BDD-based Safety-Analysis of Concurrent Software with Pointer Data Structures using Graph Automorphism Symmetry Reduction*

Farn Wang†
Dept. of Electrical Engineering, National Taiwan University
Taipei, Taiwan 106, R.O.C., farn@cc.ee.ntu.edu.tw

Karsten Schmidt‡
Dept. of Computer Science, Humboldt University of Berlin, Germany

Fang Yu, Geng-Dian Huang, Bow-Yaw Wang
Inst. of Info. Science, Academia Sinica, Taipei, Taiwan, R.O.C.

Abstract

Dynamic data-structures with pointer links are heavily used in real-world software and results in extremely difficult verification problems. Currently, there is no practical framework for the efficient verification of such systems. We investigated symmetry reduction techniques for the verification of software systems with C-like indirect reference chains like `x->y->z->w`. We formally defined the model of software with pointer data structures and developed symbolic algorithms to manipulate conditions and assignments with indirect reference chains using BDD technology. We relied on two techniques, inactive variable elimination and process-symmetry reduction in the data-structure configuration to reduce time and memory complexity. We used binary permutation for efficiency but we also identified the possibility of anomaly of false image reachability. We implemented the techniques in tool **Red** 5.0 and compare performance with Murϕ and SMC against several benchmarks.

Keywords: symbolic model checking, pointers, data structure, address manipulation, symmetry reduction, experiments

*A preliminary report of this work is to appear in the proceedings of FORTE'2002, in LNCS, Springer-Verlag.

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

‡Supported by DARPA/ITOP within the MoBIES project.

1 Introduction

The execution of real-world software may lead to the construction of complex and dynamic networks of data structures through pointers. Maintenance of correct linking through such pointers is not only cumbersome but also error-prone. Currently, although formal verification has the promise of automating the verification tasks in industrial projects, there is no practical framework that enables the formal verification of complex software with dynamic pointer links.

We investigate symmetry reduction techniques for the verification of software systems with C-like indirect reference chains. Such a chain takes the form x_1`->`x_2`->`$\ldots$`->`x_n. In C notation, if x_n is contained within a data structure of type T, then x_{n-1} is of type $*T$(type pointer to T), and for all $1 \leq i < n$, x_i points to a structure with a field x_{i+1} that points to something next in the chain. For example, we may have the following two structure type declarations for parsing trees in a C-program.

```
struct exp_type {
  int              op;
  struct atom_type  *atom;
  struct exp_type   *lhs, *rhs;
};
struct atom_type {
  char             *name;
};
```

If we declare a variable e of type `*exp_type` pointing to the parsing tree for $x + y$, then we can write an indirect reference chain like e`->lhs->atom->name` to reference to the string of "x". /labelr1:atom1 It is our goal to use BDD-based algorithms and symmetry-reduction techniques to enhance the verification performance for such systems.

Verification of networks with linear topologies like rings and buses has been widely studied. In real-world software, arbitrary and dynamic network configuration is, however, often constructed using pointers. We focus on the computational model with shared-memory concurrency involving two kinds of variables, global and local. A process has its own copy of each local variable and can directly access every process' copy of it. An action like "x_1->x_2->...->x_n = 3;" 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. Most CPUs now support hardware indirect referencing to facilitate virtual memory management. This hardware indirect referencing mechanism is transparent to software. Dynamic data structures like linear lists, trees, and graphs are constructed with pointers and used intensively in most nontrivial software.

In real-world software, an indirect reference chain may traverse through structures of various types. To lay a solid and elegant groundwork for this study, we shall assume that there is only one structure type, the *process* structure type. Without loss of generality, many structure types can be packed to one structure type. For example, we may combine the previously mentioned two structure type declarations into the following one.

```
struct process_type {
  int               exp_op;
  struct atom_type  *exp_atom;
  struct exp_type   *exp_lhs, *exp_rhs;
  char              *atom_name;
};
```

Thus from now on in this manuscript, a pointer's value not only points to a structure but can also be viewed as the identifier of the process of the structure. This assumption helps us to focus on the verification algorithms, instead of the multitude of data-types.

In example 1, we have a locking algorithm[23] which uses pointers (to process structures) to maintain a queue for critical section mutual exclusion.

