Symmetric Symbolic Safety-Analysis of Concurrent Software with Pointer Data Structures

Farn Wang*
Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.
`farn@iis.sinica.edu.tw`

Karsten Schmidt†
Dept. of Computer Science, Carnegie-Mellon University
Pittsburgh, PA, 15213, USA, `kschmidt@cs.cmu.edu`

Abstract

Pointer is a very convenient device for constructing dynamic networks and passing parameters in complex software. But with bugs like dirty pointers, it also creates challenges in maintaining system functionality. We formally define the model of software with pointer data structures. Model checking such software may cost tremendous resources because of the dynamic data structure. We developed symbolic algorithms for the manipulation of conditions and assignments with indirect operands for verification with BDD-like data-structures. We rely on two techniques, including inactive variable elimination and process-symmetry reduction in the network configuration, to contain the time and memory complexity. We argue for the indispensibility of process-symmetry reductoin in model-checking such systems and laid the theoretical groundwork for the discussion of symmetry reduction. We propose the efficient technique of IbSINC (Incomplete but Sound Isomorphism of Network Configuration) reduction to avoid the expensive but complete symmetry reduciton. We then identify the anomaly of image false reachability of the IbSINC reduction and also define a useful class of symmetric systems, for which the efficient IbSINC reduction works well without the anomaly problem. We implemented the techniques in the RED tool and tested it against the Mellor-Crummy and Scott's locking algorithms and several other benchmarks. The performance comparison with tool SMC shows that for the special class of pointer-data-structure concurrent systems, our technique can lead to significant performance improvement.

Keywords: symmetry, symbolic model-checking, pointers, data-structures

1 Introduction

Model checking[6] of networks with special topologies like rings and buses has been widely studied. In real-world software, arbitrary and dynamic network configuration is, however, often constructed using pointers. An action like "$x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n := \ldots;$" can stretch through a network and change the local memory of a peer process in the network. Such indirect references are not only very common in practice, but also extremely important in both hardware and software engineering. For example, most CPUs now support hardware indirect addressing to facilitate virtual memory management. This important hardware indirect referencing mechanism is transparent to softwares and runs silently. For another example, dynamic data-structures like linear lists, trees, and graphs are constructed with pointers and used intensively in most nontrivial softwares. In example 1, we have a locking algorithm[16] which uses pointers to maintain a queue for critical section mutual exclusion.

Example 1 : MCS (Mellor-Crummy & Scott's) locking algorithm The algorithm[16] is an example protocol in which a global waiting queue of processes is explicitly used to insure mutual exclusive access to the critical section in a concurrent system. In figure 1, The MCS locking algorithm for a process is drawn as a finite-state automaton.

---

[*] The work is partially supported by NSC, Taiwan, ROC under grants NSC 90-2213-E-001-006, NSC 90-2213-E-001-035, and the by the Broadband network protocol verification project of Institute of Applied Science & Engineering Research, Academia Sinica, 2001.

[†] supported by DARPA/ITOP within the MoBIES project

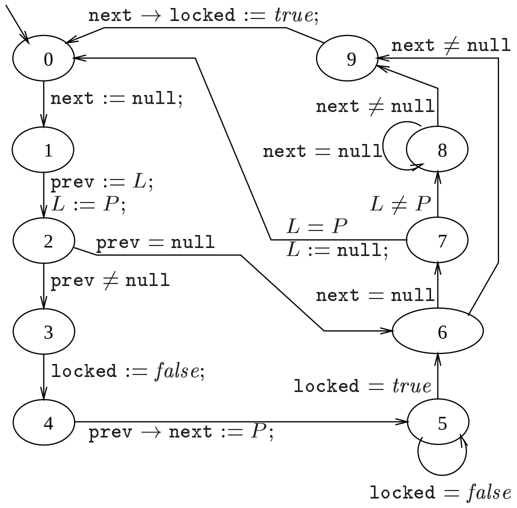

Figure 1: MCS locking algorithm

The critical section is from modes 6 to 9. The queue is constructed with one global pointer `L` (to the tail of the queue), and two local pointers of each process: `next` and `prev` (respectively to the successor and predecessor processes of the local process in the queue). P is a special symbol for the data structure address of the running process. Each process also has a local Boolean variable `locked` which is set to false when the process is permitted to the critical section by its predecessor in the queue. We want to guarantee that at any moment, no more than one process is in its critical section. ||

MCS locking algorithm is just one example for the extensive usage of dynamic pointer data-structures in software industry. In real-world, the data-structures can be defined and grow to configurations like stacks, layered trees, multi-dimensional (sparse or triangular, to make it more complicate) matrices, and even random graphs. Due to their dynamic nature, software with such data-structures has been known to be extremely difficult to maintain and to debug. For example, any experienced software engineers will agree that bugs caused by dirty pointers to freed data-structures are extremely difficult to detect and remove. It is almost like a nightmare! Such bugs, whose effect usually does not emerge until long after a data-structure is corrupted through a dirty pointer, is very difficult to trace backward. This nightmare serves as the motivation for the research reported in this manuscript.

The technique of symbolic model-checking manipulates logic predicates describing state-spaces. Since the technique can usually handle large sets of states in an abstract and concise way, it provides opportunity for higher efficiency in verification. In the last decades, BDD (Binary-Decision Diagram)[2, 5] has emerged as one of the prime industry technology in symbolic manipulation. In this paper, we have the following accomplishments toward using BDD technology for the formal verification of concurrent software with pointer data-structures.

- We define a formal model for concurrent software with pointer data structures for rigorous research on solution for the problem. Especially note that the framework is defined in such a way to explicitly allow all processes to share the same automaton template but at the same time allow them to use their local variables. This is extremely important in identifying process-symmetry in a convincing way. To our best knowledge, most other model-checkers[3, 4, 11, 12, 14, 26] accept that each process be described with its own automaton and usually create difficulty in efficiently identifying symmetric behaviors among the process automata.
- We present algorithms for the symbolic manipulation of conditions and assignments with indirect operands for verification with BDD-like data-structures. Algorithms for both forward analysis and backward analysis have been developed and implemented with tuning for verification performance. Special care is taken to allow for recurrence assignments, like $y \rightarrow x := 3y \rightarrow x + z$; where the left-hand-side may also occur in the right-hand-side.
- Then we discuss how to adapt two reduction techniques for model checking such systems.
    - *Reduction by inactive variables eliminations,* which helps the construction of concise state-space representation through the elimination of variable valuations that do not affect the system behaviors. Such a

technique has been used heavily in tools like Spin[11, 12], UPPAAL[4], SGM[14, 26], and `red`[22, 23, 24, 25]. Due to the implicit reading of pointer values in the indirection of operand references, the adaptation is not so trivial.

  - *Reduction by process-symmetry.* The idea of symmetry reduction [15, 21, 7, 9, 14, 26, 18, 22] is to keep only one state if two states turn out to be symmetric. We argue that, with the dynamic variety of network configurations that can be constructed with pointers, symmetry reduction becomes a must in verifying such systems. We shall follow the approach of process-symmetry in [9, 26, 14, 22, 23] since process represents a typical basic unit for behavioral equivalence in symmetry. A challenge here is how to design an efficient strategy to detect the equivalence among processes with process-symmetry in their dynamic data-structures. We review some of the theoretical framework of transposition (or permutation) as our groundwork for symmetry reduction. Since graph-isomorphism is generally considered a very difficult problem, the complete detection of a true symmetry relation between two states can be costly. We thus developed the reduction strategy of *incomplete but sound isomorphism of network configuration* (IbSINC reduction in short), which can be computed efficiently. We identify the anomaly of image false reachability in the IbSINC reduction. We also define *symmetric systems*, in which IbSINC reduction works well and the anomaly will not happen.

- We also implemented our modelling and verification techniques for pointer data-structure systems in our model-checker `red` version 3.1, which is available freely at `http://www.iis.sinica.edu.tw/~farn/red/`, for timed automata. The implementation not only support pointer data-structures but also support complex arithmetics on addresses (or pointers, or identifiers).
- We carried out experiments on several benchmarks to show the usefulness of our techniques and the indispensibility of the IbSINC reduction and make performance comparison with SMC[10] of Emerson, et al. The benchmarks represent dynamic configurations of doubly-linked queues, doubly-linked cycles, and forests with arbitrary number of children to internal nodes. The fact that our tool performs well with the diversity of the dynamic data-structures shows great promise of our techniques for the special class of pointer data-structure concurrent systems.

