Symmetric Temporal Theorem Proving

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Master in Philosophy

Amir Niknafs Kermani
Department of Computer Science
The Univeristy Of Liverpool

May 30, 2017

Contents

Acknowledgement

I would like to express my deepest gratitudes to my family and supervisors, Dr. Boris Konev and Prof. Michael Fisher. I could not think of anyone that could be more troublesome as I was. Despite their full effort and encouragement I feel I have let them down greatly.

Chapter 1

Introduction

1.1 Introduction

A *Symmetric system* is a system comprising of some identical processes. These systems contain some identical tasks that are the same for all identical processes. A symmetric system often satisfies a set of properties; for example, *safety*, meaning "something bad will not happen", *liveness*, meaning "some particular situation must eventually happen" and *fairness*, meaning "something should happen infinitely often". There have been numerous attempts to exploit symmetry in systems [15, 34, 30, 16]. However, they mostly involve *model checkers*. In this thesis we try to tackle symmetry using *theorem proving* instead.

Automated theorem proving is a collection of techniques to deal with mathematical theorems using a computer program. This technique has been applied into many different logics including *temporal logic*. *Temporal proof* is a technique used to ascertain whether a temporal logic formula is valid or not. In this thesis, we consider automated temporal theorem proving, focusing on an extension of the resolution procedure developed by Robinson [32], namely the *clausal resolution* method for propositional temporal logic (PTL) [19]. As the complexity of satisfiability for PTL is PSPACE complete, its use becomes more difficult as the considered PTL formulae become larger. Our approach is to infer resolution steps in larger problems by considering, and extending, resolution steps in corresponding, but smaller, versions. Based on the steps that have been carried out for smaller formulae, we make a 'guess' at the steps needed for larger formulae of a similar pattern. Clearly, this will only work if the formulae increase in a regular way and if we have a quick way to validate the 'guesses'.

On the other hand, when considering *symmetric systems*, we are normnally interested to prove the system for arbitary number of processes. Thus, another approach to specify these systems is to use a fragment of *first order temporal logic* known as *monodic FOTL* [22] instead, in which we can specify these system for arbitary number of processes. One of its sub-fragment known as *monadic FOTL* can even be decideable. Unfortunately, even this sub-fragment loses its decidability once equality is added[10]. As an inspiration from FOTL, in this thesis, in chapter *Template* we try to explain that if we can prove a *symmetric system* for i processes, we can greatly reduce the steps required to prove the same *symmetric system* with $i + 1$ processes. The detail of this has been published in *TIME International Symposium on Temporal Representation and Reasoning* [28]

The stimulus for this work comes from temporal *model checking* [7], where the main problem is the state-space explosion that occurs as problems increase in size. In that field, symmetric techniques have been used to tackle larger problems by considering smaller instances. The *SPIN* model checker [23] method has exploited symmetry to overcome the state-space problems, albeit to a limited extent, and only for certain classes of system. Thus, our aim in this paper is to introduce a symmetric way to increase the efficiency of the resolution method as the size of the problem increases.

After the introductory and preliminary sections, we define symmetric systems. Then we introduce a new notion, the *template*, which can be used to *guess* a *step resolution* of a large symmetric system from a *step resolution* of the same system with a smaller number of processes, and finally checking its correctness. By doing this, we can bypass the step resolution for the large system, thus improving the efficiency. We then apply this technique to the deductive temporal verification of a *cache coherence protocol* [11].

For small numbers of identical processes, propositional temporal provers can handle their deductive verification. However, if the number of processes increases, or if the formula is complex, even temporal resolution provers such as TeMP [24] or TSPASS [26] may fail to handle such problems. By applying our technique to such symmetric problems, we can perform automated temporal proofs for larger numbers of processes by extrapolating from simpler examples.

Thus, the ultimate aim of this thesis is to provide a technique that allows deductive temporal methods to be applied to larger problems. Just as model-checking can be applied to problems comprising large numbers of identical processes [5, 33, 6], this thesis shows how propositional temporal proving can be productively used in a similar way.

Chapter 2

Preliminaries

2.1 Temporal Logic

Temporal logic is an extension of classical logic, specifically adding operators dealing with time. In addition to operators of classical logic, temporal logic often contains extra operators dealing with time. For example, '$\square$', meaning always, '$\bigcirc$', meaning in the next moment in time and '$\Diamond$', meaning sometime in the future. These additional operators allow us to construct formalisms such as:

$$\square(active\_process \Rightarrow \Diamond task\_is\_done)$$

Meaning : "<u>whenever</u> there is an active process, then <u>at some point</u> in the future, our task is achieved."

Temporal logic was first introduced by *Arthur Prior* and later was further developed by *Hans Kamp* and *Amir Pnueli.*

2.1.1 Propositional Temporal Logic

Propositional Temporal Logic (PTL)[29] is a logic that is *discrete*, *linear* and *propositional* Additionally to classical propositional operator, PTL contains a set of *temporal operators* that deal with time.

- Discrete: the underlying model of time being isomorphic to the natural numbers, with a distinguished initial point and an infinite future.
- Linear: in each moment in time, there is at most one successor.
- propositional : with no explicit first-order quantification

Basic syntax and semantics

The set of well-formed PTL formulae is the smallest set containing all propositions (and their negations) and the boolean constants $\top$ and $\bot$, such that whenever φ and ψ are PTL formulae so are the following:

$\neg\phi$	At this moment in time ϕ is not true
$\phi \Rightarrow \psi$	At this moment in time, ψ is true if ϕ is true
$\phi \Leftrightarrow \psi$	At this moment in time, ϕ is true if and only if ψ is true
$\phi \vee \psi$	At this moment in time, ϕ or ψ (or both) are true
$\phi \wedge \psi$	At this moment in time, both ϕ and ψ are true
$\bigcirc\phi$	At the next moment in time, ϕ will be true
$\Box\phi$	ϕ is always true (at any moment in time)
$\Diamond\phi$	ϕ will eventually become true
$\phi U \psi$	ϕ will be true *until* ψ becomes true
$\phi W \psi$	ϕ will be true *unless* ψ is true (ψ doesn't have to become true at all)

In the above table $\bigcirc$, $\Diamond$, $\Box$ are unary temporal connectors while U and W are binary temporal connectors. In addition, "(" and ")" are used to avoid ambiguity. In later we avoid the use of U and W connectors as one can transform them into a combination of unary connectors.

For the semantics of PTL, let us recall that PTL has a linear and discrete basis that is isomorphic to $\mathbb{N}$, thus the model structure is the following:

$$\mathcal{M} = \langle \mathbb{N}, \pi \rangle$$

where $\pi : \mathbb{N} \mapsto \mathrm{P}$ maps each *moment in time* to the set of propositions (P) that are *true* at that moment. Thus, for a structure, $\mathcal{M}$, temporal index (the moment in time), i and a formula ϕ we have

$$\langle \mathcal{M}, i \rangle \vDash \phi$$

which is true if, and only if, ϕ is satisfied at the temporal index i within the model $\mathcal{M}$. For full semantics, let's look at the following:

$\langle \mathcal{M}, i \rangle \models \top$	
$\langle \mathcal{M}, i \rangle \models \phi$	iff ϕ is a proposition and $\phi \in \pi(i)$
$\langle \mathcal{M}, i \rangle \models \neg\phi$	iff p is **not** the case that $\langle \mathcal{M}, i \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \phi \wedge \psi$	iff $\langle \mathcal{M}, i \rangle \models \phi$ and $\langle \mathcal{M}, i \rangle \models \psi$
$\langle \mathcal{M}, i \rangle \models \phi \vee \psi$	iff $\langle \mathcal{M}, i \rangle \models \phi$ or $\langle \mathcal{M}, i \rangle \models \psi$
$\langle \mathcal{M}, i \rangle \models \phi \Rightarrow \psi$	iff if $\langle \mathcal{M}, i \rangle \models \phi$ then$\langle \mathcal{M}, i \rangle \models \psi$
$\langle \mathcal{M}, i \rangle \models \phi \Leftrightarrow \psi$	iff if $\langle \mathcal{M}, i \rangle \models \phi$ then$\langle \mathcal{M}, i \rangle \models \psi$ and if $\langle \mathcal{M}, i \rangle \models \psi$ then$\langle \mathcal{M}, i \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \phi \wedge \psi$	iff $\langle \mathcal{M}, i \rangle \models \phi$ and $\langle \mathcal{M}, i \rangle \models \psi$
$\langle \mathcal{M}, i \rangle \models \bigcirc\phi$	iff $\langle \mathcal{M}, i+1 \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \Box\phi$	iff for all j, if $(j \geq i)$ and $\langle \mathcal{M}, j \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \Diamond\phi$	iff there exists j such that$(j \geq i)$ then $\langle \mathcal{M}, j \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \phi U \psi$	iff there exists $j \geq i$ such that $\langle \mathcal{M}, j \rangle \models \psi$ and for all $k : i \leq k < j$, $\langle \mathcal{M}, k \rangle \models \phi$
$\langle \mathcal{M}, i \rangle \models \phi W \psi$	iff either$\langle \mathcal{M}, i \rangle \models \phi U \psi$ or $\langle \mathcal{M}, i \rangle \models \Box\phi$

2.1.2 Satisfiability, Unsatisfiability and Validity

Given a formula ϕ, we say ϕ is *satisfiable* if and only if there exists a *model* $\mathcal{M}$, which ϕ holds. ϕ is *valid* if it holds under all interpretations e.g.

$$\phi \Leftrightarrow true$$

Finally ϕ is *unsatisfiable* if there is no interpretation that ϕ is true.

2.1.3 System Properties

The key motivation for temporal logic has been to specify the dynamic properties for reactive systems. Some of the interesting properties are *safety*, meaning "something bad will not happen", *liveness*, meaning "some particular situation must eventually happen" and *fairness*, meaning "something should happen infinitely often".

Using temporal logic we can specify such properties as follows:

- *safety*: $\Box\neg p$
- *liveness*: $\Diamond p$
- *fairness*: $\Box\Diamond p$

2.1.4 First-order temporal logic

First-order temporal logic (FOTL) over the natural numbers is incomplete. Thus there is no finitary inference system[1] which is sound and complete. Or it could be said that the set of valid formulae of the logic is not recursively enumerable. However, monodic fragments of first-order temporal logic [22], where every subformula can have at most one free variable. In addition, a monodic fragment has a finite inference system. Some of the sub-fragments of monodic FOTL can even be decidable. An interesting sub-fragment of *monodic* FOTL is *monodic monadic* where each predicate has, at most, an arity of one. However decidability and completeness only holds for logic without equality. Unfortunately, even this sub-fragment loses its decidability once equality is added [10], thus it is not recursively enumerable.

Syntax of FOTL

Well-formed formulae of FOTL (WFF_f) are generated from the symbols of PTL together with the following:

- A set, $\mathcal{L}_p$, of predicate symbols represented by strings of lower-case alphabetic characters.

[1]A finitary inference system is a system that takes a finite input to produce an output

- A set, $\mathcal{L}_v$, of variable symbols, x, y, z, etc.
- A set, $\mathcal{L}_c$, of constant symbols, a, b, c, etc.
- A set, $\mathcal{L}_f$, of function symbols, f, g, h, etc.
- Quantifiers, $\forall$ and $\exists$

The set of terms, $\mathcal{L}_t$, is defined as follows.

1. Both $\mathcal{L}_v$ and $\mathcal{L}_c$ are subset of $\mathcal{L}_t$.
2. if $t_1, \ldots, t_n$ are in $\mathcal{L}_t$, and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is in $\mathcal{L}_t$.

The set of well-formed formulae of FOTL(WFF_f) is defined as follows.

1. If $t_1, \ldots, t_n$ are in $\mathcal{L}_t$ and P is a predicate symbol of arity n, then $P(t_1, \ldots, t_n)$ is in WFF_f.
2. if A and B are in WFF_f, then the following are in WFF_f

$\neg\phi$	At this moment in time ϕ is not true
$\phi \Rightarrow \psi$	At this moment in time, ψ is true if ϕ is true
$\phi \Leftrightarrow \psi$	At this moment in time, ϕ is true if and only if ψ is true
$\phi \vee \psi$	At this moment in time, ϕ or ψ (or both) are true
$\phi \wedge \psi$	At this moment in time, both ϕ and ψ are true
$\bigcirc\phi$	At the next moment in time, ϕ will be true
$\Box\phi$	ϕ is always true (at any moment in time)
$\Diamond\phi$	ϕ will eventually become true
$\phi U \psi$	ϕ will be true *until* ψ becomes true
$\phi W \psi$	ϕ will be true *unless* ψ is true (ψ doesn't have to become true at all)

3. If A is in WFF_f and v is in $\mathcal{L}_t$, then $\exists v.A$ and $\forall v.A$ are both in WFF_f.

2.2 Automated theorem proving

In general, automated theorem proving is a collection of techniques to deal with mathematical theorems using a computer program. There are varieties of theorem-proving techniques such as *resolution* and *tableaux* and etc. Depending on the problem, one technique may prove to become more useful than another. In this thesis we mainly focus on the proof procedure using *resolution*. Resolution for classical logic was first proposed by Robinson [32]. It is claimed to be "machine oriented" as it was deemed particularly suitable for proof to be performed by computers. Using this approach, the validity of a logical formula, φ, is checked by first negating it and translating $\neg\varphi$ into Conjunctive Normal Form (CNF). Then by using an inference rule, repeatedly checking for unsatisfiability. If a *contradiction* is derived, then $\neg\varphi$ is unsatisfiable, therefore the original formula is *valid*.

2.2.1 Robinson's Resolution method

Robinson resolution method is a method of theorem proving that proceeds by constructing refutation proofs. It is called proofs by contradiction. This method uses one rule of inference that may have to be applied many times. This approach is particularly suitable for proofs to be done by computers. In this approach, the validity of a logical formula, φ, is checked by first negating it and translating $\neg\varphi$ into a particular form, Conjunctive Normal Form (CNF) before applying resolution rules. A CNF form can be represented as:

$$\varphi = C_1 \wedge C_2 \wedge \dot{C}_n$$

where each C_i, known as a *clause*, is a disjunction of literals, and each literal is either an atomic formula[2] or its negation. Typically, this CNF formula is represented as a set of clauses $C = \{C_1, C_2, \ldots, C_n\}$.

Having a formula in CNF form, we can apply the resolution rule of inference:

$$\frac{\begin{array}{c}(R_1 \vee p)\\(R_2 \vee \neg p)\end{array}}{(R_1 \vee R_2)}$$

R_1 and R_2 are disjunctions of literals, and p is a proposition. Since one of p and $\neg p$ must be false, then one of R_1 or R_2 must be true. Then we can produce a new clause $(R_1 \vee R_2)$ that can be added to the set of CNF clauses. We continuously apply the inference rule until an empty clause (*false*) is generated or no new clause can be generated. If *false* is derived, $\neg\varphi$ is unsatisfiable, thus the original formula φ is *valid*. This procedure is guaranteed to terminate in propositional logic. The resolution principle also applies to first-order logic formulas in *skolemized form*. As justification of Robinson's original statement that resolution provides a "machine oriented" approach, many theorem provers based on resolution have been developed for computers; for example, *Otter*[27] and *Vampire*[31] are successful resolution based provers.