Example 1 *:* **MCS (Mellor-Crummy & Scott's) locking algorithm.** This algorithm[23] is an example of a protocol in which a global waiting queue of processes is explicitly used to

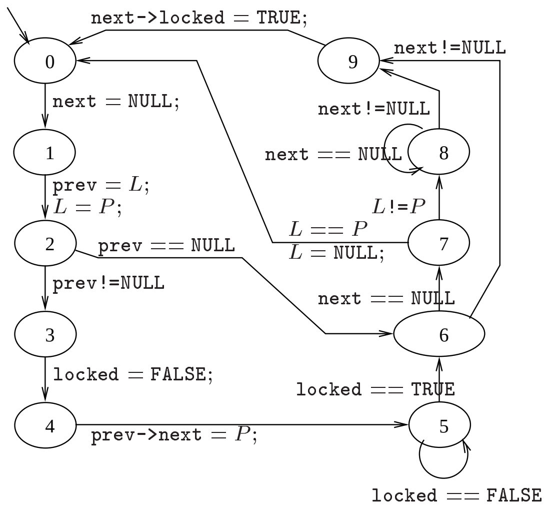

Figure 1: MCS locking algorithm

ensure mutually 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. We would like the system to have at most one process in modes six to nine (called critical section). 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` (pointing to the successor and predecessor processes of the local process in the queue respectively). P stands for a special symbol for the structure address of the current process. Each process has a local Boolean variable called `locked` which is set to true when the process is permitted to enter the critical section by its predecessor in the queue.

The transitions of the algorithm are represented by arrows. Each transition is associated with an action. Take the transition from mode four to five as an example. Its action is `prev->next = P;`. The expression `prev->next` denotes the local pointer `next` of the process structure referred to by the local pointer `prev` of the current process. Since the current process may update its local pointer `prev`, the same expression `prev->next` may refer to a different `next` pointer at different moments. Similarly, for the transition from mode nine to zero, the expression `next->locked` refers to the local pointer `locked` of the process pointed to by `next` of the current process.

We want to guarantee that at any moment, at most one process is in the critical section. ||

Due to their dynamic nature, software with pointer data structures have been known to be extremely difficult to maintain and debug. Any experienced software engineer will agree that bugs caused by the side-effects of pointer aliasing are extremely difficult to detect and remove. Such bugs, whose effect usually does not emerge until long after a data structure is corrupted through an aliasing reference, are very difficult to trace to their sources.

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 can achieve high efficiency in verification. In recent decades, the Binary Decision Diagram (BDD) [3, 6] has become a prime industry technology for symbolic manipulation. In this paper, we have the following accomplishments.

- We define a formal model for concurrent software with pointer data structures for rigorous research on solving the verification problem. Note that the framework allows all processes to share the same automaton template but have their own local variables. This is extremely important in identifying process-symmetry in a convincing way. Most other model checkers [4, 5, 18, 32] allow each process to be described by its own automaton, which usually creates difficulty in identifying symmetric behaviors among the process automata efficiently. (Note that asymmetric systems can also be modeled in our framework with processes running mutually disjointed parts of the program.)
- We have developed techniques for analyzing pointer conditions and assignments using BDD-like data structures. Algorithms for both forward and backward analysis have been developed and implemented. Both have been tuned for verification performance. Special care is taken to allow recurrent assignments, like $y\texttt{->}x = 3y\texttt{->}x + z$; where the left-hand-side may also occur in the right-hand-side.
- We have adapted two reduction techniques for model checking such systems.
  - *Reduction by inactive variables eliminations.* This helps the construction of concise state-space representation through the elimination of variable valuations that do not change system behavior [5, 18, 29, 30, 32]. Due to the implicit reading of pointers in the indirection of operand references, the adaptation cannot be done by syntactic analysis.

– *Reduction by process-symmetry.* The idea of symmetry reduction[9, 15, 25, 29, 32] is to represent symmetric states by a single state. We shall follow the approach of process-symmetry in [15, 29, 32] since process represents a typical basic unit for behavioral equivalence in symmetry. In general, to compute the full symmetry equivalence classes is expensive with factorial worst-case complexity. For efficiency, we use binary permutation of process identifiers to transform data structures to their automorphic ones with process re-indexing. How to design a symbolic predicate to detect the necessary condition for permutation is also discussed.

- We have implemented our modeling and verification techniques for pointer data-structure systems in our model-checker **Red** version 5.0, which is availabel for free at

  `http://www.cc.ee.ntu.edu.tw/~farn/red/` .

  The implementation not only supports pointer data structures but also complex arithmetics on process identifiers.

- We report experiments on several benchmarks to show the usefulness of our techniques and compare our method's performance to that of SMC[12] and Murϕ[11, 20].