In section 2, we shall define the formal framework of this research. In section 3, we shall present the algorithmic framework to integrate safety-analysis software with various reduction techniques. In section 4, we shall present the algorithm for the manipulation of symbolic predicates and symbolic assignment statements with BDD-like data-structures. In section 5, we shall discuss how to use the implicit pointer-reading operations in indirect references to facilitate the reduction by elimination of inactive variables. In section 6, we shall lay a firm groundwork on the research of symmetry reduction and investigate some of the theoretical properties of the problem of identifying isomorphism in general directed graphs. Especially, we identify the anomaly of image false reachability with bijections. In section 7, we discuss our implememntations and performance data collected with three benchmarks. Specifically in subsection 7.1, we present our efficient IbSINC reduction. Performance comparison with SMC[10] is also reported. Section 8 is the conclusion.

2 Concurrent algorithms and the safety analysis problem

For convenience, we consider concurrent algorithms with local data structures attached to each process for convenience of presentation and discussion. The address of a data structure can be viewed as the identity of the corresponding process. We shall have the convention that if p is the address of a process's data-structure, then the process is also named p. But our model and techniques can be easily adapted for the modelling and verification of systems with data-structure addresses not bound to process identifiers.

Two types of variables can be declared. The first is the type of *discrete variables* with predeclared finite integer value ranges. For each declared variable x, $\mathrm{lb}(x)$ and $\mathrm{ub}(x)$ denote its declared lowerbound and upperbound respectively. Such variables can be used in formulae and assignments with arithmetic expressions and indirect operands. For convenience, we can also assign symbolic macro names to integer values. Traditionally, *false* is interpreted as 0 while *true* as 1.

The second is the type of *pointers (address variables)* to processes (data-structures). The value ranges of pointers are from zero (or `NULL`) to the number of processes. As in example 1, L is used as a pointer to the tail of a queue. We also support arbitrary address arithmetics. A special pointer value constant symbol is `NULL`, which in C's tradition is equal to zero. Or in the same notations as of discrete variables, $\mathrm{lb}(x) = \texttt{NULL}$ and $\mathrm{ub}(x)$ is the number of processes for all declared pointers x.

Variables can be declared as *global* variables which all processes can access, or *local* variables of a process which only the declaring process can directly access. A name can be used to represent the respective local variables of different processes. For instance, in example 1, different processes access different variables which are all locally called `locked`.

In the following, we shall first formally define the syntax and semantics of our systems, and then define the safety analysis problem.

2.1 Syntax of algorithm descriptions

Conceptually, a concurrent algorithm S is a tuple $(G^d, G^p, L^d, L^p, A(P))$ where G^d and L^d are respectively the sets of *global* and *local discrete variables*, G^p and L^p are respectively the sets of *global* and *local pointers*, and $A(P)$ is the *process program template*, with *process identifier* symbol P.

Given a set X^d of global and local discrete variables and a set X^p of global and local pointers, a *local state predicate* η of X^d and X^p can be used to describe the triggering condition of state transitions and has the following syntax.

$$\begin{array}{rl} \eta ::= & \epsilon_1 \sim \epsilon_2 \mid \neg\eta \mid \eta_1 \vee \eta_2 \\ \epsilon ::= & c \mid \mathtt{NULL} \mid P \mid x \mid y \rightarrow \epsilon \mid x[p] \mid y[p] \rightarrow \epsilon \mid \epsilon_1 \oplus \epsilon_2 \end{array}$$

where $\sim \in \{\leq, <, =, \neq, >, \geq\}$, $c \in \mathcal{N} - \{0\}$, $x \in X^d \cup X^p$, $y \in X^p$, and $\oplus \in \{+, -, *, /\}$. Parenthesis can be used for disambiguation. Traditional shorthands are $\epsilon_1 \neq \epsilon_2 \equiv \neg(\epsilon_1 = \epsilon_2)$, $\eta_1 \wedge \eta_2 \equiv \neg((\neg\eta_1) \vee (\neg\eta_2))$, and $\eta_1 \Rightarrow \eta_2 \equiv (\neg\eta_1) \vee \eta_2$, Thus a process may operate on conditions of the global and local variables, and also on the local variables of peer processes pointed to by pointers. We let $B(X^d, X^p)$ be the set of all local state predicates constructed on the discrete variable set X^d and the pointer set X^p.

In our concurrent algorithms, once the triggering condition is satisfied by global variables and the local variables of a process, the process may execute a finite sequence of actions with the following syntax: "$y_1 \rightarrow y_2 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;$," where $n \geq 0$. Conveniently, let $T(X^d, X^p)$ be the set of all finite sequences of actions constructed of discrete variable set X^d and pointer set X^p.

Given a concurrent algorithm $S = (G^d, G^p, L^d, L^p, A(P))$, $A(P)$ is the program template, with identifier symbol P, for all processes. Program template $A(P)$ has a syntax similar to that of finite-state automata. $A(P)$ is conceptually a tuple (Q, q_0, E, τ, π) with the following restrictions:

- Q is a finite set of operation modes.
- $q_0 \in Q$ is the initial operation mode.
- $E \subseteq Q \times Q$ is the set of transitions between operation modes.
- $\tau : E \mapsto B(G^d \cup L^d, G^p \cup L^p)$ is a mapping that defines the triggering condition of each transition.
- $\pi : E \mapsto T(G^d \cup L^d, G^p \cup L^p)$ is a mapping that defines the action sequence performed upon occurrence of a transition. *We assume that transitions are atomic actions.*

We require that there is a variable `mode` $\in L^d$ that records the current operation mode of the corresponding process. However, when drawing $A(P)$ as an automaton like in figure 1, we omit the description of `mode` values in the triggering conditions and action sequences for simplicity and clarity.

2.2 Computation of systems

Given a system of $\mathcal{M}$ processes, we assume the processes are indexed with integer from 1 to $\mathcal{M}$. Given a concurrent algorithm S, $S^{\mathcal{M}}$ denotes the implementation of S by exactly processes one through $\mathcal{M}$. A *state* ν of $S^{\mathcal{M}}$ is a mapping from

$$\{\mathtt{NULL}, 1, \ldots, \mathcal{M}\} \times (\mathcal{N} \cup G^d \cup G^p \cup \{\perp, \mathtt{NULL}, P\} \cup L^d \cup L^p)$$

such that

- $\nu(\mathtt{NULL}, x) = \perp$ (memory fault) for all $x \in \mathcal{N}$ and all variable x.
- for all $1 \leq p \leq \mathcal{M}$, $\nu(p, \perp) = \perp$; $\nu(p, P) = p$; $\nu(p, c) = c$ if $c \in \mathcal{N}$; and
    - for all $x \in G^d$, $\nu(p, x) \in [\mathrm{lb}(x), \mathrm{ub}(x)]$ is the value of global discrete variable x at state ν;
    - for all $x \in G^p$, $\nu(p, x) \in \{\mathtt{NULL}\} \cup \{1, \ldots, \mathcal{M}\}$ is the value of global pointer x at state ν;
    - for all $x \in L^d$, $\nu(p, x) \in [\mathrm{lb}(x), \mathrm{ub}(x)]$ is the value of local discrete variable x of process p at state ν; and
    - for all $x \in L^p$, $\nu(p, x) \in \{\mathtt{NULL}\} \cup \{1, \ldots, \mathcal{M}\}$ is the value of local pointer x of process p at state ν.

Given a global state ν, a process $1 \leq p \leq \mathcal{M}$, and a process predicate $\eta \in B(G^d \cup L^d, G^p \cup L^p)$, we define the mapping of p *satisfies* η at ν, written $\nu(p, \eta)$, to $\{\mathit{true}, \mathit{false}, \bot\}$ in the following inductive way.

- $\nu(p, y \rightarrow \epsilon) = \nu(\nu(p, y), \epsilon)$ if $p \neq \mathtt{NULL}$.
- $\nu(p, y[c] \rightarrow \epsilon) = \nu(c, y \rightarrow \epsilon)$ if $1 \leq c \leq \mathcal{M}$; otherwise, $\nu(p, y[c] \rightarrow \epsilon) = \bot$.
- $\nu(p, \epsilon_1 / \epsilon_2) \ = \bot$ if $\oplus =$ '/' $\wedge\, \nu(p, \epsilon_2) = 0$.
- $\nu(p, \epsilon_1 \oplus \epsilon_2) \ = \ \nu(p, \epsilon_1) \oplus \nu(p, \epsilon_2)$ if either $\oplus \in \{+, -, *\}$ or $\oplus =$ '/' $\wedge\, \nu(p, \epsilon_2) \neq 0$. Integer-division is assumed, that is x/y is defined as $\frac{x*y}{|x*y|}\lfloor |x/y| \rfloor$, where $\frac{x*y}{|x*y|}$ is the sign of x/y.
- $\nu(p, \epsilon_1 \sim \epsilon_2) \ = \ \nu(p, \epsilon_1) \sim \nu(p, \epsilon_2)$
- "$\bot \sim \epsilon$" equals to $\bot$ and "$\epsilon \sim \bot$" equals to $\bot$.
- The negation of the satisfaction mapping is defined as follows.