2.2.2 Temporal Automated Proving

There have been attempts to use Robinson's resolution technique for Temporal Logic. These fall into two main classes: *non-clausal* and *clausal*. The

[2]Atomic formula is a formula that contains no logical connectives

non-clausal method described in [1] and extended to First-Order Temporal Logic in [2] requires a large number of resolution rules. Although it is unclear what the maximum number of rules is, having many resolution rules can make implementation problematic, as each rule must be implemented separately, and the decision of which rule should follow is often difficult. On the other hand, a *clausal* resolution technique was originally introduced in [19] by Michael Fisher. This technique requires the PTL formula to be in a normal form, namely *Separated Normal Form* (SNF) [18]. This was later refined into *Divided Separated Normal Form*, which has been found to be particularity useful in developing clausal resolution for FOTL, as it preserve satisfiability of the original formula with only linear growth in size [25].

Separated Normal Form (SNF)

PTL formulae can become complex and difficult to understand. Thus it is useful to replace one complex formula by several simpler formulae which have behaviour equivalent to the original. This transformation process preserves satisfiability and ensures that any model of the transformed formula is a model of the original one.

Therefore *Separated Normal Form* [19] was introduced. In SNF formulae are represented as:

$$\Box \bigwedge_{i=1}^{n} R_i$$

where each R_i, known as a *clause*, must be of one of the following forms

$$start \Rightarrow \bigvee_{b=1}^{r} l_b \quad \textit{(an initial rule)}$$

$$\bigwedge_{a=1}^{g} k_a \Rightarrow \bigcirc \bigvee_{b=1}^{r} l_b \quad \textit{(a step rule)}$$

$$\bigwedge_{a=1}^{g} k_a \Rightarrow \Diamond l \quad \textit{(a sometime rule)}$$

where each k_b, l_b or l is simply a literal.

Divided Separated Normal Form (DSNF)

DSNF is an equivalent alternative to SNF. A temporal problem is comprised of four sets of formulae $\mathcal{I}$,$\mathcal{U}$, $\mathcal{E}$ and $\mathcal{S}$ where

$\mathcal{I}$: represents the initial part of the problem and contains non-temporal formulae that should be satisfied at the first moment in time. Each element of $\mathcal{I}$ would be of the form:

$$\bigvee_a l_a$$

$\mathcal{U}$: represents the universal part of the problem and contains non-temporal formulae that are universally true. Each element of $\mathcal{U}$ would be of the form:

$$\Box(\bigvee_b l_b)$$

$\mathcal{S}$: represents the step part of the problem containing step clauses. Each element of $\mathcal{S}$ would be of the form:

$$\Box(\bigwedge_c l_c \Rightarrow \bigcirc \bigvee_d l_d)$$

$\mathcal{E}$: represents the sometime part, containing eventualities that must be satisfied infinitely often Each element of $\mathcal{E}$ would be of the form:

$$\Box\Diamond l$$

The full temporal problem is characterised by the formula

$$\bigwedge \mathcal{I} \wedge \bigwedge \mathcal{U} \wedge \bigwedge \mathcal{S} \wedge \bigwedge \mathcal{E}$$

where "$\bigwedge$" means that all elements of subsequent set are conjoined.
As mentioned above, DSNF is particularly useful in developing clausal resolution for FOTL.

2.2.3 Clausal Temporal Resolution

An overview of the Clausal Temporal Resolution method is as follows:

1. Transform formula A into *Separate Normal Form* (SNF) [17], giving a set of clauses A_s.
2. Perform *step resolution* [17] on clauses from A_s until either:
   (a) A contradiction is derived, in which case A is unsatisfiable; or
   (b) No new resolvent are generated, in which case we continue to step(3).
3. Select an eventuality from within A_s, and perform *temporal resolution* [17] with respect to this - if any new formulae are generated, go back to step (2).

4. If all eventualities have been resolved, then A is satisfiable; otherwise, go back to step (3).

The *soundness*, *completeness* and *termination* of this method was proved in [19]. The basis of the above method is the following two-resolutions rule:

- Step Resolution:

$$\begin{array}{llcll} (\textit{initial}) & (R_1 \lor \text{p}\ ) & (\textit{step}) & \Box(L_1 & \Rightarrow \bigcirc(R_1 \lor p)) \\ & (R_2 \lor \neg\text{p}) & & \Box(L_2 & \Rightarrow \bigcirc(R_2 \lor \neg p)) \\ \hline & (R_1 \lor R_2) & & \Box(L_1 \land L_2) & \Rightarrow \bigcirc(R_1 \lor R_2)) \end{array}$$

- Temporal Resolution:

$$\begin{array}{l} \Box\Diamond l \\ \Box(L \Rightarrow \bigcirc\Box\neg l) \\ \hline \Box\neg L \end{array}$$

While step resolution closely resembles the classical resolution rule, the key inference rule in clausal temporal resolution, namely the temporal resolution itself, requires further steps. The translation to SNF restricts the PTL clauses to be of a certain form.

The formula $\Box(L \Rightarrow \bigcirc\Box\neg l)$ is called the *loop formula* and the process must reconstruct this formula. The process of reconstructing the *loop formula* is known as *loop search*. The conclusion of *temporal resolution* is a clause that is always true at any moment in time. Such a clause is called *universal clause*. However, $\Box(L \Rightarrow \bigcirc\Box\neg l)$ can only be constructed by a combination of universal and step clauses. L is a disjunction of conjunctions of literals (i.e. in DNF); therefore, its negation is a conjunction of clauses, which is added to the set of universal clauses. *Loop search* was originally developed by Clare Dixon in [12]. It was further developed in [14], so classical automated reasoning can be used for this purpose.

2.2.4 TSPASS

TSPASS[26] is an implementation of clausal temporal resolution based on the first-order theorem prover SPASS [35]. It implements temporal resolution for *monodic* first-order temporal logic. TSPASS implements a number of simplifications and enhancements over the clausal temporal resolution approach as follows:

- It uses the *simplified* clausal temporal resolution calculus [9].
- It is based on ordered resolution [3].
- It uses TSPASS underneath.

TSPASS is currently available and can be found at:

`http://lat.inf.tu-dresden.de/~michel/software/tspass/`

2.2.5 TeMP

TeMP [24] is another implementation of clausal temporal resolution based for *monodic* first-order temporal logic. It is based on the first-order theorem prover *Vampire* [31]

TeMP is currently available and can be found at:

`http://www.csc.liv.ac.uk/~konev/software/TeMP/`

TSPASS and TeMP are usually competitive; sometimes one can perform better than the other on different scenarios and examples, but a combination of both can also be useful.

2.3 Symmetric system

A *symmetric* system comprises of some identical processes. These systems contain some identical tasks that are the same for all identical processes. Preferably these systems should be specified for arbitrary number of processes.

The aim is to first specify these systems and then check some of the properties they might have. One way of specifying these systems is to use a *temporal logic* specification; for instance, we can use PTL to specify a system for i processes, then use a PTL prover to verify and check certain properties.

