Symmetric Temporal Theorem Proving

Amir Niknafs-Kermani, Boris Konev, and Michael Fisher
Department of Computer Science, University of Liverpool, UK
Email: {niknafs, konev, mfisher }@liv.ac.uk

Abstract

In this paper we consider the deductive verification of propositional temporal logic specifications of symmetric systems. In particular, we provide a heuristic approach to the scalability problems associated with analysing properties of large numbers of processes. Essentially, we use a temporal resolution procedure to verify properties of a system with few processes and then generalise the outcome in order to reduce the verification complexity of the same system with much larger numbers of processes. This provides a practical route to deductive verification for many systems comprising identical processes.

1. Introduction

Automated theorem proving, or automated deduction, involves the mechanical proof of mathematical theorems by a computer program. It 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. We here consider automated temporal theorem proving, and focus on an extension of the resolution procedure developed by Robinson [23], namely the *clausal resolution* method for propositional temporal logic (PTL) [12]. As this proof technique has a exponential time complexity, its use becomes more difficult as the PTL formulae considered become larger. Our approach here 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 very regular way and if we have a very quick way to check correctness of the 'guesses'.

The stimulus for this work comes from temporal *model checking* [5], 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. For example, the *SPIN* model checker [18] has used symmetry in order to overcome the state space problems to some extent, and for some 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 sections we show, in Section 3, how refutations in larger problems can be 'guessed' from refutations in smaller, but similar, problems and checked for their correctness. In Section 4, we apply this technique to the deductive temporal verification of a *cache coherence protocol* [7]. For small numbers of identical processes, propositional temporal provers can handle their deductive verification. However, if the number of processes becomes larger, or if the formula is fairly complex, then even temporal resolution provers such as TeMP [19] or TSPASS [21] may well fail to handle such problems. By applying our technique to such symmetric problems, we can carry out automated temporal proofs for larger numbers of processes just by extrapolating from simpler examples.

Thus, the aim of this paper 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 [2, 24, 4], this paper shows how propositional temporal proving can be productively used in a similar way.

2. Preliminaries

PTL is an extension of classical propositional logic providing operators dealing with time. We assume a discrete and linear model of time, with finite past and infinite future (the temporal structure is thus isomorphic to the set of Natural Numbers $\mathbb{N}$). The set of well-formed PTL formulae is the smallest set containing all propositions and the Boolean constant $\top$, such that whenever ϕ and ψ are PTL formulae so are $\neg\phi$, $\phi \wedge \psi$, $\bigcirc\phi$ and $\phi \,\mathsf{U}\, \psi$. We call a *literal* a proposition or its negation. As always, other Boolean connectives and temporal operators ($\bot$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, $\square$, $\lozenge$, W) can be defined as abbreviations, for example, $\bot$ is defined as $\neg\top$; $\phi \vee \psi$ is defined as $\neg(\neg\phi \wedge \neg\psi)$; $\lozenge\phi$ is defined as $\top \,\mathsf{U}\, \phi$ and $\square\phi$ is defined as $\neg\lozenge\neg\phi$.

A *state* is a finite set of propositions, and a sequence of states $\sigma = s_1, s_2, \dots$ is an *interpretation*. For an interpretation σ, $i \in \mathbb{N}$ and a PTL formula ϕ, we define the relation $(\sigma, i) \models \phi$ by induction on the construction of well-formed PTL formulae as follows

$$\begin{array}{ll}
(\sigma,i) \models \top & \\
(\sigma,i) \models p & \text{iff } p \text{ is a proposition and } p \in s_i \\
(\sigma,i) \models \phi \wedge \psi & \text{iff } (\sigma,i) \models \phi \text{ and } (\sigma,i) \models \psi \\
(\sigma,i) \models \bigcirc\phi & \text{iff } (\sigma,i+1) \models \phi \\
(\sigma,i) \models \phi \,\mathsf{U}\, \psi & \text{iff there exists } j \geq i \text{ such that } (\sigma,j) \models \psi \\
 & \quad \text{and for all } k : i \leq k < j,\ (\sigma,k) \models \phi
\end{array}$$

We say that ϕ is true in σ (in symbols, $\sigma \models \phi$) if, and only if, $(\sigma, 0) \models \phi$. Formula ϕ is *valid* if it is true in any σ, while ϕ is unsatisfiable if there is no σ in which ϕ is true.