$\nu(p, \eta)$	*false*	$\bot$	*true*
$\nu(p, \neg\eta)$	*true*	$\bot$	*false*

- The disjunction of the satisfaction mapping is defined as follows.

$\nu(p, \eta_1 \vee \eta_2)$	*false*	$\bot$	*true*
false	*false*	$\bot$	*true*
$\bot$	$\bot$	$\bot$	$\bot$
true	*true*	$\bot$	*true*

Given an action α of S, the new global state obtained by applying $y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;$, with $n \geq 0$, to p at ν, written $\text{next\_state}(p, \nu, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;)$, is defined as follows:

- When $\nu(p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x) \neq \bot$ and $\nu(p, \epsilon) \neq \bot$, $\text{next\_state}(\nu, p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;)$ is identical to ν except that $\text{next\_state}(\nu, p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;)(\nu(p, y_1 \rightarrow \ldots \rightarrow y_n), x) = \nu(p, \epsilon)$.
- When either $\nu(p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x) = \bot$ or $\nu(p, \epsilon) = \bot$, $\text{next\_state}(\nu, p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon;)$ is undefined.

Note that the semantics is defined to allow for recurrence of a variable in both the left-hand-side and right-hand-side of an assignment. Given an action sequence $\alpha_1 \ldots \alpha_n \in T(G^d \cup L^d, G^p \cup L^p)$, we let

$$\text{next\_state}(\nu, p, \alpha_1\alpha_2 \ldots \alpha_n) = \text{next\_state}(\text{next\_state}(p, \nu, \alpha_1), p, \alpha_2 \ldots \alpha_n).$$

The *initial state* ν_0 of an implementation $S^{\mathcal{M}}$ must satisfy $\bigwedge_{1 \leq p \leq \mathcal{M}} \nu_0(p, \mathtt{mode}) = 0$. We assume that the system runs with *interleaving semantics* in the granularity of transitions, that is at any moment, at most one process can execute a transition. Execution of a transition is atomic.

A *computation* of an implementation $S^{\mathcal{M}}$ is a (finite or infinite) sequence $\rho = \nu_0\nu_1 \ldots \nu_k \ldots\ldots$ of states such that for all $k \geq 0$,

- ν_0 is the initial state of $S^{\mathcal{M}}$; and
- for each ν_k with $k > 0$, either $\nu_k = \nu_{k-1}$ or there is a $p \in \{1, \ldots, \mathcal{M}\}$ and a transition from q to q' such that $\nu_{k-1}(p, \tau(q, q')) = \mathit{true}$ and $\text{next\_state}(\nu_{k-1}, p, \pi(q, q')) = \nu_k$ is defined.

2.3 Safety analysis problem

To write a specification for the interaction among processes in a concurrent system, we need to define *global predicates* with the following syntax.

$$\begin{array}{rl} \phi ::= & \psi_1 \sim \psi_2 \mid \neg\phi \mid \phi_1 \vee \phi_2 \\ \psi ::= & c \mid \mathtt{NULL} \mid y \mid x[p] \mid z \rightarrow \epsilon \mid w[p] \rightarrow \epsilon \mid \psi_1 \oplus \psi_2 \end{array}$$

where $c \in \mathcal{N}$, $y \in G^d \cup G^p$, $x \in L^d \cup L^p$, $z \in G^p$, $w \in L^p$, and $1 \leq p \leq \mathcal{M}$.

Given a state ν and a global predicate ϕ, we define the valuation of ν on ϕ, written $\nu(\phi)$, in the following inductive way.

- $\nu(\psi_1 \sim \psi_2) \ = \ \nu(\psi_1) \sim \nu(\psi_2) \in \{\mathit{true}, \mathit{false}\}$
- $\nu(x[p]) = \nu(p, x)$
- $\nu(\neg\phi) = \neg\nu(\phi)$.
- $\nu(\phi_1 \vee \phi_2) = \nu(\phi_1) \vee \nu(\phi_2)$.

The rest is the same as the corresponding rules for local state predicates.

A computation $\rho = \nu_0\nu_1 \ldots \nu_k \ldots\ldots$ of $S^{\mathcal{M}}$ violates safety property ϕ iff there is a $k \geq 0$ such that either ν_k is undefined or $\nu_k(p,\phi) \neq \mathit{true}$ for some $1 \leq p \leq \mathcal{M}$. The *safety analysis problem* instance $\mathrm{SAP}(S,\mathcal{M},\phi)$ is to determine if for all computations ρ of $S^{\mathcal{M}}$ starting from some initial states, ρ does not violate safety property ϕ.

Example 2 *:* Consider the MCS locking algorithm in example 1. The critical section consists of modes 6 through 9. Thus the safety analysis problem for mutual exclusive access to the critical sections of two processes can be formulated as $\mathrm{SAP}(S, 2, \neg(6 \leq \mathtt{mode}[1] \leq 9 \wedge 6 \leq \mathtt{mode}[2] \leq 9))$. ||

3 Framework for safety analysis and reductions

We adopt the model-checking[6] technology for the verification of software implementations with pointer data structures. With such a technology, we are given a description of behaviors (typically as finite-state automata) and a specification of behaviors (typically in temporal logics). The goal is to explore and construct a representation of the reachable state-space and analyze if the automaton satisfies the specification. Our general algorithmic framework for symbolic safety analysis is shown as follows.

```
SAP($S,\mathcal{M},\phi$) {
  $reachable := \bigwedge_{1\leq p\leq\mathcal{M}}(\nu_0(p,\mathtt{mode})=0)$; /* the initial state-predicate */
  $next := true$;
  Loop until $next = false$, do {
    $next := false$;
    Sequentially for each $1 \leq p \leq \mathcal{M}$ and for each transition $(q,q')$, do {
      /* filter through triggering condition. */
      $new := \mathtt{indirect\_condition}(reachable, p, \tau(q,q'))$;                (1)
      /* symbolic execution. */
      $new := \mathtt{indirect\_assignment}(new, p, \pi(q,q'))$;                      (2)
      $new := \mathtt{reduce}(new)$; /* application of reduction techniques */        (3)
      $next := next \cup (new - reachable)$;
    }
    $reachable := reachable \cup next$;
  }
  if ($reachable \wedge \phi \neq false$) return "unsafe"; else return "safe";
}
```

The procedure iterates through the outer loop until *reachable* becomes a fixpoint. At line (1), $\mathtt{indirect\_condition}(D,p,\eta)$ returns a global predicate in BDD representing the subspace of D in which η is true of process p. At line (2), $\mathtt{indirect\_assignment}(D,p,\pi(q,q'))$ calculates a a global predicate in BDD representing the result after applying action sequence $\pi(q,q')$ to states in subspace represented by D. Symbolic implementations of procedure `indirect_condition()` and `indirect_assignment()` will be discussed in section 4. At line (3), reduce() is about application of various reduction techniques to control the complexity of reachable state-space representations.

At the first glance, model checking technology looks straightforward. The real challenge comes from the fact that in practice, the representation sizes of reachable state-spaces of any reasonably interesting software implementations are usually tremendous. In sections 5 and 6, we shall present two techniques to reduce the complexity of state space representations.

4 Symbolic computation of predicates with indirections

In our presentation of symbolica algorithms with BDD, we shall conveniently write Boolean combinations of BDDs, like $D_1 \vee D_2$, with the assumption that Boolean operations on BDDs are already defined. Details of such BDD operations can be found in [2, 5].

4.1 Symbolic evaluation of conditions with indirect operands

In a pointer data-structure system, users may write a predicate with operands of aribitrary indirections. For example, we may have a pointer data-structure system with the following declarations.

```
global pointer  L;
local pointer  parent, leftchild, rightchild;
local discrete  count: 0..5;
```

All these variables are to be encoded by finite number of bits in BDD-like data-structure. This is possible because their value ranges are finite. Specifically, $\texttt{lb}(\mathit{count}) = 0$, and $\texttt{ub}(\mathit{count}) = 5$.