This procedure works as follows; say ϕ_i is a specification of a symmetric system S_i with i processes. We also have a property ψ_i of system S_i. We want to check whether $\vdash \phi_i \Rightarrow \psi_i$. This can be achieved by *proving* that one temporal formula implies another. One way of doing that is by using clausal resolution [19] for PTL, in which we check for *unsatisfiablity* of $\neg(\phi_i \Rightarrow \psi_i)$. This technique is based on a proof procedure for classical logics by Robinson[32]. However, as previously mentioned, there might be an unknown number of processes in such systems, therefore, proving $\neg(\phi_i \Rightarrow \psi_i)$ can only provide proof for i processes and cannot determine the proof for, say, $i + 1$ processes.

Another approach might be to specify these systems is to use *monadic FOTL* instead, in which we can specify these system for arbitrary numbers of processes. But a problem arises when we come to a situation where equality becomes an issue. The full details of such issues has been explained in [20] and [4]. It becomes difficult to specify such system in *monadic FOTL*. Even though these papers suggests some techniques to deal with the specification, none of these techniques are complete. We will instead show a way of dealing with specification and proof of a *strict* set of symmetric systems for an arbitrary number of processes.

2.4 Parameterised System

A *parameterised* system is a distributed system comprising of N identical processes. Every process has inner states known as *'local state'*; e.g. initial and final states. Processes in a parameterised system can interact with each other. This interaction results in some general states known as *'global states'* within a system. *Cache Coherence Protocol* [11, 20] is an example of such a system. Parameterised systems are considered to be symmetric as systems contain of identical processes.

2.4.1 Cache Coherence Protocol

In a shared-memory multiprocessor system *local caches* are used to reduce memory access latency and network traffic. Although local caches improve system performance they introduce the *cache coherence problem* as multiple cached copies of the same block of memory must be consistent at any time during a run of the system. A *cache coherence protocol* ensures the *data consistency* of the system: the value returned by a read must be always the last value written to that location. *cache coherence protocols* are an example of Parameterised systems.

Automatically checking the safety properties independently from the number of processors has been tackled using general purpose, infinite-state symbolic model checking [11]. Alternatively, one may verify cache coherence protocols for all possible dimensions by first translating into monodic first-order temporal specification and then applying temporal theorem proving. This has been done in [20]. The following correctness conditions are the most important for cache coherence protocols:

1. **non co-occurrence of state:** some specific local states q1 and q2 cannot appear in the same global state.

2. **at most one:** a local state q can appear at most once - i.e. be the state of no more than one processor - in any global state.

Even though *cache coherence protocols* can be represented as monodic first-order formulae, and then be verified using a first-order prover, it is still hard to prove some properties. The reason for this difficulty is the general difficulty of first-order formula.

Another approach to verify these system is to first translate them into Propositional Temporal Logic for some number of processes. The trade off for this approach compared with monodic first-order translation is simple;

we lose expressiveness but on the other hand gain advantage over decidability of the formula.

MSI Protocol

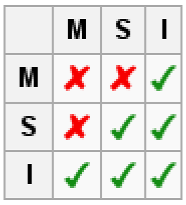
The MSI protocol is a basic cache coherence protocol that is used in multi-processor systems.

MSI has three components. Every process can be in exactly one of these components:

- **Modified:** The block has been modified in the cache. The data in the cache is then inconsistent with the backing store (e.g. memory). A cache with a block in the "M" state has the responsibility to write the block to the backing store when it is evicted.
- **Shared:** This block is unmodified and exists in at least one cache. The cache can evict the data without writing it to the backing store.
- **Invalid:** This block is invalid, and must be fetched from memory or another cache if the block is to be stored in this.

For any given pair of caches, the permitted states of a given cache line are as follows:

Figure 2.1: Relation between two processes in MSI Protocol



	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

As the figure 2.1 shows, two processes cannot be in the *Modified* state at the same time. Furthermore, if one process is in the *Shared* state, no other process can be in the *Modified* state.

Figure 2.2 demonstrates this concept in more detail, how each process in MSI can read/write either locally or globally, meaning; reading from the local cache (LR), writing to the local cache (LW), writing to memory (BW) and reading from memory (BR).
If the process is in *invalid* state then

- LR : process has to put a read request on the bus and then move to *modified* state.

- LW : process has to put a write request on the bus and then move to *shared* state.
- BR : process doesn't have to do anything (stays in the same state).
- BW : process doesn't have to do anything (stays in the same state).

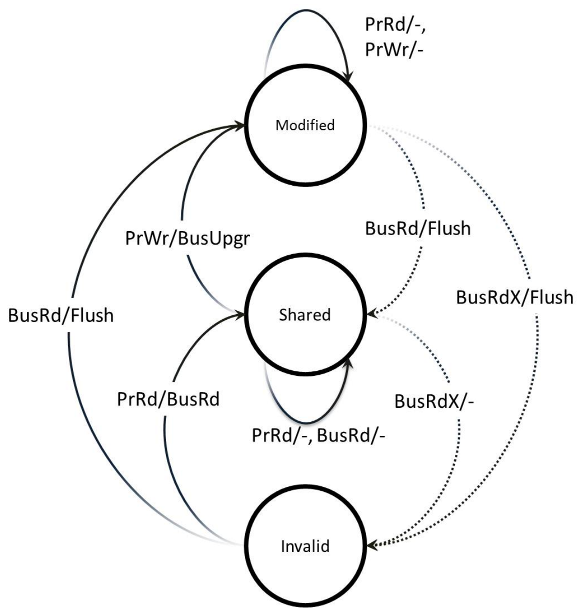
If the process is in *shared* state then :

- LR : process doesn't have to do anything (stays in the same state).
- LW : process doesn't have to do anything (stays in the same state).
- BR : process has to write back to memory then move to *shared* state
- BW : process has to write back to memory then move to *invalid* state.

If the process is in *modified* state then:

- LR : process doesn't have to do anything (stays in the same state).
- LW : put invalidation request on the bus to ask everyone to invalidate their copy then move to *modified* state.
- BR : process has to write to memory then move to *shared* state.
- BW : move to *invalid* .

Figure 2.2: Relation between two processes in MSI Protocol



Chapter 3

Template

In this chapter we describe a formal representation of formulas in a symmetric system. This representation shows how we can capture such similarties and eventualy use them in order to improve our proof.

3.1 Definition

Template is a mean to sort and group disjunctions of logic formula from a symmetric system. Typically it is a way of representing a disjunction or a conjunction of propositions. More formally we can define a *template* as :

Definition 1 *A* template *is a set of expressions of the form q or p_x, (The latter is just a predicate labelled with x) where p is a* state *of a symmetric system and x is a variable referrering to the process. If the template does not have a variable it is therefore called constant*

In above definition, p is the encoded state of a process as proposition and x is the process number that p is referring to. For example, p_1 is the state p of process 1, where as q because it doesn't have any variable is therefore a *constant.*
The *variables* in a *template* is bounded to the total number of identical processes in a specification of a symmetric system.

3.2 Creation

Algorithm 1 is used to produce a template from a disjuntion of propositions.

