : (\vdash_s P)}{\mathrm{inl}\ p\ q : (\vdash_s P \vee Q)} \qquad \frac{q : (\vdash_s Q)}{\mathrm{inr}\ p\ q : (\vdash_s P \vee Q)}$$

AND

$$\frac{p : (\vdash_s P) \quad q : (\vdash_s Q)}{\mathrm{pair}\ p\ q : (\vdash_s P \wedge Q)}$$

BOX

$$\frac{p_1 : (\vdash_{s_1} \Phi), \ldots, p_n : (\vdash_{s_n} \Phi)}{\mathrm{lemma\_box\ prove\_state\_list} : (\vdash_s [a]\Phi)}(\{s_1, \ldots, s_n\} = \mathrm{Filter\ M\ (Succ\ s)})$$

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

DIA

$$\frac{p' : (\vdash_{s'} P)}{\mathrm{lemma\_dia\ (ExIntro}\ s'\ p') : (\vdash_s \langle K\rangle P)}(s' \in \mathrm{Filter\ M\ (Succ\ s)})$$

NU

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

MU

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

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

4.2 The Implementation

We have implemented LegoMC as a separate program in ML given in Appendix 3. In the following, we explain the implementation of And, Dia and Mu operators, and the others are omitted.

AND
Assume we want to find a proof term p of $\vdash_s P_1 \wedge P_2$. We should find the proof term p_1 of $\vdash_s P_1$ and the proof term p_2 of $\vdash_s P_2$. If we can find both p_1 and p_2, then p is 'pair p_1 p_2'.

DIA
Assume we want to find a proof term p of $\vdash_s \langle K \rangle P$. By lemma_dia, that is $\exists s' \in$ Filter K (Succ s). $\vdash_{s'} P$. Therefore we try to find the proof term p' of $\vdash_{s'} P$ for all the states in *Filter K (Succ s)*. If p' exists, then p is 'lemma_dia (ExIntro s' p')'.

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

5 An Example

One of the applications is to find an abstract finite-state model which is *bisimular* to the original model. Since *bisimulation equivalence* preserves the properties of a model[19], we can then use abstract model instead of the original one. Here bisimilarity is proved in LEGO, and LegoMC is used to prove the abstract finite-state model. We take a simple token ring network from [3] as an example to explain the above approach.

Assume that there are n workstations in a ring network as shown in Fig. 2. 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 on 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:

$$I = \tau.I + pass.IT$$
$$IT = enter.exit.IT + \tau.IT + \overline{pass}.I$$
$$Ring(n) = (IT|I|\ldots|I)\backslash\{pass\} \text{ with } n+1 \; Is(n \geq 0)$$

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

We can find that the abstract model

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

is a bisimular of $Ring(n)$. The Bisimulation is $\{(Ring_{abst}, Ring(n)), (exit.Ring_{abst}, (exit.IT|I|\ldots|I)\backslash\{pass\})\}$. As a result, we can use LegoMC to prove $Ring_{abst}$ against various properties such as mutual exclusion and deadlock freedom.

First, we prove in LEGO the bisimularity,

$$\vdash \text{Bisimular } Ring(n) \; Ring_{abst},$$

and we have the lemma

$$\vdash \text{Bisimular } A \; B \text{ and } \vdash_A \Phi \quad \longrightarrow \quad \vdash_B \Phi.$$

Therefore the proof term of $\vdash_{Ring_{abst}} \Phi$, which is generated by LegoMC, can be integrated into LEGO to complete the whole proof.