When we are given a state-space representation D in BDD-like data-structure, how can we compute the maximal subspace representation D', of D, in which a complicate condition η with indirections like

$$\texttt{parent}[1] \rightarrow \texttt{count} - 2 * \texttt{leftchild}[2] \rightarrow \texttt{rightchild} \rightarrow \texttt{count} < L \rightarrow \texttt{count}$$

is true. The condition says that difference of the count of parent of the 1st process ($\texttt{parent}[1] \rightarrow \texttt{count}$) and twice the count of the right child of the left child of the 2nd process ($2 * \texttt{leftchild}[2] \rightarrow \texttt{rightchild} \rightarrow \texttt{count}$) is less than the count of process L ($L \rightarrow \texttt{count}$). Since there is no restriction on lengths of indirections, we need a flexible algorithm to construct such D'. Our algorithm is simplified for presentation and explanation as the following function `indirect_condition()`, which in turns invokes functions `indirect_ref()`, `indirect_arith()`, and `indirect_effect()`.

```
indirect_condition($D, p, \eta$) {
  Collect the operands $\omega_0, \ldots, \omega_n$ used in $\eta$;
  Rewrite $\eta$ into $\eta'$ in the form of $\omega_0 \sim \epsilon$.
  Construct $D_\epsilon := D \wedge \texttt{indirect\_arith}(\epsilon, p)$;
  if $\omega_0$ is $h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k \rightarrow x$ with $k > 0$, then {
    Let $R := \mathit{false}$;
    Construct $D_{\omega_i} := D_\epsilon \wedge \texttt{indirect\_ref}(h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k, p)$;
    for $j := 1$ to $\mathcal{M}$, do {
      Let $H := \texttt{var\_eliminate}(D_{\omega_i} \wedge (\texttt{PI} = j), \texttt{PI})$;
      Let $H := \texttt{condition\_effect}(x[j], \sim, H)$;
      Let $R := R \vee H$;
    }
  }
  else if $\omega_0$ is $x[i]$ with local variable $x$ with specific process reference $i$, then
    Let $R := \texttt{condition\_effect}(x[i], \sim, D_\epsilon)$;
  else if $\omega_0$ is $x$ with local variable $x$ with no specific process reference, then
    Let $R := \texttt{condition\_effect}(x[p], \sim, D_\epsilon)$;
  else if $\omega_0$ is a global variable $x$, then
    Let $R := \texttt{condition\_effect}(x, \sim, D_\epsilon)$;
  return $R$;
}
```

Procedure $\texttt{var\_eliminate}(D, x)$ filters x out of D. For a local discrete variable x, $\texttt{var\_eliminate}(D, x) = \bigvee_{v \in [lb(x), ub(x)]} D|_{x=v}$ where $D|_{x=v}$ is the new local state predicate obtained by instantiating x to v. For a local pointer x, $\texttt{var\_eliminate}(D, x) = \bigvee_{v \in \{\text{NULL}, 1, \ldots, \mathcal{M}\}} D|_{x=v}$.

The presentation is simplified in that when it invokes `indirect_arith()`, we assume that we don't have to worry about problems like divide-by-zero and imprecision caused by integer division. In our real implementation, the algorithm is more involved and iteratively solves the linear inequality constraints with respect to operand ω_0. In the iterations to solve the inequality constraints, such problems are properly taken care of with case-analysis. Due to page-limit, we only use the simplified presentations of algorithms in the following. The algorithm uses two auxiliary variables, `VALUE` and `PI`. `VALUE` is used to hold the value of ann arithmetic expression. `PI` is used to hold the destination process identifier of an indirection of arbitrary length.

Function $\texttt{indirect\_ref}(h_i \rightarrow l_{i+1} \rightarrow \ldots \rightarrow l_k, p)$ constructs the necessary condition at a state ν when $\nu(h_i \rightarrow l_{i+1} \rightarrow \ldots \rightarrow l_k, p)$ is identical to the process identifier recorded in variable `PI`.

```
indirect_ref($h_i \rightarrow l_{i+1} \rightarrow \ldots \rightarrow l_k, p$) {
  if $i \geq k$, return($\texttt{PI} = p$);
  Let $R := \mathit{false}$;
  if $h_i$ is a local pointer $l_i[j]$ with specific process reference $j$, then
    for $f := 1$ to $\mathcal{M}$, do
```

```
            $R := R \vee (l_i[j] = f \wedge$ indirect_ref($l_{i+1} \rightarrow \ldots \rightarrow l_k, f$));
    else if $h_i$ is a local pointer $l_i$ with no specific process reference, then
        for $f := 1$ to $\mathcal{M}$, do
            $R := R \vee (l_i[p] = f \wedge$ indirect_ref($l_{i+1} \rightarrow \ldots \rightarrow l_k, f$));
    else if $h_i$ is a global pointer $g_i$, then
        for $f := 1$ to $\mathcal{M}$, do $R := R \vee (g_i = f \wedge$ indirect_ref($l_{i+1} \rightarrow \ldots \rightarrow l_k, f$));
    return($R$);
}
```

Function `indirect_arith`(ϵ, p) uses the auxiliary variable `VALUE` to symbolically record the value of expression ϵ at process p.

```
indirect_arith($\epsilon, p$) {
    $R := false$;
    if $\epsilon$ is $h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k \rightarrow x$ with $k > 0$, then {
        $H :=$ indirect_ref($h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k, p$);
        for $j := 1$ to $\mathcal{M}$, lb($x$) $\leq v \leq$ ub($x$), do
            $R := R \vee$ (var_eliminate($H \wedge$ PI $= j$, PI) $\wedge x[j] = v \wedge$ VALUE $= v$);
    }
    else if $\epsilon$ is $x[i]$ with local variable $x$ and specific process reference $i$, then
        for lb($x$) $\leq v \leq$ ub($x$), do $R := R \vee (x[i] = v \wedge$ VALUE $= v$);
    else if $\epsilon$ is local variable $x$ with no specific process reference, then
        for lb($x$) $\leq v \leq$ ub($x$), do $R := R \vee (x[p] = v \wedge$ VALUE $= v$);
    else if $\epsilon$ is a global variable $x$, then
        for lb($x$) $\leq v \leq$ ub($x$), do $R := R \vee (x = v \wedge$ VALUE $= v$);
    else if $\epsilon$ is $\epsilon_1 \oplus \epsilon_2$ where $\oplus \in \{+, -, *, /\}$, then {
        $R_1 :=$ indirect_arith($\epsilon_1, p$);
        $R_2 :=$ indirect_arith($\epsilon_2, p$);
        for every possible value $v_1, v_2$ of variable VALUE, do {
            $H_1 :=$ var_eliminate($R_1 \wedge$ VALUE $= v_1$, VALUE);
            $H_2 :=$ var_eliminate($R_2 \wedge$ VALUE $= v_2$, VALUE);
            $R := R \vee (H_1 \wedge H_2 \wedge$ VALUE $= v_1 \oplus v_2$);                  (1)
        }
    }
    return $R$;
}
```

To evaluate an expression like $\epsilon_1 \oplus \epsilon_2$, the values recorded in the `VALUE` variable respectively in the symbolic predicates of ϵ_1 and ϵ_2 are used as in line (1) in procedure `indirect_arith`().

```
condition_effect($x, \sim, D$) {
    $R := false$;
    for every possible value $v$ of variable VALUE, do
        $R := R \vee (x \sim v \wedge$ var_eliminate($D \wedge$ VALUE $= v$, VALUE));
    return $R \wedge$ lb($x$) $\leq x \leq$ ub($x$);
}
```

4.2 Symbolic manipulation of assignments with indirect operands

Given a state-space predicate D and an assignment statement like $\omega_0 := \epsilon;$, the symbolic postcondition by process p in traditional wisdom is

$$\texttt{indirect\_condition}(\texttt{var\_eliminate}(D, \omega_0), p, \omega_0 = \epsilon;)$$

But this fails in two ways. First, there can be indirections in ω_0. Second, the destination of ω_0 can occur in ϵ in a recurrence assignment. In fact, such recurrence assignment is really very common and indispensible in practice.

Our algorithm is given as follows.

```
indirect_assignment($D, p, \omega_0 := \epsilon;$) {
    Construct $D_\epsilon := D \wedge$ indirect_arith($\epsilon, p$);
    if $\omega_0$ is $h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k \rightarrow x$ with $k > 0$, then {
```

```
        Let $R := false$;
        Construct $D_{\omega_i} := D_\epsilon \wedge$ indirect_ref($h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k, p$);
        for $j := 1$ to $\mathcal{M}$, do {
            Let $H :=$ var_eliminate(var_eliminate($D_{\omega_i} \wedge$ (PI $= j$), PI), $x[j]$);        (2)
            Let $H :=$ condition_effect($x[j]$, $\sim$, var_eliminate($H, x[j]$));
            Let $R := R \vee H$;
        }
    }
    else if $\omega_0$ is $x[i]$ with local variable $x$ with specific process reference $i$, then
        Let $R :=$ condition_effect($x[i]$, $\sim$, var_eliminate($D_\epsilon, x[i]$));        (3)
    else if $\omega_0$ is local variable $x$ with no specific process reference, then
        Let $R :=$ condition_effect($x[p]$, $\sim$, var_eliminate($D_\epsilon, x[p]$));        (4)
    else if $\omega_0$ is a global variable $x$, then
        Let $R :=$ condition_effect($x$, $\sim$, var_eliminate($D_\epsilon, x$));        (5)
    return $R$;
}
```

In this algorithm, the problem with the recurrence assignment are solved since we use variable `VALUE` as a temporary recorder for the expression value and the destination variable are eliminated from the symbolic predicate at line (2), (3), (4), and (5) with procedure `var_eliminate()` before being assigned by procedure `condition_effect()`.

In our implementation, performance tuning has also been made to efficiently manipulate non-recurrence assignments.