Algorithm 1

```
1: procedure templateCreation(Formula $\phi$)
2:     $\mathfrak{t} \leftarrow \emptyset$
3:     for all proposition $p$ in $\phi$ do
4:         if $p$ is constant then                ▷ if $p$ doesn't end with number
5:             $\mathfrak{t}.add(p, 0)$
6:         else
7:             $p' \leftarrow stateOf(p)$
8:             $k \leftarrow processOf(p)$
9:             if isUsed(k) then
10:                $v = thecurrentvariablethatisusedforprocessk$
11:                $\mathfrak{t}.add(p', v)$
12:            else
13:                $v = afreshvariable$
14:                $\mathfrak{t}.add(p'.v)$
15:            end if
16:        end if
17:    end for
18:    return $\mathfrak{t}$
19: end procedure
```

An example of using Algorithm 1 to create a template as follows: Supposedly we a have a disjunct ϕ such as

$$\phi = p_1 \vee p_2 \vee q_3 \vee r_3 \vee c$$

Lets assume these propositions refer to a symmetric system's states of p, p, q, r, c and numbers stating their unique process. For simplicity of tracking the relation between states and propositions same letters are used for refer to each state and its represntitive proposition. We define a template $\mathfrak{t} = \emptyset$.
Now lets look at the first proposition in ϕ, p_1 is state p belonging to process 1. since process 1 has yet used in $\mathfrak{t}$, we define a new *variant* x_1 and add p_{x_1} to $\mathfrak{t}$.
By looking at the next proposition in $\phi(p_2)$ we can add p_{x_2} to $\mathfrak{t}$. q_{x_3} can be added to $\mathfrak{t}$ with the same procedure from q_3. The next proposition is r_3. however process 3 has been used in $\mathfrak{t}$ before, therefore we are not going to create a new variant and used the variant that this process was referred to. Thus adding r_{x_3} to $\mathfrak{t}$. The last proposition is c. However this is not referred to any process. Therefore its a constant. so we simply add c to $\mathfrak{t}$. We finally end up with the following for our template.

$${}^1\mathfrak{t} = \{p_{x_1}, p_{x_2}, q_{x_3}, r_{x_3}, c\}^3$$

[1]Template set is referring to the total number of distinct processes used to create this template.

.

Definition 2 *Template lead is a means to expand a template t to be used for the same system with more processes.*

We say template $ŧ'$ is a symbolic lead of $ŧ$ iff

$$ŧ' = ŧ \cup q$$

where q is:

- $q = \emptyset$, or
- q is a set of all states in $ŧ$ that are marked with at most one variable within $ŧ$

Algorithm 2 is used to produce all the symbolic lead for a template. In addition to check if a template $ŧ'$ is a *lead* of $ŧ$ we can use Algorithm 3.

Algorithm 2

```
procedure TemplateSymbolicLead($template\ t$)
    let $lead$ be a set of Templates.
    array $v \leftarrow$ Distinct variables in $t$
    $lead.add(t)$                    ▷ every template is a symbolic lead of itself
    $newVar \leftarrow$ a variable that is not in $v$
    for all variant $v$ in $t$ do
        $s \leftarrow$ variants of $v$ in $t$
        $q \leftarrow \emptyset$
        for all $i \in s$ do
            $q.add(i_{newvar})$      ▷ Add variant $i$ with variable $newvar$ to q
        end for
        $lead.add(q)$
    end for
    return $lead$
end procedure
```

Algorithm 3

```
procedure checkTSL($templates\ t, t'$)          ▷ check if $t'$ is a TSL of $t$
    let set $t'' \leftarrow TemplateSymbolicLead(t)$
    for all template $tt$ in $t''$ do
        if $tt == t'$ then
            return $true$
        end if
    end for
    return $false$
end procedure
```

3.3 Producing clauses from Template

As we can create a template from a formula we can also produce the formula using a template. For example given *template*:

$$\mathfrak{t} = \{p_{x_1}, p_{x_2}, r_{x_2}, c\}^3$$

we can create the following DNF:

$$((p_1 \vee p_2 \vee r_2 \vee c) \wedge (p_1 \vee p_3 \vee r_3 \vee c) \wedge (p_2 \vee p_1 \vee r_1 \vee c) \wedge (p_2 \vee p_3 \vee r_3 \vee c) \wedge (p_3 \vee p_1 \vee r_1 \vee c) \wedge (p_3 \vee p_2 \vee r_2 \vee c))$$

Definition 3 *Two templates $\mathfrak{t}$,$\mathfrak{t}'$ are equal iff they produce identical clauses regardless of clause orderings for k processes.*

In Algorithm 4 it is shown how to create a DNF clauses from a *template*. Let's consider $\mathfrak{t}$ set. We know there are three processes invovled and the set contains four expressions $\{p_{x_1}, p_{x_2}, r_{x_2}, c\}$ with two variables $\{x_1, x_2\}$ and a constant c. Algorithm 4 simply create DNF clauses containing $\#Variables * \#Processes (2 * 3 = 6)$ disjunctions. since c is a *constant*, then it will be repeated in every clause. and variables will be replaced by process numbers. For example, if in a clause p_{x_1} is replaced by p_1 then p_{x_2} and r_{x_2} will be replaced by a process number that is not 1 meaning either p_2, r_2 or p_3, r_3 for that clause.

Algorithm 4

```
procedure CLAUSEFROMTEMPLATE($\mathfrak{t}$,$N$)                ▷ Template T and N processes.
    $mapper \leftarrow$ array of distinct variables
    $FF \leftarrow \emptyset$                                    ▷ A set of conjunctions of formula
    array A $\leftarrow \{1, \ldots, v\}$
    $changeable \leftarrow v - 1$
    while $true$ do
        $clause \leftarrow$ conjunction of constatns in T
        for i = 0 to v do
            for all $stat \in T$ do
                if $stat.variable = mapper[i]$ then
                    $clause \leftarrow clause \wedge state_{A[i]}$
                end if
            end for
        end for$FF.add(clause)$
        if $A[0] = n - v + 1$ then
            return FF                                        ▷ TERMINATE
        end if
        if $A[changeable] = (N - (v - (changeable + 1)))$ then
            $changeable - -$
        end if
        for $i \leftarrow v - 1$ $downto$ 1 do
            if $A[i] < (N - (v - (i + 1)))$ then
                $changeable = i$ ; $break$;
            end if
        end for
        $q \leftarrow populate(V, A, changeable)$
        $A \leftarrow q$
    end while
    return $FF$
end procedure
procedure POPULATE($V, A, changeable$)
    array q
    for $i = 0$ to changeable-1 do
        $q[i] \leftarrow A[i]$
    end for
    $q[changeable] \leftarrow A[changeable] + 1$
    for $i \leftarrow changeable + 1$ to $V$ do
        $q[i] \leftarrow q[changeable] + (i - changeable)$
    end for
    return q
end procedure
```

We now provide an overview of how we use symmetry to remove some of the complexity in temporal resolution refutations for larger instances of a problem. For simplicity of presentation, in what follows we assume that ϕ contains both the specification of the system and the negation of the property (and thus we are applying temporal resolution to ϕ in an attempt to show unsatisfiability) and that $\mathcal{E} = \{\Box\Diamond l\}$.

1. Instantiate a monodic first-order temporal formula ϕ to give the PTL formula ϕ_i with a fixed i (initially, $i = 1$).
2. Run the clausal propositional resolution method on ϕ_i.
3. If a contradiction is *not* derived, increment i and return to step 1 (in this case, the property requires some minimal number of processes before it holds). If a contradiction for i processes *is* derived, move to step 4.
4. From the results gathered from step 2, try to *guess* a loop set of potential loop formulae for the eventuality $\Box\Diamond l$ in the instance of the specification with a larger number of processes ϕ_j (with $j > i$).
5. If the loop set is empty then increment i and go to step 1, else continue to step 6.
6. Select a loop formula candidate L from the loop set and check if L is indeed a loop formula in this larger instance of the problem (i.e. ϕ_j). If the *loop check* procedure returns 'yes' then go to step 7, otherwise remove L from the loop set and return to step 5.
7. Once it is confirmed that L is indeed a loop formula, replace the eventuality rule applied for $\Box\Diamond l$ with the conculision $\neg L$ in ϕ_j and run the temporal theorem prover.
8. If a contradiction is obtained, go to step 1 setting $i = j$, otherwise go back to step 5.