We say that a PTL formula ϕ *entails* a PTL formula ψ, in symbols $\phi \models \psi$, if, and only if, for every interpretation σ, in which ϕ is true, formula ψ is also true in σ. ψ is entailed by ϕ if, and only if, the PTL formula $\phi \Rightarrow \psi$ is valid (or $\phi \wedge \neg\psi$ is unsatisfiable). Establishing the (un)satisfiability of a PTL formula can be done with the help of clausal temporal resolution [11]. Clausal temporal resolution operates on formulae in a special normal form and then applies simple resolution-like inference rules in search of a contradiction.

It has been shown in [6] that every PTL formula can be translated in a satisfiability preserving way into *Divided Separated Normal Form* (DSNF), which consists of four parts, where l_a, l_b, l_c, l_d and l are literals:

- a set of initial clauses, $\mathcal{I}$, of the form $\bigvee\limits_a l_a$;
- a set of step clauses, $\mathcal{S}$, of the form
$$\Box(\bigwedge_c l_c \Rightarrow \bigcirc \bigvee_d l_d);$$
- a set of universal clauses, $\mathcal{U}$, of the form $\Box(\bigvee\limits_b l_b)$; and
- a set of eventuality clauses, $\mathcal{E}$, of the form $\Box\Diamond l$.

The meaning of a DSNF representation, $(\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E})$, is given by its characteristic formula

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

and, when we talk about properties of the DSNF (such as satisfiability), we mean the corresponding property of its characteristic formula. Here, and in the follows, we do not make a distinction between a set of formulae and the conjunction of formulae in this set. Notice that clauses from $\mathcal{I}$ are only required to hold at moment 0 while clauses from $\mathcal{U}$, $\mathcal{S}$, $\mathcal{E}$ are required to hold at any moment of time.

Importantly, it was shown in [6] that any DSNF problem can be transformed, in a satisfiability-preserving manner, to a DSNF problem having at most one element in $\mathcal{E}$.

The rules of clausal temporal resolution can be found in [12, 6]. The core of the method is the eventuality resolution rule, which deals with the eventuality clauses, and which can be abstractly described by:

$$\frac{\Box\Diamond l \qquad \Box(L \Rightarrow \bigcirc\Box\neg l)}{\Box\neg L}$$

The conclusion of an application of the rule is a universal clause. However, $\Box(L \Rightarrow \bigcirc\Box\neg l)$ can only be constructed by a combination of universal and step clauses. This *loop formula*, 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 formulae can be found in DSNF by using *loop search* [8] procedures, which are a key part of the clausal temporal resolution approach. Experiments show that, in practice, clausal temporal resolution provers spend most of their time searching for loop formulae and, therefore, speeding the loop search aspect up will significantly improve the performance of our provers.

3. Symmetric Temporal Theorem Proving

We now consider the application of the temporal resolution approach to the deductive verification of properties of parametrized systems, typically distributed systems comprising N identical processes. We will consider how to prove properties of such systems as N grows. Thus, the basic inputs to our method are sequences of PTL formulae (specifications) $\phi_1, \phi_2, \dots$ and (properties) $\psi_1, \psi_2, \dots$, representing instances of problems with 1, 2, etc, processes. Essentially, we aim to re-use information from the proof of the unsatisfiability of $\phi_i \wedge \neg\psi_i$ in order to prove the unsatisfiability of $\phi_{i+1} \wedge \neg\psi_{i+1}$.

To exploit symmetry in the input, every such ϕ_i should be an instance of the same parametrized problem, ϕ, and every ψ_i should be an instance of the same parametrized property, ψ. In this paper, we specify ϕ and ψ, in a first-order temporal language (FOTL) and then instantiate free variables of ϕ and ψ as elements of a fixed domain $D_n = \{1, \dots, N\}$. We sometime refer to elements of this domain D_n as *processes* as our examples here stem from the verification of parametrized distributed systems.

As a practical example, consider the *MSI cache-coherence protocol* [7]. This protocol involves a set of identical processes $P = \{p_1, p_2, \dots, p_N\}$. Each process $p_i \in P$ can be in one of the three different states of $\{i_i, s_i, m_i\}$, and if p_i is active then it can interact with memory. In Fig. 1 we provide a selection of PTL formulae used to describe the MSI protocol instantiated with N processes. A full explanation of MSI protocol in FOTL is given in [14].