5 Reduction by inactive local variable elimination

The idea is that from some states, some variables will not be used until they are written again. Such variables are called *inactive* in such states and their values can be forgotten without affecting the behavior of the software implementation. Such a technique has been used heavily in tools like Spin[11, 12], UPPAAL[4], SGM[14, 26], and `red`[22, 23, 24, 25]. But for systems with pointers, it is important to note that pointers used for indirect referencing are also implicitly read in the execution of the corresponding action. With this caution in mind, we develop a fixed-point procedure to derive an upper approximation local state predicate that describes the states in which a local variable is active. Once we find that a variable is inactive in all states described by a BDD, we can

- replace the values of those inactive local discrete variables in a state with zeros; and
- replace the values of those inactive local pointers in a state with nulls;

With such replacements, we expect to greatly cut the complexity of our reachable state space representations.

However, it can be difficult to determine the exact description of a state set in which a local variable is inactive. In fact, we shall aim at constructing a local state predicate for an upper approximation of the active condition. Given a local discrete variable x, the local state predicate will be in $B(G^d \cup L^d - \{x\}, G^p \cup L^p)$. For a local pointer x, it will be in $B(G^d \cup L^d, G^p \cup L^p - \{x\})$. That is, the upper approximation is described in terms of the variables, except $x[p]$, directly observable by the local process p. Then a lower approximation of the corresponding inactive condition of $x[p]$ is obtained by negating the just-obtained upper approximation of the active condition.

A local variable $x[p]$ is *possibly read* by process p in assignment $\omega_0 := \epsilon$; iff either

- an indirect reference like $y_1 \rightarrow \ldots \rightarrow y_m \rightarrow y$ occurs in ω_0 and $x[p] \in \{y_1[p], \ldots, y_m[p]\}$; or
- an indirect reference like $y_1 \rightarrow \ldots \rightarrow y_m$ occurs in ϵ and $x[p] \in \{y_1[p], \ldots, y_m[p]\}$.

Given a local variable $x[p]$, an upper approximation of its active condition is constructed in two steps. First, we construct a base approximation from the triggering conditions and actions of all transitions in the algorithm as follows.

```
base_uapprox_active($x$)
    let $\eta_x := false$;
    for each transition $(q, q')$ in $A(P)$,
        if $x$ is possibly read in actions in $\pi(q, q')$,
            then $\eta_x := \eta_x \vee$ (mode $= q \wedge$ var_eliminate($\tau(q, q'), x$)).
        else {
            break $\tau(q, q')$ into DNF $\Delta_1 \vee \Delta_2 \vee \ldots \vee \Delta_k$;
            for each $\Delta_i$, if $x$ appears in $\Delta_i$,
```

```
            then $\eta_x := \eta_x \vee (\texttt{mode} = q \wedge \texttt{var\_eliminate}(\Delta_i, x))$.
        }
    }
    return $\eta_x$;
}
```

In the second step, from the base approximation, we calculate an upper approximation of the backward weakest precondition through each transition until a least fixpoint is reached. This is done in the following steps.

```
uapprox_active($x$) {
    $\eta_x := \texttt{base\_uapprox\_active}(x); \eta'_x := \textit{false}$;
    Repeat until $\eta'_x = \eta_x$ {
        $\eta'_x := \eta_x$;
        sequentially for each transition $(q, q')$ in $A(P)$, {
            let $\delta$ be an upper approximation of the weakest precondition
                of $\eta_x$ before applying $(q, q')$.
            let $\eta'_x := \eta'_x \vee \delta$;
        }
    }
    return $\eta_x$ as active$_x$.
}
```

It can be proven that the upper approximation local state predicate is indeed independent of x. For example, by applying the above-mentioned procedure to the MCS algorithm in figure 1, we find that

$$\begin{aligned}
\text{active}_{\texttt{locked}} &= 4 \leq \texttt{mode} \leq 5 \\
\text{active}_{\texttt{next}} &= \texttt{mode} = 1 \vee (2 \leq \texttt{mode} \leq 4 \wedge \texttt{prev} \neq P) \vee \texttt{mode} \geq 5 \\
\text{active}_{\texttt{prev}} &= 1 \leq \texttt{mode} \leq 4
\end{aligned}$$

It shows that local variable `locked`, for example, will not be read and thus affect the system behaviors outside local modes 4 and 5. The elimination of values of `locked` when it becomes inactive makes the state-space representation more concise and compact.

6 Theoretical foundation of symmetry reduction

6.1 Basics

Symmetrically structured systems behave symmetrically. That is, if a transition can transform a state s into a state s' then symmetric counterparts of that transition will transform a symmetric counterpart of s into a symmetric counterpart of s'.

Formally, a symmetry σ in our context is a bijection (i.e. permutation) on the process identifiers $\sigma : [1, \mathcal{M}] \overset{1:1}{\rightarrow} [1, \mathcal{M}]$. For convenience, we may represent a bijection σ as a sequence $(i_1, \ldots, i_{\mathcal{M}})$ such that $\{i_1, \ldots, i_{\mathcal{M}}\} = \{1, \ldots, \mathcal{M}\}$. Such a bijection σ defines a transformation $\overline{\sigma}$ on state ν by

- for $p = \texttt{NULL}$, $\overline{\sigma}\nu(p, x) = \perp$;
- for $x \in G^d$ and $p \in [1, \mathcal{M}]$, $\overline{\sigma}\nu(p, x) = \nu(p, x)$;
- for $x \in G^p$ and $p \in [1, \mathcal{M}]$, $\overline{\sigma}\nu(p, x) = \sigma(\nu(p, x))$;
- for $x \in L^d$ and $p \in [1, \mathcal{M}]$, $\overline{\sigma}\nu(\sigma(p), x) = \nu(p, x)$;
- for $x \in L^p$ and $p \in [1, \mathcal{M}]$, $\overline{\sigma}\nu(\sigma(p), x) = \sigma(\nu(p, x))$.

(Assume $\sigma(\texttt{NULL}) = \texttt{NULL}$, and $\sigma(\perp) = \perp$). That is, values of local discrete variables of process p in state ν become values of local variables in $\sigma(p)$ in state $\overline{\sigma}\nu$, global pointers pointing to p in ν will point to $\sigma(p)$ in $\overline{\sigma}\nu$, and if p has a local variable that points to p' in ν, the same variable in $\sigma(p)$ will point to $\sigma(p')$ in $\overline{\sigma}\nu$. Global discrete variables, `NULL`s and $\perp$'s remain unchanged by $\overline{\sigma}$.

Since all processes run identical programs, the only way to break symmetry is to have process identifier constants occur in the program. Other than this, programs are insensitive to arbitrary bijection in the following sense:

THEOREM 1 *Let σ be a bijection on the process identifiers such that $\sigma(p) = p$ for all process identifiers p that occur explicitly in expressions of the program, Let α be an action that is enabled in process p and state ν. Then α is enabled in process $\sigma(p)$ and state $\sigma(\nu)$ as well, and it holds next_state$(\sigma(p), \sigma(\nu), \alpha) = \overline{\sigma}$next_state$(p, \nu, \alpha)$.* ||

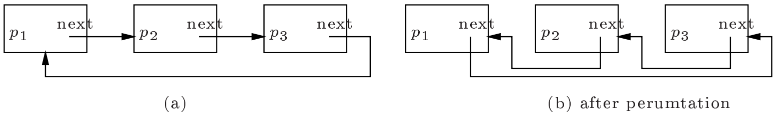

Figure 2: Anomaly of image false reachability

This result is an immediate consequence of the following lemma. Remember that we assume the operation mode to be a local discrete process variable.

LEMMA 2 *Let E be an expression of discrete type, F an expression of pointer type, and B be a boolean predicate. Let σ be a bijection on the process identifiers such that $\sigma(p) = p$ for all process identifier constants that appear in any of E, F, or B. Let $\nu(p, E)$ ($\nu(p, F)$, $\nu(p, B)$) be the value that E (F, B) evaluates to in ν when local variables take values from p. Then the following equations hold:*

$$\begin{array}{rcl} \nu(p, E) & = & \overline{\sigma}\nu(\sigma(p), E) \\ \nu(p, B) & = & \overline{\sigma}\nu(\sigma(p), B) \\ \sigma(\nu(p, F)) & = & \overline{\sigma}\nu(\sigma(p), F) \end{array}$$

Proof : Omitted due to page-limit. ||

From lemma 2, we can deduce

COROLLARY 3 *If σ is a bijection such that $\sigma(p) = p$ for all process identifiers that are explicitly mentioned in the program, then a state ν' is reachable from a state ν if and only if $\overline{\sigma}\nu'$ is reachable from $\overline{\sigma}\nu$.* ||

The idea of symmetry reduction [15, 21, 7, 9, 14, 26, 18, 22] is to keep only one state if two states turn out to be symmetric.