Thus, if we *fail* to successfully guess a loop that works for refutations in problems with larger numbers of processes, then we must continue applying the full temporal resolution procedure to successively larger instances of the problem. If we do guess a suitable loop for a larger instance, then we can carry out proof in that instance *without* the necessity of temporal loop search (i.e. with a DSNF problem with empty $\mathcal{E}$) — this is *significantly* faster.

In what follows, we introduce machinery to analyse loop formulae collected from a successful run of the theorem prover on ϕ_i in step 2 of the algorithm.

3.4 "Guessing" Loop Formulae

In order to capture symmetry in the system, we have defined the notion of a *template*, which can be used to group together a set of formulae with the same behaviour and therefore, give a direction to our guesses. *Template* is mainly used to "guess" the loops needed for the temporal specification of the system containing more processes.

Given resolvent clauses for $\Diamond l$ of system ϕ_k , we create the templates of individul disjunct and add them to a set of *templates*. This procdure is called *grouping*.

Definition 4 *A group is the transformation of a template $\mathfrak{t}$ to a set of formulae F*

$$group : template \rightarrow F$$

The function to achieve such transformation is called grouping

To group the formulae with the 5 algorithm below, we assume that the give formulae is a resolvent of the loop for a symmetric system.

Algorithm 5

```
 1: procedure GroupFormulae($L$)    ▷ L is a loop formula in DNF form
 2:     let $F = \{l_1 , \ldots , l_k\}$
 3:     let $\mathfrak{t}$ be the template for a $l_i$ in $F$
 4:     let g= $\emptyset$
 5:     while $F \neq \emptyset$ do
 6:         for all $f$ in $F$ do
 7:             let $Q = \emptyset$
 8:             if $f$ can be represented with $\mathfrak{t}$ then
 9:                 add $f$ to $Q$ and remove $f$ from $F$
10:             end if
11:         end for
12:         add $t \mapsto Q$ to $g$
13:         let $\mathfrak{t}$ be the template for another $l_i$ in $F$
14:     end while
15:     return $g$
16: end procedure
```

In Algorithm 6 below, we return all the possibilities for $\mathfrak{t}'$ based on what we know about $\mathfrak{t}$.

Algorithm 6

```
procedure TemplateSymbolicLead(template $\mathfrak{t}$)
    let lead be a set of Templates.
    array $v \leftarrow$ Distinct variables in $\mathfrak{t}$
    $lead.add(\mathfrak{t})$                ▷ every template is a symbolic lead of itself
    $newVar \leftarrow$ a variable that is not in $v$
    for all variant $v$ in $\mathfrak{t}$ do
        $s \leftarrow$ variants of $v$ in $\mathfrak{t}$
        $q \leftarrow \emptyset$
        for all $i \in s$ do
            $q.add(i_{newvar})$        ▷ Add variant $i$ with variable newvar to q
        end for
        $lead.add(q)$
    end for
    return lead
end procedure
```

Similar to a loop formula, which can be thought of as a set of conjunctions, we define a *loop-template* as a *set* of templates $\mathfrak{t}_1$,$\mathfrak{t}_2$,...,$\mathfrak{t}_k$. Intuitively, a *loop template* represents all conjunctions of literals from a loop formula. A *loop-template* $\mathfrak{T}$ is a *predecessor* of $\mathfrak{T}'$ if, and only if,

- $\mathfrak{T}$ and $\mathfrak{T}'$ both contain the same number of *templates*, and
- for each template $\mathfrak{t}$ in $\mathfrak{T}$ there is a template $\mathfrak{t}'$ in $\mathfrak{T}'$ such that that $\mathfrak{t}'$ follows from $\mathfrak{t}$.

Theorem 1 *Let $\mathfrak{t}$, $\mathfrak{t}'$ be templates and L be a loop formula such that for no two conjuncts F, F' of L we have $F \subseteq F'$. Then if $\mathfrak{t}'$ follows from $\mathfrak{t}$, the loop formula L cannot contain instances of both $\mathfrak{t}$ and $\mathfrak{t}'$, unless $\mathfrak{t}'$ is a syntactic variation of $\mathfrak{t}$.*

PROOF Let t and t' be two templates for loop formula L, since t' follows from t, then $t' = t \cup q$ if $q = \emptyset$ then t' is a syntatic variabtion of t.(1) if $q \neq \emptyset$ then we have to prove that for every conjunct l_1 that is produced from t for n number of processes, there is a conjuct l_2 that will be produced from t' for n in such a way that $l_1 \subseteq l_2$ or $l_2 \subseteq l_1$. [2]
This will be sufficient because for every cunjuct F_1 and F_2 in L we have $F_1 \nsubseteq F_2$. Therefore this proves that L cannot contain both conjuncts from t and t' unless (1). Since l_2 is a bigger conjunts than l_1 because it has at least one more proposition. therefore,

$$l_2 \subseteq l_1$$

[2] number n is greater than the number of distinct varibles of t and t'.

Based on this result, we first present a naive GuessedLoopTemplate algorithm that enumerates possible loop templates that follow from the current loop template obtained by the GroupFormulae algorithm.

There are two problems with the algorithmGuessedLoopTemplate. First, it can potentially introduce n^2 guesses, where n is the maximum number of propositions in a loop formula , which can be too many to practically consider. The second, more serious problem, is that the algorithm only produces templates that follow from the current loop template. In some cases, the loop formula for an instance of the symmetric system for a larger number of processes is not an instance of any template for a smaller number of processes. Therefore, we need more information to produce a loop. The algorithm GuessedLoopTemplate2 produces a smaller number of guesses based on loop formulae for L_i and L_{i+1} where L_i is the loop for a specification with specification with i processes and L_{i+1} is the loop for specification for a specification with $i+1$ processes

Example. Suppose that for an instance of a symmetric system ϕ_i the loop formula is just a single conjunction of literals. Then the loop template is a singleton $\{t_i\}$, where $t_i = \{a_x, b_y\}$. Using GuessedLoopTemplate we obtain the following set of possible loop templates G for ϕ_{i+1}:

$$G = \{\{a_x, b_y\}, \{a_x, a_y, b_z\}, \{a_x, b_y, b_z\}\}.$$

Suppose now that the loop template for ϕ_{i+1} is $t_{i+1} = \{a_x, a_y, b_z\}$. Guided by this additional knowledge, the algorithm GuessedLoopTemplate2, generates a smaller guess for t_{i+2}:

$$G' = \{\{a_x, a_y, a_z, b_d\}, \{a_x, a_y, b_z\}\}.$$

Now, once we have a guess for a loop formula we need to check whether it is indeed a loop or not.

3.5 Checking Loop Guesses

In order to present an algorithm checking whether a given formula is a loop formula, we need to give more detail on how eventuality resolution works. Given a DSNF $(\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E})$ with $\mathcal{E} = \{\Box\Diamond l\}$, the loop formula L should satisfy the following properties (for details see [13]):

1. $\mathcal{U} \cup \mathcal{S} \models \Box(L \Rightarrow \bigcirc\neg l)$,
2. $\mathcal{U} \cup \mathcal{S} \models \Box(L \Rightarrow \bigcirc L)$.