In section 2, we discuss the related work briefly. In section 3, we define the formal framework of this research. In section 4, we present an algorithm that integrates safety-analysis software with various reduction techniques. In section 5, we present an algorithm that manipulates symbolic predicates and assignment statements using BDD-like data structures. In section 6, we discuss our reduction techniques. In section 7, we discuss our implementations and experiments. Section 8 gives the conclusion.

2 Related Work

The classic framework of model checking problem was first proposed by Clark and Emerson in [8] using finite-state automata for model descriptions. While we are investigating the verification of concurrent software, *parameterized systems*, in which many processes run the same piece of program, attract our attention. The verification of parameterized systems with unknown number of processes[2, 10, 13, 20, 21, 26, 27] may result in undecidability. Instead, we are more interested in verification of systems with given number of processes. In such a framework, it is important to reduce the verification complexity with respect to the number of processes.

In [12, 14, 15], a group-theoretic approach is used to reduce the state space for symmetric processes. In [20], a new data type is introduced for state-space reduction to exploit data symmetry. Using graph isomorphism, a symmetry reduction algorithm is proposed in [24]. But it was unclear whether the explicit-state reduction algorithm in [24] could be computed symbolically.

In general, it can be expensive to calculate the full equvivalence class of process symmetry with its factorial complexity. Facing this intrinsic challenge, researchers generally have to settle for abstraction of equivalence classes [12, 20]. In practice, for each verification task, we may need specific abstraction techniques for optimal performance. In the past, people have not focused on the development of such abstraction techniques for specific types of systems. In comparison, our GASR technique is specifically designed for abstraction of symmetry reduction in pointer data-structure systems.

To our knowledge, people have only implemented explicit-state algorithms for symmetry reduction [12, 20]. In 2000 and 2002, Wang experimented to use BDD-like data structure to implement symmetry reduction for timed automata[29, 31].

We also use binary permutations on process identifiers to transform states to their automorphic representatives in a sorting-like framework. Such an idea has previously been used in [29, 32, 31].

Other than symmetry reduction, many reduction techniques can also be found in literature. In partial-order reduction [16, 19, 22], sequences of independent transitions are represented by their representative. For internal transitions, internal action hiding [17, 18] merges them and thus reduce the number of states and transitions as well. The abstraction technique [7] tries to build an abstract model for the original model. The abstract model may disregard some of the behavior of the original, but it still has the necessary computations to verify specifications. However, it is unclear whether these reduction techniques are applicable to pointer data structures due to the complexity of pointer analysis.

Finally, in the last few years, people have also worked on the verification problems of parameterized systems with unknown number of processes. Such verification problems are in general undecidable and rely on various abstraction and widening techniques[2, 10, 13, 20, 21, 26, 27] to converge the state-space fixpoint evaluation.

3 Concurrent Algorithms and the Safety Analysis Problem

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. If p is the address of a process's data structure, we shall also name the process as p by convention.

Our model and techniques can be easily adapted to model and verify data symmetry instead of process symmetry. The idea is to declare a dummy process for each piece of allocated structure. Such dummy processes do not have corresponding transitions. The other executing processes can access, through pointers, the local data-fields (actually local variables) of those dummy processes.

Two types of variables can be declared. The first is the type of *discrete variables* of pre-declared finite integer value ranges. For each declared variable x, $\mathrm{lb}(x)$ and $\mathrm{ub}(x)$ denote its lower-bound and upper-bound respectively. Such variables can be used in formulae and assignments in arithmetic expressions and indirect operands. For convenience, we also assign symbolic macro names to integer values. Traditionally, `FALSE` is interpreted as 0 while `TRUE` as 1. The second type of variables are *pointers (address variables)* which point to processes (data structures). The values of pointers range from zero (`NULL`) to the number of processes. In example 1, L is used as a pointer to the tail of a queue. We also support arbitrary address arithmetics. A special constant pointer symbol is `NULL`, which in C's tradition is equal to zero. Or in the same notations as of discrete variables, $\mathrm{lb}(x) = \mathtt{NULL}$ and $\mathrm{ub}(x)$ is the number of processes for all declared pointers x.

Variables which are declared to be *global* are accessible to all processes. *Local* variables of a process can only be accessed by its corresponding process directly. A process must use indirect reference chains of pointers to access local variables of peer processes. 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`.

3.1 Syntax of Algorithm Description

Let us say that 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.

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

where $\sim \in \{$"<=", "<", "==", "!=", ">", ">="$\}$ is an inequality operator in C's notations, $c \in \mathcal{N} - \{0\}$, p is an integer from 1 to the number of processes, $x \in X^d \cup X^p$, $y \in X^p$, and $\oplus \in \{+, -, *, /\}$.