Let Σ be a group of bijections, i.e. a set of bijections closed under composition and inversion. Then the relation $\equiv_\Sigma$ on states, defined by $\nu \equiv_\Sigma \nu'$, iff $\exists \sigma . \sigma \in \Sigma$ and $\overline{\sigma}\nu = \nu'$ is an equivalence relation. A reduced transition system with respect to Σ would contain, consequently, one member of each equivalence class of states. With Cor. 3, this reduced system covers all reachable states in the sense that every reachable state is equivalent to one that is part of the reduced transition system.

Obviously, the larger Σ is, the more states become equivalent, and the smaller the reduced transition system becomes compared with the original one. On the other hand, we do not want to spend too much time for searching a set Σ that satisfies the requirements of Cor. 4. This space/time tradeoff can be solved in various ways. We propose some solutions. However there are at least two challenges in designing an efficient and yet correct symmetry-reduction procedure:

correctness: In defining the equivalence classes, we have to take reachability issues into consideration. For example, we may have $\mathcal{M} = 3$, such that the local pointers *next* of the processes initially form the following static clockwise cycle in figure 2(a). If we choose to use the image cycle after bijection $\sigma = (132)$ as the representative, then the representative state in the equivalence class will be the counter-clockwise cycle shown in figure 2(b). But the problem is that the chosen counter-clockwise cycle image may never be reachable from the initial state since the cycle is a static one. We call this problem the *anomaly of image false reachability*. Thus when the goal state is specified as $next[p_1] = p_2$, the reachability analysis procedure equipped with such a problematic symmetry-reduction may actually give a wrong answer to the query.

complexity: The pointers of processes can be used to construct various graph configurations. Symmetry-reduction, in other words, is try to partition set of graphs into isomorphic classes. But graph isomorphism has not been known to be a PTIME problem. Thus, to compute the unique representatives of different classes can be very costly.

In the following subsections, we first discuss how to transposition (binary bijection) of process identifiers to efficiently compute equivalent state images in a symmetric subclass; and then we discuss the complexity issues in the general class.

6.2 Efficient symmetric image computation in symmetric class

For convenience, we use $\sigma_{p_ip_j}$ to represent the *transposition (binary bijection)* which only switches the position of p_i and p_j. The following corollary helps identify the "symmetry" in program, initial state predicate, and goal state predicate.

COROLLARY 4 *Assume a reduced transition system that contains one member per equivalence class of states w.r.t. the group of bijections Σ. If all $\sigma \in \Sigma$ satisfy*

- *$\sigma(p) = p$ on all process identifier p that are mentioned anywhere in the program, and*
- *if ν is an initial state then so is $\sigma(\nu)$,*

then the set of all states that are equivalent to states in the reduced transition system is exactly the set of reachable states of the transition system. If Σ satisfies additionally that for all $\sigma \in \Sigma$ and all states ν, ν satisfies the goal condition iff $\overline{\sigma}\nu$ does, then the reduced set of states intersects with the goal condition if and only if the original transition system does. ||

That is, under these conditions we can replace the original transition system with the reduced one for solving a safety analysis problem. Observe that the result of Lemma 2 holds not only for expressions occurring in the program, but also for predicates used for initial and goal conditions (in these predicates, expression of the form $x[p]$ replace local variables but behave identically w.r.t. the calculations in the proof of Lemma 2). Therefore,

THEOREM 5 *If Σ is a set of bijections where all $\sigma \in \Sigma$ satisfy $\sigma(p) = p$ for all process identifiers that occur anywhere in the program, in the initial condition, or in the goal condition, then Σ meets all conditions required in Cor. 4.* ||

In particular, this set is closed under composition and inversion. Moreover, such a group Σ has a well-structured generating set—the set of all transpositions $\sigma_{p_ip_j}$ where $p_i \neq p_j$, neither p_i nor p_j are among the "forbidden" process identifiers, $\sigma_{p_ip_j}(p_i) = p_j$, $\sigma_{p_ip_j}(p_j) = p_i$, and $\sigma_{p_ip_j}(p_k) = p_k$ for all other p_k. This means that every member of this Σ can be represented as a sequence of exchanging two process identifiers. Using this fact, a state can be stepwise transformed by applying transpositions until some kind of "lexicographically" smallest state is achieved. Since this process resembles usual sorting procedures, it yields a unique, smallest member of the equivalence class of the original state in polynomial time. Thus, this procedure can be used to efficiently solve the problem of how to construct the reduced transition system.

The just-mentioned method for detecting symmetry can be efficiently performed by examining the syntax of the program, initial state predicate, and goal state predicate. There is a way to find a larger Σ also having transpositions $\sigma_{p_ip_j}$ as generating set, but covering cases where p_i, p_j, or both do appear in the formula. We can construct, for some process p_i and another process p_j, a BDD of the initial condition two times—where the second BDD has the variables corresponding to process p_i change places with the variables corresponding to p_j. We can use the uniqueness of optimal BDD to check whether this exchange of roles between p_i and p_j leads to the same initial condition, leads to the same initial condition. If this is the case then $\sigma_{p_ip_j}$ leaves the initial condition invariant.

6.3 Symmetry detection with asymmetric systems or asymmetric predicates

As argued in figure 2, there may be initial predicates that are invariant w.r.t. to some symmetries, but not with respect to any transposition σ_{ij}. Such (possibly not all) symmetries of this kind can be found using the concept of graph automorphisms. The idea is to consider the syntax tree of the formula representing the initial predicate. If we can permute subformulas of commutative and associate operators ($\wedge$, $\vee$), or terms of symmetric expressions ($=, \neq$) such that the permuted formula looks identical to the original formula with the exception that some of the process identifiers have been permuted, then the corresponding bijection will be a symmetry leaving the initial condition invariant (since such bijection does not change the value of the formula, the value for an original state and a permuted state will be the same. Formally, graph automorphisms are defined on labeled graphs. A labeled graph [V,E,L] consists of a set V of vertices, a set $E \subseteq V \times V$ of edges, and a labeling function $L : V \cup E \to \mathcal{D}$ that assigns an element of some domain $\mathcal{D}$ to each vertice and each edge. A graph automorphism is a bijection σ on V such that for all $v_1, v_2 \in V$, $L(v_1) = L(\sigma(v_1))$, $[v_1, v_2] \in E$ iff $[\sigma(v_1), \sigma(v_2)] \in E$, and $L([v_1, v_2]) = L([\sigma(v_1), \sigma(v_2)])$, i.e., σ preserves the labeling and the edge relation on $[V, E]$. In a graph constructed out of a formula, we would chose $D = \mathcal{N} \cup G^p \cup G^d \cup L^p \cup L^d \cup \{\vee, \neg, =, \neq, <, >, \leq, \geq, \mathtt{NULL}\}$ and construct a graph as follows:

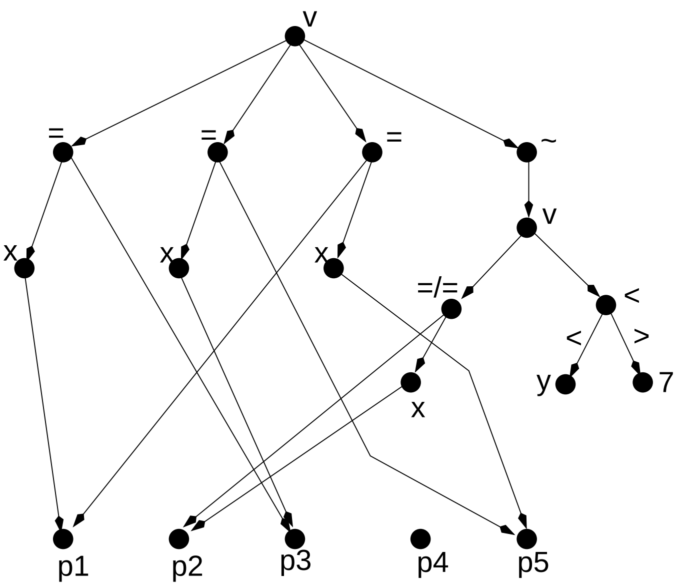

Figure 3: Labeled graph constructed for the formula $x[1] = 3 \vee x[3] = 5 \vee x[5] = 1 \vee \neg(x[2] \neq 2 \vee y < 7)$ and $\mathcal{M} = 5$

- include, for each $p \in [1, \mathcal{M}]$, a vertice p labeled *empty*;
- For $\phi = \phi_1 \vee \phi_2 \vee \dots \phi_n$, construct graphs for the ϕ_i, and add a vertice labeled with $\vee$ and edges labeled with `NULL` pointing to the roots of the constructed subtrees; the new node becomes the root;
- for $\phi = \neg\phi_1$, construct a tree graph ϕ_1 and add a vertice labeled with $\neg$ and an edge labeled with `NULL` to the root of the subtree; the new node becomes the root;
- for $\phi = T_1 = T_2$, (or $\phi = T_1 \neq T_2$), construct graphs for T_1 and T_2, and include a new root node labeled with $=$ (or $\neq$, resp.), and edges labeled `NULL` to the roots of the two subtrees;
- for $\phi = T_1 < T_2$, (or $\phi = T_1 \leq /geq/ > T_2$), construct graphs for T_1 and T_2, and include a new root node labeled with $<$ (or $\leq, \geq, >$, resp.), an edge labeled $<$ to the root of T_1, and an edge labeled $>$ to the root of T_2— the different labeling of the two edges assures that left and right subexpression of asymmetric operations are not exchanged;
- for an expression $T =$ `NULL`, insert a new root vertice labeled `NULL`;
- for an expression $T = c$ ($c \in calN$, build a new root vertice labeled c;
- for an expression $T = y$ (for global variable $y \in G^p \cup G^d$), add a new root vertice labeled y;
- for an expression $T = x[p]$ (for a local variable $x \in L^d \cup L^p$ and a process identifier p), add a new root node labeled x and an edge labeled `NULL` to the unique vertice labeled p;

Notice that the process identifiers are nodes, not labels of nodes. Thus, we may permute them.