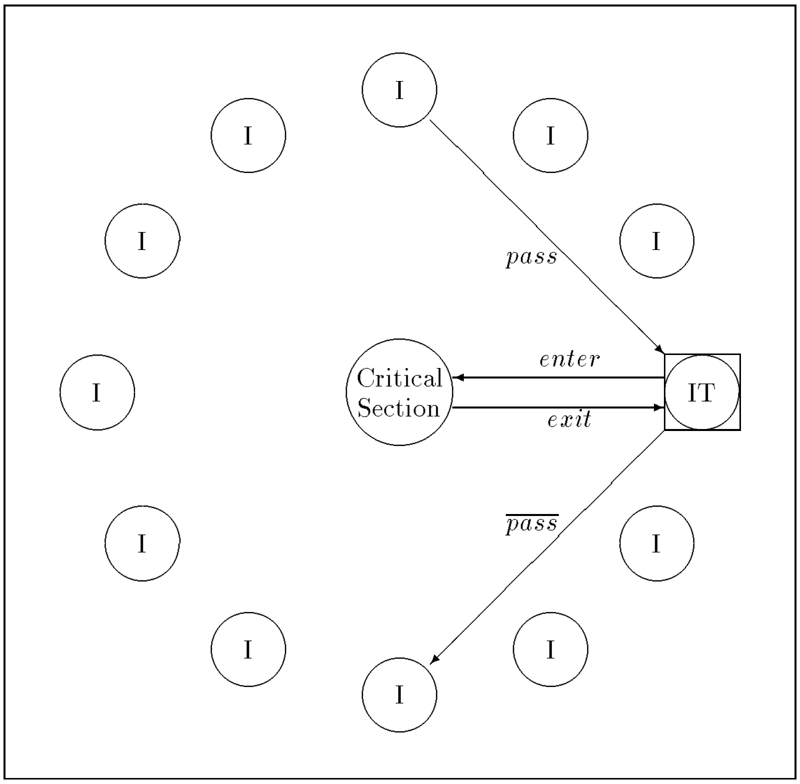

Fig. 2. A token ring network with 12 workstations

6 A Simple Imperative and Concurrent Language

Our system has been extended to a simple imperative and concurrent language and used to verify some finite state examples. In the following subsection, we describe the syntax and semantics of the imperative language. An example is given in subsection 6.2.

6.1 The Syntax and Semantics

We consider a concurrent program as several sequential processes in progress at the same time by interleaved execution sequences of atomic statements. There is an underlying set of global variables that are shared among the processes for inter-process communication and synchronisation. We define labels as primitive statements and boolean expressions, the sequential processes as lists of statements and the programs as lists of processes. The syntax of our language can be described as follows, where BE ranges over boolean expressions, NE ranges over natural number expressions, and *wait* and *signal* are semaphore statements.

1. primitive statements

$$\text{Primitive} ::= x := NE \mid \text{skip} \mid \text{wait\_until } BE \mid \text{wait } s \mid \text{signal } s$$

2. processes

$$\text{Statement} ::= \text{Primitive} \mid \text{if } BE \text{ Process Process} \mid \text{while } BE \text{ Process}$$

$$\text{Process} = \text{Statement list}$$

$$\text{Program} = \text{Process list}$$

3. labels

$$\text{Label} ::= \text{Primitive} \mid \text{BE}$$

We define a state as a pair (P, M), consisting of a program P and a memory M. The memory is a table containing the current values of variables represented 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, the operational semantics of our language can be defined via a labelled transition system as follows.

$$\frac{}{([x := e, p], M) \xrightarrow{x:=e} ([p], M_e^x)}$$

$$\frac{}{([skip, p], M) \xrightarrow{skip} ([p], M)}$$

$$\frac{M(b) = true}{([wait\_until(b), p], M) \xrightarrow{wait\_until(b)} ([p], M)}$$

$$\frac{M(s) > 0}{([wait(s), p], M) \xrightarrow{wait(s)} ([p], M_{s-1}^s)}$$

$$\frac{}{([signal(s), p], M) \xrightarrow{signal(s)} ([p], M_{s+1}^s)}$$

$$\frac{M(b) = true}{([\text{if b then } p_1 \text{ else } p_2, p], M) \xrightarrow{b} ([p_1, p], M)}$$

$$\frac{M(b) = false}{([\text{if b then } p_1 \text{ else } p_2, p], M) \xrightarrow{\neg b} ([p_2, p], M)}$$

$$\frac{M(b) = true}{([\text{while b do p}, p'], M) \xrightarrow{b} ([p, \text{while b do p}, p'], M)}$$

$$\frac{M(b) = false}{([\text{while b do p}, p'], M) \xrightarrow{\neg b} ([p'], M)}$$

$$\frac{(p_1, M) \xrightarrow{l} (p, M')}{(p_1 || p_2, M) \xrightarrow{l} (p || p_2, M')}$$

$$\frac{(p_2, M) \xrightarrow{l} (p, M')}{(p_1 || p_2, M) \xrightarrow{l} (p_1 || p, M')}$$