It should be clear that under these conditions, $\Box(L \Rightarrow \bigcirc\Box\neg l)$ is a consequence of the DSNF as required to apply the eventuality resolution rule.

Both properties can be checked in a similar manner. To check (2), we form a new DSNF[3] $(\mathcal{I}', \mathcal{U}', \mathcal{S}', \mathcal{E}')$ as follows

$$\begin{array}{ll} \mathcal{I}' = \{L\} & \mathcal{S}' = \mathcal{S} \cup \{\bigcirc \neg L\} \\ \mathcal{U}' = \mathcal{U} & \mathcal{E}' = \emptyset \end{array}$$

and run temporal resolution on the resulting set of clauses. If a contradiction is derived, L is a loop formula for the original DSNF.

Theorem 2 *For a DSNF $(\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E})$ with $\mathcal{E} = \{\Box \Diamond l\}$ and a formula L we have $\mathcal{U} \cup \mathcal{S} \models \Box(L \Rightarrow \bigcirc L)$ if, and only if, $(\mathcal{I}', \mathcal{U}', \mathcal{S}', \mathcal{E}')$ is unsatisfiable.*

PROOF Loop formuale is created using *step* and *universal* clauses thus, the properties of the loop has nothing to do with the *start* and *eventuality* clauses. now let's assume our system specification is

$$\varphi = \mathcal{U} \cup \mathcal{S}$$

and our property to check is

$$\Box(L \Rightarrow \bigcirc L)$$

by *refutation* $\mathcal{U} \cup \mathcal{S} \models \Box(L \Rightarrow \bigcirc L)$ is valid *iff* $\varphi \wedge \neg(\Box(L \Rightarrow \bigcirc L)$ is *unsatisfiable.* Note:

$$(\neg(\Box(L \Rightarrow \bigcirc L) \Leftrightarrow \Diamond(L \wedge \bigcirc \neg L))$$

Another way of writing an *eventuality formuale* is

$$\Diamond(L \wedge \bigcirc L) \Leftrightarrow ((L \wedge \bigcirc \neg L) \vee \bigcirc(L \wedge \bigcirc \neg L) \vee \bigcirc \bigcirc (L \wedge \bigcirc \neg L) \ldots)$$

Since the assumption is that L is the loop formulae thus it is sufficient enough to check for satisfiability of the first portion of the formual that is :

$$\varphi \wedge (L \wedge \bigcirc \neg L)$$

Now we can create our DSNF formula to be the set:

$$\begin{array}{ll} \mathcal{I}' = \{L\} & \mathcal{S}' = \mathcal{S} \cup \{\bigcirc \neg L\} \\ \mathcal{U}' = \mathcal{U} & \mathcal{E}' = \emptyset \end{array}$$

[3]Notice that although L and $\neg L$ are not in the required clausal form, they can be easily transformed in this form by applying de Morgan rules.

3.6 Applying the Technique

The algorithms in the previous section have been applied to the verification of the MSI Protocol [11]. To check the "non-co-occurrence of states" property, we add the negated property:

$$\Box\Diamond(\exists x, y.(S(x) \wedge M(y)))$$

to the specification and then run the proof procedure. If the combined formula is unsatisfiable, then we know that the "non-co-occurrence" property holds.

It is also worth mentioning that, we are using two different Theorem Provers, namely TSPASS[26] and TeMP [24]. There are two reason for this combinations, one is that based on the specification one can perform better than other one. In addition, since they are used as 'Black Box' each can produces different set of useful outputs which then can be used for later investigations. To describe the approach in detail, we provide a walk through a run of the algorithms. All the used algorithms have been programmed, however they have not yet put together. Also the decision of what prover to be used is decided by the user. Nevertheless, the automation of each part is relatively easy and is set for our future work. First, we ran the temporal prover TSPASS on an instance of the MSI protocol with two processes. Through the internal processes of TSPASS, the property is transformed into

$$\Box\Diamond(\neg \mathit{wait\_for\_l})$$

and loop search returns the following loop formula $loop_2$:

$$(\mathit{wait\_for\_l} \wedge i_1) \vee (\mathit{wait\_for\_l} \wedge i_2) \vee \\ (\mathit{wait\_for\_l} \wedge m_1 \wedge m_2) \vee (\mathit{wait\_for\_l} \wedge s_1 \wedge s_2).$$

From this, we can extract the loop template T_2 to be:

$$\left\{ \begin{array}{c} \{\mathit{wait\_for\_l}, i_X\}, \\ \{\mathit{wait\_for\_l}, m_X, m_Y\}, \\ \{\mathit{wait\_for\_l}, s_X, s_Y\} \end{array} \right\}$$

Now we can use the GUESSEDLOOPTEMPLATE algorithm to create a set of loop templates TT to be used for MSI_3 and then derive a potential loop formula to be tested using the TESTLOOP algorithm. Unfortunately, at this stage we are unable to find an appropriate loop for MSI_3, and therefore turn to the GUESSEDLOOPTEMPLATE2 algorithm. We run TSPASS on MSI_3 and

[2]Time to prove property in MSI protocols without any changes

[3]Time to prove property with removing eventuality and adding the formulae as step clauses

Number of processes	Original Problem[2]	Modified Problem[3]	results
2	0.060s	0.011s	Unsatisfiable
3	1.240s	0.036s	Unsatisfiable
4	16.124s	0.134s	Unsatisfiable
5	119.662s	0.640s	Unsatisfiable
6	1717.886s	4.138s	Unsatisfiable
7	∞	35.108s	Unsatisfiable
8	∞	340.408s	Unsatisfiable
9	∞	4249.012s	Unsatisfiable

Table 3.1: Performance Comparison using the TeMP [24] Clausal Resolution Prover

extract the loop $loop_3$ from it:

$$\begin{gathered}(wait\_for\_l \wedge i_1 \wedge i_2)\\ \vee\ (wait\_for\_l \wedge i_1 \wedge i_3) \vee\ (wait\_for\_l \wedge i_2 \wedge i_3)\\ \vee\ (wait\_for\_l \wedge m_1 \wedge m_2 \wedge m_3) \vee\ (wait\_for\_l \wedge s_1 \wedge s_2 \wedge s_3)\\ \vee\ (wait\_for\_l \wedge s_1 \wedge s_2 \wedge i_3) \vee\ (wait\_for\_l \wedge s_1 \wedge s_3 \wedge i_2)\\ \vee\ (wait\_for\_l \wedge s_2 \wedge s_3 \wedge i_1)\\ \vee\ (wait\_for\_l \wedge m_1 \wedge m_2 \wedge i_3)\\ \vee\ (wait\_for\_l \wedge m_1 \wedge m_3 \wedge i_2)\\ \vee\ (wait\_for\_l \wedge m_2 \wedge m_3 \wedge i_1)\end{gathered}$$

The loop template for the above formulae is T_3 (X, Y and Z are variants):

$$\left\{\begin{gathered}\{wait\_for\_l, i_X, i_Y\},\\ \{wait\_for\_l, m_X, m_Y, m_Z\},\\ \{wait\_for\_l, s_X, s_Y, s_Z\}\},\\ \{wait\_for\_l, m_X, m_Y, i_Z\},\\ \{wait\_for\_l, s_X, s_Y, i_Z\}\end{gathered}\right\}$$

At this stage we use T_2 and T_3 in GuessedLoopTemplate2(T_2, T_3) to produce a set of template guesses TT_4 for MSI_4. As before, once we have the templates we produce the formulae and test them. One of the loop formulae candidates extracted from one of the templates in TT_4, namely from

$$\left\{\begin{gathered}\{wait\_for\_l, i_X, i_Y\},\\ \{wait\_for\_l, m_X, m_Y, m_Z, m_q\},\\ \{wait\_for\_l, s_X, s_Y, s_Z, s_q\},\\ \{wait\_for\_l, m_X, m_Y, m_q, i_Z\},\\ \{wait\_for\_l, s_X, s_Y, s_q, i_Z\}\end{gathered}\right\}$$

turns out to be the loop for MSI_4.