We now proceed to develop some definitions which we will use in characterising the pattern of temporal resolution

$(i_1 \wedge i_2 \wedge \cdots \wedge i_N)\wedge$	At the start, all processes are in state i
$\Box(a_1 \vee a_2 \vee \cdots \vee a_N)\wedge$	There is always at least one active process
$\Box((s_1 \wedge \neg r_1) \wedge (r_1 \vee \cdots \vee r_N) \Rightarrow \bigcirc s_1)\wedge$	
$\vdots$	if a process p_i is in state s and is not reading and there
$\Box((s_N \wedge \neg r_N) \wedge (r_1 \vee \cdots \vee r_N) \Rightarrow \bigcirc s_N)\wedge$	exists a process that reads then in the next moment p_i is in state s
$\Box((m_1 \wedge \neg w_1) \wedge (w_1 \vee \cdots \vee w_N) \Rightarrow \bigcirc i_1)\wedge$	
$\vdots$	if a process p_i is in state m and is not writing and there
$\Box((m_N \wedge \neg w_N) \wedge (w_1 \vee \cdots \vee w_N) \Rightarrow \bigcirc i_N)\wedge$	exists a process that writes then in the next moment p_i is in state i
$\wedge \cdots$	

Figure 1. Fragment of the MSI Protocol PTL Specification (for n processes).

loop formulae occurring in parametrized problems.

Definition 1 *A one-symmetric system is a system that can be represented by a monadic FOTL formula with equality.*

For example, the formula ϕ:

$$\Box \forall x, y.(A(x) \wedge B(y) \Rightarrow \bigcirc c)$$

is a one-symmetric system where A and B are predicate names, which we also call *states* of a process, and c is a proposition. On the other hand,

$$\forall x, y.A(x, y)$$

is *not* one-symmetric because predicate A has arity greater than 1. Then, an instance ϕ_2 of the first-order temporal formula ϕ above is

$$\begin{array}{ccccc} \Box(a_1 \wedge b_1 \Rightarrow \bigcirc c) & \wedge & \Box(a_2 \wedge b_1 \Rightarrow \bigcirc c) & \wedge \\ \Box(a_1 \wedge b_2 \Rightarrow \bigcirc c) & \wedge & \Box(a_2 \wedge b_2 \Rightarrow \bigcirc c) & \end{array}$$

and for the formula ψ:

$$\exists x. \bigcirc P(x)$$

its instance ψ_N is

$$\bigcirc(p_1 \vee p_2 \vee ... \vee p_N).$$

It should be clear that the MSI protocol presented above can be represented by a monadic FOTL formula and is, therefore, a one-symmetric system.

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 monadic 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 negation rule applied for $\Box\Diamond l$ with the conculision of $\neg L$ 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.

3.1. "Guessing" Loop Formulae

In order to capture symmetry in the system, we define 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. This template is also used to "guess" the loops needed for the temporal specification of the system containing a larger number of processes.

Definition 2 *A* template *is a set of expressions of the form p or p_x, where p is a state and x is a variable.*

Unlike monadic temporal formulae used to specify parametrized problems, templates are instantiated in such a way that different variables are mapped into different domain elements. To highlight this difference, we call variables occurring in a template *variants*. For example, in a template

$$\{a_x, b_y\}$$

variants x and y can correspond to any process i and j where $i \neq j$. For D_2, i.e. $\{1, 2\}$, the above template instantiates into two sets $\{a_1, b_2\}$ and $\{a_2, b_1\}$. Conversely, given a PTL formula

$$f = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_2 \wedge b_1 \wedge c)$$

we can extract two templates:

- $\{a_x, b_x\}$ and
- $\{a_x, b_y, c\}$

Note that $f' = f \vee (a_1 \wedge b_2 \wedge c)$ gives rise to the same templates as *f*.

Theorem 1 *A conjunction of propositional literals has at most one template.*

Given a loop formula L (recall that a loop formula is in DNF), we simultaneously search for templates for conjunctions in L and group these conjunctions together so that all disjunctions with identical templates will be grouped together. We call this process *grouping*.

Definition 3 *A group is a function that maps a template t to a set of formulae F*

$$group : template \rightarrow F$$

To group the formulae with the GroupFormulae algorithm below, we assume that the formulae are a loop (in DNF form) for a one-symmetric system.

We say that a template t' *follows* from another template t if, and only if,

$$t' = t \cup q,$$

where q has the following properties:

Algorithm 1

```
1:  procedure GroupFormulae($L$)        ▷ L is a loop formula in form of $\bigvee_{j=1}^{k} l_j$
2:      let $F = \{l_1, \ldots, l_k\}$
3:      let $t$ 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 $t$ 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 $t$ be the template for another $l_i$ in $F$
14:     end while
15:     return $g$
16: end procedure
```

- q is either empty
- or, if q not empty, then it contains all the states in t that are marked with the same variant with the variant renamed with a fresh one.