Figure 3 depicts a labeled graph of a formula.

It is immediately clear that a graph automorphism replaces a formula's syntax tree into an identical tree, with just the process identifier vertices permuted. Since a graph automorphism respects the edge relation, permuting inner nodes of the graph corresponds to changing the order of subformulas of symmetric operations. Therefore, for a projection of a graph automorphism to the process identifier vertices, we obtain a symmetry that leaves the initial (or goal) predicate invariant.

Graph automorphism groups can have exponential size (in the size of the graph), but there is always a generating set of at most $\frac{|V|(|V|-1)}{2}$ elements. For the computation of the generating set, no polynomial time algorithm is known. In fact, the problem is closely related to the graph isomorphism problem which is widely believed to be a possible candidate for a problem that lies properly between the complexity classes P and $NP-complete$. However, existing tools computing graph automorphisms (for instance, [17, 20]) show that a computation is possible in reasonable time for fairly large graphs (thousands of nodes) with practical background.

Since a set of symmetries arising from graph automorphism does not necessarily contain transpositions, we need other ways to find out whether states in the transition system are equivalent. This problem has been studied in [19] in the context of Petri nets; the results of that paper are without major efforts applicable to the situation of pointer structures.

Beyond the graph automorphism solution, there may be still more symmetries taking semantic equivalences of syntactically asymmetric formulas into consideration. We are not going into details in this respect.

So far, the second of the proposed solutions using BDD equivalence has been implemented and is used for the experimental results. The algorithmically much more involved graph automorphism solution could be implemented if further studies suggest that a further condensation of the state space is worth the computational complex calculations of the automorphism approach.

7 Implementation and experiments

We implemented our techniques in a symbolic verification tool called `red` (Region-Encoding Diagram) [22, 23] which is available for free at

`http://www.iis.sinica.edu.tw/~farn/red/`

`red` supports verification of timed automata [1] with a new BDD-like data structure [2]. The reduction by inactive variable elimination is automatic.

7.1 IbSINC reduction: Incomplete but Sound Isomorphism of Network Configuraitons

Remember that we represent the process identifiers as integers 1 to $\mathcal{M}$. We shall define a total-ordering among processes in a state. If in a state ν, process i precedes process j in the total-ordering, we then write $\nu \models i \prec j$. It is our goal that for each symmetry-equivalence class of states, we shall only keep the state ν in which $\nu \models 1 \prec 2 \prec \ldots \prec \mathcal{M}$. We shall define condition $\texttt{reverse}(i,j)$, $1 \leq i < j \leq \mathcal{M}$, such that for all state-predicates η, $\eta \wedge \texttt{reverse}(i,j)$ is true if for all state ν in the space reprsented by η, $\nu \not\models i \prec j$, which implies processes i, j should be permuted according to our heuristics. Then given a symbolic state-space representation η, the following procedure reorders the process identifiers to cut down the number of equivalent state subspaces.

```
reduce_symmetry($\eta$) {
    Sequentially for $i := 1$ to $\mathcal{M} - 1$, do
        Sequentially for $j := i + 1$ to $\mathcal{M}$,
            let $\eta := (\eta - \texttt{reverse}(i,j)) \vee \texttt{permute}(\eta \wedge \texttt{reverse}(i,j), i, j)$;
    return $\eta$;
}
```

Here $\texttt{permute}(\eta, i, j)$ is obtained from η by

- switching the values of $x[i]$ and $x[j]$ for every local variable x; and
- changes the value i to j, or vice versa, of all pointer variables.

$\texttt{permute}(\eta, i, j)$ is actually a binary transposition on process i and j.

In the following, we implement our transposition heuristics in constructing the predicate of $\texttt{reverse}(i,j)$. In practice, it will be translated into BDD-like data structures for symbolic manipulations in calculating the fixpoint. Suppose the global pointers are ordered as $g_1, g_2, \ldots, g_{|G^p|}$ while the local pointers are ordered as $l_1, l_2, \ldots, l_{|L^p|}$ in the declaration. $\texttt{reverse}(i,j)$, with $i < j$, reasons as follows.

```
$\texttt{reverse}(i,j)$ {
    if /* global pointers point to $j$ before pointing to $i$ in syntax order */
        $\exists 1 \leq a \leq |G^p| (g_a = j \wedge \forall 1 \leq h < a (g_h \neq i))$, return $\mathit{true}$;
    else if $g_a < i$ in the state for all $1 \leq a \leq |G^p|$, then {
        if /* local pointers of earlier processes point to $j$
              * before pointing to $i$ in syntax order. */
            there is a $1 \leq h < i$ and a $1 \leq b \leq |L^p|$ such that in the state,
            • for all $1 \leq h' < h$ and for all $1 \leq b' \leq |L^p|$, $l_{b'}[h'] \neq i$; and
```

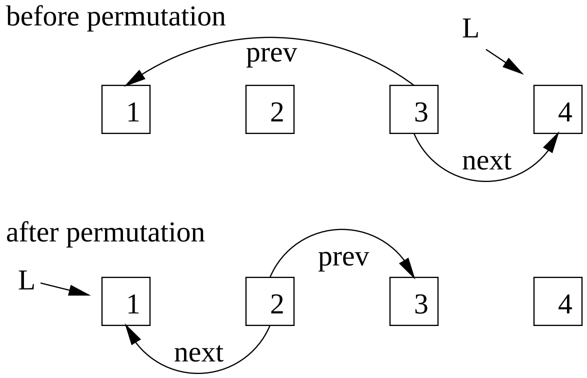

Figure 4: Transposition of process identifiers

```
        • for all $1 \le b' < b$, $l_{b'}[h] \neq i$;
        • $l_b[h] = j$,
        then return true;
    else if /* process $j$'s local pointers pointing to earlier processes before $i$'s
               * local pointers do so in syntax order. */
        there is a $1 \le b \le |L^p|$ such that
        • for all $1 \le b' < b$, $l_{b'}[i] = l_{b'}[j] \vee (l_{b'}[i] > i \wedge l_{b'}[j] > i)$, and
        • $l_{b'}[j] \le i \wedge l_{b'}[j] < l_{b'}[i]$,
        then return true;
  }
  return false;
}
```

In figure 4, we have drawn the four process network constructed by the MCS algorithm respectively before and after our transposition in figure 1. After the transposition, the network nodes are reordered in a linear sequence according to the queue formation.

7.2 Experiments

The IbSINC reduction in tool `red` is invoked by option "Sp." We compared the performance of `red`, both with and without the IbSINC reduction technique, with that of SMC[10]. SMC was developed with the theoretical framework in [9]. Seven different options of symmetry reduction with various precisions are supported in SMC.