6.2 An Example

As an example, we consider a semaphore solution of the mutual exclusion problem for two processes. The individual sequential process is as follows.

$$Critical = skip$$

$$\text{P} = [\text{while True [wait } S\text{, Critical, signal } S]]$$

The program is $Pro = [P, P]$. The initial value of semaphore S is 1, $init = [(S, 1)]$. The mutual exclusion property defined in μ-calculus is "For all the states after the initial state, if the program can perform *wait* S to enter the critical section, the program cannot perform *wait* S again unless it performs *signal* S first." The μ-calculus formula is as follow.

$$\text{ME} = \Box([\text{wait } S]\nu Z.(\text{inable [wait } S]) \wedge [\text{-(signal } S)]Z)$$

where $\Box\Phi = \nu Z.[-]Z \wedge \Phi$ and inable $X = [X]\mu Z.Z$.

We can prove the following two theorems by LegoMC.

$$\vdash_{(init,Pro)} ME$$

$$\vdash_{(init,Pro)} deadlockfree$$

where deadlockfree $= \Box(\langle - \rangle \nu Z.Z)$.

The second author of this paper used the direct LEGO formalisation to prove these properties, it is much harder.

7 Conclusions and Future Work

Theorem proving based on type theory produces not only a 'TRUE' or 'FALSE' answer to a problem but also an explicit proof term. We can therefore integrate various *proof generators*, interactive or automatic ones, if they can produce proof terms. No matter how complicated those proof generators are, the correctness of proofs is assured by the simple proof checking algorithm. In this paper, we have showed how to verify concurrent programs in LEGO by combining interactive theorem proving with model checking. This approach can be generalised to other temporal logic model checker such as SMV [18].

Beside the proof terms, another difference of LegoMC with model checkers in HOL and PVS is the domain languages. Rather than automata, we use CCS and the imperative language which are more natural to express a software system. Another difference is that we use deep embedding instead of shallow embedding so that we can prove the correctness of our model checking rules.

In this paper, the proof of infinite part is mostly based on the semantics. Although we can use LegoMC to simplify the proof work significantly, part of the proof which cannot use LegoMC to solve can still be difficult. Bradfield and Stirling developed a sound and complete tableau proof system of local model checking for infinite state spaces [2]. It is expected that we can formalise their proof system in LEGO to help the verification of infinite problems.

Since LegoMC generates the proof terms of LEGO syntax, it depends on the formalisation of concurrent languages (CCS) in LEGO. Once the formalisation is changed, the model checker has to be changed as well. However, we design the interface to μ-calculus as a *succ* function which takes a state and returns a list of successor states so that we do not have to change the μ-calculus part if we change the concurrent languages. In this way, we have implemented a simple imperative language using the same formalisation of μ-calculus. In the future, it is expected that the model checker can accept the inductive definition of LEGO directly so that the model checker can become generic.

At the moment, the size of generated proof terms is quite big so that LegoMC is not very efficient and also needs a lot of memory. To enhance the efficiency, we need to further develop some pre-proved lemmas and use abbreviations; this will be done in the near future.

References

1. 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.
2. Julian Bradfield and Colin Stirling. Local model checking for inifinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.
3. Glenn Bruns. Algebraic Abstraction with Process Preorders. in preparation, 1995.
4. 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.
5. 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.
6. R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Pretice-Hall, 1986.
7. 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.
8. G. Dowek et al. *The Coq Proof Assistent: User's Guide (version 5.6).* INRIA-Rocquencourt and CNRS-ENS Lyon, 1991.
9. 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.
10. 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.
11. 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.

12. 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.
13. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, Princeton, New Jersey, 1994.
14. 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.
15. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* International Series of Monographs on Computer Science. Oxford University Press, 1994.
16. 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.
17. L. Magnusson. The new implementation of ALF. In *Informal Proceedings of Workshop on Logical Frameworks*, Bastad, 1992.
18. Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.
19. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
20. 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.
21. 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.
22. Randy Pollack. *Incremental Changes in LEGO: 1994*, May 1994. Available by ftp with LEGO distribution.
23. 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.
24. 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.
25. 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 ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 761–772. Springer-Verlag, 1989.
26. 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.