For example, given a template $t = \{a_x, b_x, c_y\}$; a_x and b_x are both marked with x, so a possible q is $\{a_z, b_z\}$, where z is a new variant name.

In the templateFollowUps algorithm below, we return all the possibilities for t' based on what we know about t.

Algorithm 2

```
1: procedure templateFollowUps($template\ t$)
2:     let $tem = t$              ▷ every template follows itself
3:     for all variant $v$ in $t$ do
4:         Define $q$ to be the set of elements in $t$ that have
           $v$ as their variant (with new variant assigned to $q$)
5:         $tem$.add($t \cup q$)
6:     end for
7:     return $tem$
8: 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 t_1,t_2,...,t_k. Intuitively, a loop template represents all conjunctions of literals from a loop formula. A loop-template T is a *predecessor* of T' if, and only if,

- T and T' both have the same number of templates, and
- for each template t in T there is a template t' in T' such that that t' follows from t.

Theorem 2 *Let t, 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 t' follows from t, the loop formula L cannot contain instances of both t and t', unless t' is a syntactic variation of 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

Algorithm 3

```
1: procedure GUESSEDLOOPTEMPLATE($L$)
2:     let $g$ = GROUPFORMULAE($L$)
3:     let $guessedSet = \emptyset$
4:     let $templateArray[g.size]$
5:     for all template $t_i$ in $g$ do
6:         templateArray[i] = TEMPLATEFOLLOWUPS($t_i$)
7:     end for
8:     take a template from each $templateArray[i]$ add to $guessedSet$ as a new guessedLoopTemplate
9:     repeat the previous step until there is no more combinations left.
10:    return $guessedSet$
11: end procedure
```

problems with the GUESSEDLOOPTEMPLATE algorithm. First, it can potentially introduce n^2 guesses, where n is the maximum number of propositions in a loop formula conjunct, 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 GUESSEDLOOPTEMPLATE2 algorithm, gener-

Algorithm 4

```
1: procedure GUESSEDLOOPTEMPLATE2($L_i$,$L_{i+1}$)
2:     let $g$ = GROUPFORMULAE($L$)
3:     let $g'$=GROUPFORMULAE($L_{i+1}$)
4:     let $guessedSet = \emptyset$
5:     let $templateArray[g'.size]$
6:     for all template $t'_i$ in $g'$ do
7:         if $t'_i$ follows a template $t \in g$ then
8:             let $q = t'_i/t$ with a new variant
9:             let $t'' = t'_i \cup q$
10:            templateArray[i]=$\{t'', t'_i\}$
11:        else
12:            templateArray[i] = TEMPLATEFOLLOWUPS($t'_i$)
13:        end if
14:    end for
15:    take a template from each $templateArray[i]$ add to $guessedSet$ as a new guessedLoopTemplate
16:    repeat the previous step until there is no more combinations left.
17:    return $guessedSet$
18: end procedure
```

ates 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.

3.2. 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 [8]):

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[1] $(\mathcal{I}', \mathcal{U}', \mathcal{S}', \mathcal{E}')$ as follows

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

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

[1]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.

Theorem 3 *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.*

The TESTLOOP algorithm checks whether a formula L is indeed a loop formula. Note that carrying out such a check is typically *much* faster than attempting to find the loop itself.

Algorithm 5