Using the same template and the same follow ups we can then produce a template for $MSI_5, MSI_6, ...$ Table 3.1 provides some practical results in establishing the non-co-occurrence property for the MSI protocol of up to 9 processes.

Chapter 4

Related work

A survey on the use of symmetry in model checking can be found in [34]. Indexed Simplified Computational Tree Logic (ISCTL), introduced in [16], allows one to specify and verify properties of parametrised systems, where the index corresponds to the number of processes. Our research differs in that our main focus is the speed of reasoning rather than representation issues.

Emerson and Kahlon [15] tackle the verification of temporal properties for parametrized model checking problems in an asynchronous systems. They reduced model checking for systems of arbitrary size n to model checking for systems of size (up to) a small *cutoff* size c. Furthermore, in [30], Pnueli, Ruah and Zuck used the standard deductive INV rule for proving invariance properties, to be able to automatically resolve by finite-state (BBD-based) methods with no need of theorem proving. They have developed a system to model check a small instances of the parametrised system in order to derive candidates for invariant assertions. Therefore, their work results in an incomplete but fully automatic sound method for verifying bounded-date parametrized system.

Giorgio Delzanno proposed a method [11] to verify safety properties of parametrized cache coherence protocols using *symbolic model checkers* for *infinite-state* based on *real arithmetics* [21].

In [8], Clarke, Jha and Filkorn investigated techniques for reducing the complexity of temporal logic model. They show how symmetry can be frequently used to reduce the size of the state space that must be explored during model checking in finite state systems.

Even though the work above are all concern using symmetry in model checking, our work differs from them as it provides a heuristic method to reduce the inner complexity of theorem provers itself rather than combining it with different techniques.

Chapter 5

Future work

The algorithms presented in this thesis constitute a first step towards practical deductive verification of symmetric systems. While the preliminary results presented in Table 3.6 are encouraging, we clearly need a much greater performance improvement before systems involving a realistically large number of processes can be verified.

We are also exploring ways to to extend the approach presented in this thesis to proof generalisation: Rather than just guessing loop formulae, one can try to construct the entire temporal proof for a larger number of processes guided by the proof for a smaller number of processes.

Chapter 6

Conclusion

Model checking is by far the most popular approach to verification. One of the reason is due to the large amount of research and development into implementation techniques that has been carried out. Especially there has been huge investigation in verifying systems that contain symmetry using *model checker*. The focus of this thesis was to investigate what it is meant for a system to be symmetric and how we could exploit symmetry in *theorem proving*. We focus on *Cache Coherence Protocol* which contains of many processes that carry out the same task. As the result once these systems are specified in *PTL* there can be many similarities between formulas. We later use FOTL specification as an inspiration to create the notion of *template*, which helps us to prove a larger symmetric system based on the proof that was carried out on the same system but with lower number of processes. This shows although *model checking* is more popular, *theorem proving* can still be competitive.

Bibliography

[1] Martín Abadi and Zohar Manna. Nonclausal temporal deduction. In *Proceedings of the Conference on Logic of Programs*, pages 1–15, London, UK, 1985. Springer-Verlag.

[2] Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. *J. ACM*, 37(2):279–317, 1990.

[3] Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of binary relations, 1995.

[4] Abdelkader Behdenna, Clare Dixon, and Michael Fisher. Deductive verification of simple foraging robotic behaviours. *Int. J. Intell. Comput. Cybern.*, 2(4):604–643, 2009.

[5] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Inf. Comput.*, 81(1):13–31, 1989.

[6] Muffy Calder and Alice Miller. Automatic Verification of any Number of Concurrent, Communicating Processes. In *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE)*, pages 227–230. IEEE Computer Society, 2002.

[7] Edmund M. Clarke, Ansgar Fehnker, Sumit Kumar Jha, and Helmut Veith. Temporal logic model checking. In *Handbook of Networked and Embedded Control Systems*, pages 539–558. Birkhäuser, 2005.

[8] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 450–462, London, UK, UK, 1993. Springer-Verlag.

[9] Anatoli Degtyarev, Michael Fisher, and Boris Konev. A Simplified Clausal Resolution Procedure for Propositional Linear-Time Temporal Logic. In *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *LNCS*, pages 85–99. Springer, 2002.

[10] Anatoli Degtyarev, Michael Fisher, and Alexei Lisitsa. Equality and monodic first-order temporal logic. *Studia Logica*, 72(2):147–156, 2002.

[11] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th International Conference on Computer Aided Verification (CAV '00)*, pages 53–68. Springer, 2000.

[12] Clare Dixon. Temporal resolution using a breadth-first search algorithm, 1998.

[13] Clare Dixon. Temporal resolution using a breadth-first search algorithm. *Annals of Mathematics and Artificial Intelligence*, 22:87–115, 1998.

[14] Clare Dixon. Using otter for temporal resolution. In *In Advances in Temporal Logic*, pages 149–166. Kluwer, 2000.

[15] E. Emerson and Vineet Kahlon. Reducing model checking of the many to the few. *Automated Deduction - CADE-17*, pages 236–254, 2000.

[16] E. A. Emerson and J. Srinivasan. A decidable temporal logic to reason about many processes. In *Proc. PODC'90*, pages 233–246. ACM, 1990.

[17] M. Fisher. *Introduction to Practical Formal Methods Using Temporal Logic.* John Wiley & Sons, 2011.

[18] Michael Fisher. A normal form for temporal logic and its application in theorem-proving and execution. *Journal of Logic and Computation*, 7:429–456, 1997.

[19] Michael Fisher, Clare Dixon, and Martin Peim. Clausal temporal resolution. *ACM Trans. Comput. Logic*, 2:12–56, January 2001.

[20] Michael Fisher and Alexei Lisitsa. Deductive verification of cache coherence protocols. In *Proc. 3rd International Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, pages 177–186, Southampton, UK, 2003.

[21] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. *HyTech: A model checker for hybrid systems*, pages 460–463. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[22] Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable fragments of first-order temporal logics, 1999.

[23] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[24] Ullrich Hustadt, Boris Konev, Alexandre Riazanov, and Andrei Voronkov. TeMP: A Temporal Monodic Prover. In *Proc. 2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 326–330. Springer, 2004.

[25] Boris Konev, Anatoli Degtyarev, Clare Dixon, Michael Fisher, and Ullrich Hustadt. Mechanising first-order temporal resolution. *Inf. Comput.*, 199(1-2):55–86, may 2005.

[26] Michel Ludwig and Ullrich Hustadt. Implementing a fair monodic temporal logic prover. *AI Commun.*, 23:69–96, April 2010.

[27] William Mccune. Otter 3.3 reference manual.

[28] Amir Niknafs-Kermani, Boris Konev, and Michael Fisher. Symmetric temporal theorem proving. *2012 19th International Symposium on Temporal Representation and Reasoning (TIME)*, 00:21–28, 2012.

[29] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[30] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97. Springer, 2001.

[31] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2-3):91–110, aug 2002.

[32] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12:23–41, 1965.

[33] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

[34] T. Wahl and A. F. Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2(2):799–847, 2010.

[35] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *CADE*, pages 140–145, 2009.