Appendix 1: The Transition Relation of μ-calculus

```
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_act a R)(member_act 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')];
```

Appendix 2: The Semantics of μ-calculus

```
Sem : Form -> map_Form -> State.Pred

Sem (VarF X) V = V X
Sem (OrF P Q) V = Or (Sem P V) (Sem Q V)
Sem (AndF P Q) V = And (Sem P V) (Sem Q V)
Sem (Dia (Modal K) P) V = [s:State]Ex[l:Label]Ex[s':State]
and3 (Member l K)(Trans l s s')(Sem P V s')
Sem (Dia (Negmodal K) P) V = [s:State]Ex[l:Label]Ex[s':State]
and3 (not(Member l K))(Trans l s s')(Sem P V s')
Sem (Box (Modal K) P) V = [s:State]All[l:Label]All[s':State]
((Member l K).and (Trans l s s'))->(Sem P V s')
Sem (Box (Negmodal K) P) V = [s:State]All[l:Label]All[s':State]
((not(Member l K)).and (Trans l s s'))->(Sem P V s')
Sem (Tnu T P) V = [s:State]Ex[Q:State.Pred]
(Q.Subset ((Sem P (change V Q one)).Union T).and (Q s)
Sem (Tmu T P) V = [s:State]All[Q:State.Pred]
(((Sem P (change V Q one)).Minus T).Subset Q)->(Q s)
```

Appendix 3: The Model Checking Algorithm

```
fun check s Φ =
    case Φ of
    Var V      ⟶ error
    Φ1 ∨ Φ2    ⟶ return "inl "+(check s Φ1) or "inr "+(check s Φ2)
    Φ1 ∧ Φ2    ⟶ return "pair "+(check s Φ1)+(check s Φ2)
    ⟨K⟩Φ'      ⟶ if exists a state s'∈ xs=Filter K (succ s) such that check s' Φ'
                   is provable
                   then return "lemma_dia (ExIntro "+state2str(s')+")
                   ([s':State] and (Member s' (Filter "+(modality2str K)
                   +"(Succ "+state2str(s)+")))(sem_Form "+(form2str Φ')
                   +" V s'))(pair "+(prove_member s' xs)+(check s' Φ')
                   else error
    [K]Φ'      ⟶ return "lemma_box|"+(modality2str K)+"|?|"+(state2str s)
                   +" ([s':State]"+(checklist (Filter K (Succ s)) Φ')+"))"
    νX.UΦ'     ⟶ if s ∈ U then return "nu_base "+(prove_member s U)
                   else return "nu_unfold|?|"+(form2str Φ')
                   +(check s Φ'[νX.(U ∪ s)Φ'/X])
    μX.UΦ'     ⟶ if s ∈ U then error
                   else return "mu_unfold|?|"+(form2str Φ')
                   +(check s Φ'[μX.(U ∪ s)Φ'/X])
fun checklist xs P =
    case xs of
    [ ]        ⟶ return "([h: Member s' (nil State)] Not_Member_nil h
                   (sem_Form "+(form2str P)+" V s'))"
    y::ys      ⟶ return "([h:Member s' (cons ("+(state2str y)+") ("
                   +(liststate2str ys)+"))]Mem_ind1 h "+(checklist ys P))
                   +"([h:Eq "+(state2str y)+" s'] Eq_subst h ([z:State]sem_Form"
                   +(form2str P)+" V z) "+(check y P)+"))"
fun prove_member s U =
    case U of
    [ ]        ⟶ error
    x::xs      ⟶ if s=x then "member_head|?|?|?" else "member_tail "
                   +prove_member s xs
```

where *Succ* is a function with type *state* → *list (label*state)* which takes a state and returns a list of successor states with the corresponding labels, *Filter* is a function which takes a list of *(label*state)* pairs and returns the list of states with corresponding labels which satisfy the modality K. Several ****2str* functions are used to convert a type value to a corresponding string in LEGO's syntax.

This article was processed using the LaTeX macro package with LLNCS style