```
1:  procedure TESTLOOP( $L$ , $\phi_i$)
      ▷ $L$ is a guessed loop for $\Box\Diamond l$
      ▷ $\phi_i$ is the specification of the system
2:    if $L \not\models \bigcirc\neg l$ then
3:      return false
4:    end if
5:    $\phi_i \leftarrow \phi_i$ - {initialClauses $\cup\ \Box\Diamond l$ }
6:    $\phi_i \leftarrow \phi_i \wedge (L) \wedge \bigcirc(\neg L)$
7:    if $\phi_i \models \bot$ then
8:      return true                          ▷ $L$ is a loop
9:    else
10:     return false
11:   end if
12: end procedure
```

4. Applying the Technique

The algorithms described in the previous section have been applied to the verification of the MSI Protocol [7]. 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[21] and TeMP [19]. 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

$$\Box\Diamond(\neg wait\_for\_l)$$

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

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

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

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

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 extract the loop $loop_3$ from it:

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

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

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

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

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

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 4

[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 1. Performance Comparison using the TeMP [19] Clausal Resolution Prover

provides some practical results in establishing the non-co-occurrence property for the MSI protocol of up to 9 processes. We have also successfully applied this technique

to proving properties of foraging behaviours of robotic swarms of increasing sizes described in [1]. If we run the Temp prover to prove some properties in this system without using the algorithm described in this paper, it could prove for up to 3 processes, however, once we use this techniques we can prove the properties for up to 11 processes, thought we have not space to present the full results here.

5. Concluding Remarks

Temporal specification and verification have many applications, not only through model-checking techniques [17, 3], but also in security protocol analysis [16, 20], analysis of parametrized systems [13], and temporal planning [15]. Deductive verification reduces checking properties of a system to checking the satisfiability of a particular formula. While this approach to verification is appealing due to its conceptual simplicity, it suffers from complexity issues as the size of the temporal formulae grows.

In this paper we have considered a simplified, but widely applicable, model based on parametrized systems. The key idea is to consider a number of, essentially identical, components whose temporal specification leads to similar formulae. Then we use a standard clausal temporal resolution prover to prove some property of a system comprising a small number of processes. Once we have this proof, the *loop formula* within the clausal resolution refutation is captured and generalised for use in problems with a larger number of processes. Effectively, we use information about refutations on smaller versions of the problem to help us ‘guess’ key formulae in refutations for larger problems. This guessing process, though heuristic, is easy to carry out. Importantly, it is also relatively easy to check whether the key formulae we guess do, in fact, lead us to a refutation for our larger problem.

We have provided here the basis for this approach, implemented the key algorithms, and have applied it to the analysis of some parametrized systems.

5.1 Related Work

A survey on the use of symmetry in model checking can be found in [25]. Indexed Simplified Computational Tree Logic (ISCTL), introduced in [10], 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 [9] 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 [22], 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. 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.

5.2 Future Work

The algorithms presented in this paper constitute a first step towards practical deductive verification of symmetric systems. While the preliminary results presented in Table 4 are encouraging, we clearly need a much greater performance improvement before systems involving a realistically large number of processes can be verified. In addition, the procedure itself is not yet fully automated, which is a part of future work.

We are looking at a possibility to extend this approach beyond the limitations of the one-symmetric systems. Point-to-point communication, even between absolutely identical processes, cannot be described by unary first-order predicates. One advantage of one-symmetric systems is that it suffices to require that different variants in a template are instantiated as different domain elements. Generalising the notions of a template and a loop template to the n-ary case proves to be challenging as we have to deal with more sophisticated restrictions on variants.

We are also exploring ways to to extend the approach presented in this paper 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.

Finally, we are looking for a wider variety of examples to which we can apply this approach. This will not only test and refine the approach, but will also let us to identify sub-classes of parametrized systems in which this approach works particularly well.

References

[1] A. Behdenna, C. Dixon, and M. Fisher. Deductive Verification of Simple Foraging Robotic Behaviours. *Int. J. Intell. Comput. Cybern.*, 2(4):604–643, 2009.

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

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, pages 209–224, Berkeley, USA, 2008. USENIX Association.

[4] M. Calder and A. 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.

[5] E. M. Clarke, A. Fehnker, S. K. Jha, and H. Veith. Temporal Logic Model Checking. In *Handbook of Networked and Embedded Control Systems*, pages 539–558. Birkhäuser, 2005.

[6] A. Degtyarev, M. Fisher, and B. 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.

[7] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proc. 12th International Conference on Computer Aided Verification (CAV '00)*, pages 53–68, London, UK, 2000. Springer.

[8] C. Dixon. Temporal Resolution Using a Breadth-First Search Algorithm. *Annals of Mathematics and Artificial Intelligence*, 22:87–115, 1998.

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

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

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

[12] M. Fisher, C. Dixon, and M. Peim. Clausal Temporal Resolution. *ACM Trans. Comput. Logic*, 2:12–56, January 2001.

[13] M. Fisher, B. Konev, and A. Lisitsa. Practical Infinite-State Verification with Temporal Reasoning. In *Verification of Infinite State Systems and Security*, volume 1 of *NATO Security through Science Series: Information and Communication*, pages 91–100. IOS Press, 2006.

[14] M. Fisher and A. 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.

[15] M. Fox and D. Long. Time in Planning. In *Handbook of Temporal Reasoning in AI*, pages 497–537. Elsevier Science, 2005.

[16] J. Y. Halpern. Reasoning about knowledge: A survey. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 1–34. Oxford University Press, 1995.

[17] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *SPIN*, 1999.

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

[19] U. Hustadt, B. Konev, A. Riazanov, and A. 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.

[20] S. A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

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

[22] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. pages 82–97. Springer, 2001.

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

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

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