We have tested our implementation with three benchmarks. The first is the MCS locking algorithm whose implementation of two processes is in appendix A. The safety condition to check is that no two processes are in the critical section at the same time, that is

$$\neg \bigvee_{1\le i<j\le m}(6 \le \texttt{mode}[i] \le 9 \wedge 6 \le \texttt{mode}[j] \le 9)$$

The second benchmark is a dynamic double-link cycle insertion and deletion algorithm. We have a set of symmetric processes which insert itself in and delete itself from the cycle maintained by two local pointers: `next` and `prev`. We also have a global pointer `L` pointing to the tail of the cycle. If there is no process in the cycle, then `L` = `NULL`. The safety condition to check is that when a process thinks it itself is in the cycle, global pointer `L` ≠ `NULL`, i.e. there is no cycle at all. Formally, the safety condition is

$$(\exists i, \texttt{next}[i] \neq \texttt{NULL}) \Rightarrow \texttt{L} \neq \texttt{NULL}$$

The third benchmark is a leader-election alogorithm. Each process has a local pointer `parent`. A set of symmetric processes randomly request to be a child of another process, who is not yet somebody's child. The process responds

benchmarks	Tools	Options ?	3	4	5	6	7	8	9
MCS	red	-fSp	0.98s	5.25s	27.09s	141.56s	928.26s	8584.44s	115072.26s
			48k	131k	397k	1064k	3299k	22824k	197599k
		-f	2.33s	23.13s	291.21s	5442.33s	Not available		
			139k	1125k	12776k	234497k			
	SMC	-s1	145.0 s core dumped	Not available					
		-s2	596.4s	>17 hours not finished	Not available				
			13472k						
		-s3	600.3s	>17 hours not finished	Not available				
			13477k						
		-s4	1601.8s	20252.6s core dumped	Not available				
			13460k						
		-s5	1624.0s	>17 hours not finished	Not available				
			13457k						
		-s6	1600.3s	>17 hours not finished	Not available				
			13459k						
		-s7	1620.8s	>17 hours not finished	Not available				
			13457k						
leader election	red	-fSp	0.02s	0.08s	0.23s	0.59s	1.55s	4.43s	15.12s
			17k	38k	69k	113k	171k	387k	1012k
		-f	0.02s	0.05s	0.10s	0.16s	0.25s	0.35s	0.54s
			17k	38k	69k	113k	171k	246k	337k
	SMC	-s1	0.3s	0.5s	0.3s	0.7s	4.8s	62.7s	2096.4s
			1k	7k	34k	193k	1224k	8444k	68240k
		-s2	0.2s	0.2s	0.4s	2.4s	42.7s	1097.2s	34511.2s
			1k	7k	29k	135k	681	3707k	21799k
		-s3	0.2s	0.2s	0.4s	2.3s	29.5s	944.1s	16335.1s
			1k	7k	28k	134k	567	3451k	14964k
		-s4	0.2s	0.4s	0.4s	1.4s	9.2s	73.7s	619.0s
			1k	6k	19k	62k	196	604k	1857
		-s5	0.3s	0.3s	0.4s	1.3s	8.9s	70.4s	591.0s
			1k	6k	19k	62k	195	602k	1851
		-s6	0.3s	0.3s	0.5s	1.4s	9.1s	73.3s	621.0s
			1k	6k	19k	62k	196	604k	1857
		-s7	0.2s	0.3s	0.4s	1.3s	8.8s	70.7s	592.6s
			1k	6k	19k	62k	195	602k	1851
double cycle insertion	red	-fSp	0.06s	0.26s	0.95s	3.59s	14.86s	59.65s	228.72s
			16k	38k	98k	254k	618k	1462k	3418k
		-f	0.93s	67.89s	>15 mins not finished	Not available			
			165k	10073k					
	SMC	No termination in 1 hour							

s: CPU time in seconds; k: Memory in kilobytes;

Table 1: Performance data table of three benchmarks

to a request will write its identifier to a global pointer `respond_id`. Then the requesting process will write the content of `respond_id` to its local variable `parent`. The processes use their variables `parent` to construct a dynamic forest structure. We want to make sure that at any time, at least some process is not somebody's child. Formally speaking, that is

$$\exists i, \texttt{parent}[i] = \texttt{NULL}$$

The performance data table is in table 1. The first row of columns 3 to 11 are for the numbers of processes in the implementation. All the data is collected on a Sun Sparc station with dual 450MHz processors and 4 GB memory running Solaris. All data of `red` are collected with forward analysis (option -f). In each entry of the rows, the CPU times (in seconds) and memory consumptions (in kilobytes) are shown. The memory complexity for `red` is collected only for data structures, which includes the `red` nodes and the 2-3trees used to manage the `red` nodes.

The three benchmarks represent three different types of dynamic data-structures: doubly-linked queues, doubly-linked cycles, and forests with arbitrary number of children of each internal nodes. According to the performance table, we found that our techniques indeed greatly reduced time and memory complexity and had shown promise to perform well with data-structure diversity.

8 Conclusion

Data structures with pointers are important abstract devices in software engineering to construct complex and dynamic networks. In this work, we have proposed a formal framework for investigating the issues in model-checking such systems. We have developed symbolic manipulation routine for BDD-like data-structures to calculate the pointer-references in a state-space. Two reduction techniques are then adapted to such systems. And our experiments have also shown that network configuration permutation is an indispensible technique in controlling the complexity of such software systems.

References

[1] R. Alur, C. Courcoubetis, D.L. Dill. Model Checking in Dense Real-Time, Information and Computation **104**, pp.2-34 (1993).

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond, IEEE LICS, 1990.

[3] M. Bozga, C. Daws. O. Maler. Kronos: A model-checking tool for real-time systems. 10th CAV, June/July 1998, LNCS 1427, Springer-Verlag.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. Hybrid Control System Symposium, 1996, LNCS, Springer-Verlag.

[5] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., C-35(8), 1986.

[6] E. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, Proceedings of Workshop on Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[7] E. Clarke, R. Enders, T. Filkorn, S. Jha. Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9, 77-104, 1996.

[8] D.L. Dill. Timing Assumptions and Verification of Finite-state Concurrent Systems. CAV'89, LNCS 407, Springer-Verlag.

[9] E.A. Emerson, A.P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM TOPLAS, Vol. **19**, Nr. 4, July 1997, pp. 617-638.

[10] A.P. Sistla, V. Gyuris, E.A. Emerson. SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties. TOSEM 9(2): Pages 133-166

[11] G.J. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4.

[12] G.J. Holzmann. The Spin Model Checker, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[13] G.J. Holzmann. An analysis of bitstate hashing, Workshop on Formal Methods in Systems Design, Nov. 1998.

[14] P.-A. Hsiung, F. Wang. User-Friendly Verification. Proceedings of 1999 FORTE/PSTV, October, 1999, Beijing. Formal Methods for Protocol Engineering and Distributed Systems, editors: J. Wu, S.T. Chanson, Q. Gao; Kluwer Academic Publishers.

[15] K. Jensen. Condensed state spaces for symmetrical coloured Petri nets. Formal Methods in System Design 9, 7-40, 1996.

[16] J.M. Mellor-Crummey, M.L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." ACM Transactions on Computer Systems, Vol. 9, No.1, Feb. 1991, pp.21-65.

[17] B. McKay. The nauty home page. `http://cs.anu.edu.au/~bdm/nauty/`.

[18] K. Schmidt. How to claculate symmetries of Petri nets. Acta Informatica 36, 545-590, 2000.

[19] K. Schmidt. Integrating low level symmetries into reachability analysis. Proc. TACAS 2000, LNCS 1785, 315-331, 2000.

[20] K. Schmidt. LoLA - a low level analyser. Proc. Int. Conf. Application and Theory of Petri nets 2000, LNCS 1825, 465-474, 2000.

[21] P. Starke. Reachability analysis of Petri nets using symmetries. J. Syst. Anal. Model. Simul. 8: 294-303, 1991.

[22] F. Wang. Efficient Data-Structure for Fully Symbolic Verification of Real-Time Software Systems. TACAS'2000, March, Berlin, Germany. in LNCS 1785, Springer-Verlag.

[23] F. Wang. Region Encoding Diagram for Fully Symbolic Verification of Real-Time Systems, in proceedings of COMPSAC'2000, Oct, Taipei, ROC.

[24] F. Wang. RED: Model-checker for Timed Automata with Clock-Restriction Diagram. Workshop on Real-Time Tools, Aug. 2001, Technical Report 2001-014, ISSN 1404-3203, Dept. of Information Technology, Uppsala University.

[25] F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram, to appear in Proceedings of FORTE, August 2001, Cheju Island, Korea.

[26] F. Wang, P.-A. Hsiung. Automatic Verification on the Large. Proceedings of the 3rd IEEE HASE, November 1998.

A MCS locking algorithm in RED for 2 processes

```
process count = 2;
global pointer L;
local pointer next, prev;
local discrete locked;

mode zero true { when true may next= NULL; goto one; }
mode one true { when true may prev= L; L= P; goto two; }
mode two true {
  when prev != NULL may goto three;
  when prev == NULL may goto six;
}
mode three true { when true may locked= 1; goto four; }
mode four true { when true may prev->next = P; goto five; }
mode five true {
  when locked == 1 may ;
  when locked == 0 may goto six;
}
mode six true {
  when next == NULL may goto seven;
  when next != NULL may goto nine;
}
mode seven true {
  when L == P may L= NULL; goto zero;
  when L != P may goto eight;
}
mode eight true {
  when next == NULL may ;
  when next != NULL may goto nine;
}
mode nine true { when true may next->locked = 0; goto zero; }

initially zero[1] and zero[2] and L == NULL;
risk
    (six[1] or seven[1] or eight[1] or nine[1])
and (six[2] or seven[2] or eight[2] or nine[2]);
```