Here the notations $y\texttt{->}\epsilon$ and $y[p]\texttt{->}\epsilon$ for indirect references should be self-explaining to C-programmers. In the expression of $y\texttt{->}\epsilon$, y is either a local or global pointer variable that points to a structure and ϵ starts with a data-field in that structure type. If y is a local pointer, then it is interpreted as the one for the executing process. In the expression of $y[p]\texttt{->}\epsilon$, we specifically refer to the local pointer y of process p.

Parenthesis can be used for disambiguation. Traditional shorthands are $\epsilon_1\texttt{!=}\epsilon_2 \equiv \neg(\epsilon_1 = \epsilon_2)$, $\eta_1 \wedge \eta_2 \equiv \neg((\neg\eta_1) \vee (\neg\eta_2))$, and $\eta_1 \Rightarrow \eta_2 \equiv (\neg\eta_1) \vee \eta_2$. Thus a process may operate on conditions of global and local variables, and also on local variables of peer processes via 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 algorithm, once the triggering condition is satisfied by global and local variables of a process, the process may execute a finite sequence of actions of the form:

$$y_1\texttt{->}y_2\texttt{->}\ldots\texttt{->}y_n\texttt{->}x = \epsilon;$$

Let $T(X^d, X^p)$ be the set of all finite sequences of actions constructed with X^d and 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 a tuple (Q, Q_0, E, τ, π) with the following restrictions:

- Q is a finite set of operation modes.
- $Q_0 \subseteq Q$ is the set of initial operation modes.
- $E \subseteq Q \times Q$ 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 require that there is a variable $\texttt{mode} \in L^d$ that records the current operation mode of the corresponding process. When drawing $A(P)$ as an automaton, as in figure 1, we omit the description of `mode` values in the triggering conditions and action sequences for simplicity.

3.2 System Computation

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

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

such that

- $\nu(\texttt{NULL}, x) = \perp$ (memory fault) for all $x \in \mathcal{N}$ and all variables x.
- for all $1 \leq p \leq \mathcal{M}$, $\nu(p, \perp) = \perp$; $\nu(p, P) = p$; and $\nu(p, c) = c$ if $c \in \mathcal{N}$;
- for all $1 \leq p \leq \mathcal{M}$, $\nu(p, x)$ is the value of x at state ν, or more precisely,
  - for all $x \in G^d \cup L^d$, $\nu(p, x) \in [\text{lb}(x), \text{ub}(x)]$; and
  - for all $x \in G^p \cup L^p$, $\nu(p, x) \in \{\texttt{NULL}\} \cup \{1, \ldots, \mathcal{M}\}$ such that for all $1 \leq p' \leq \mathcal{M}$, $\nu(p, x) = \nu(p', x)$.

Given a 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 to $\{\texttt{TRUE}, \texttt{FALSE}, \perp\}$ as follows.[1]

- $\nu(p, y\texttt{->}\epsilon) = \nu(\nu(p, y), \epsilon)$ if $p \neq \texttt{NULL}$.
- $\nu(p, y[c]\texttt{->}\epsilon) = \nu(c, y\texttt{->}\epsilon)$ if $1 \leq c \leq \mathcal{M}$; otherwise, $\nu(p, y[c]\texttt{->}\epsilon) = \perp$.
- $\nu(p, \epsilon_1 \oplus \epsilon_2) = \perp$ 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 $\frac{x*y}{|x*y|}\lfloor |x/y| \rfloor$.
- $\nu(p, \epsilon_1 \oplus \epsilon_2) = \perp$ if $\epsilon_1 = \perp$ or $\epsilon_2 = \perp$.
- $\nu(p, \epsilon_1 \sim \epsilon_2) = \nu(p, \epsilon_1) \sim \nu(p, \epsilon_2)$
- "$\perp \sim \epsilon$" equals $\perp$, and "$\epsilon \sim \perp$" equals $\perp$.
- The negation of the satisfaction mapping is defined as

$\nu(p,\eta)$	FALSE	$\perp$	TRUE
$\nu(p,\neg\eta)$	TRUE	$\perp$	FALSE

- The disjunction of the satisfaction mapping is defined as

$\nu(p,\eta_1 \vee \eta_2)$	FALSE	$\perp$	TRUE
FALSE	FALSE	$\perp$	TRUE
$\perp$	$\perp$	$\perp$	$\perp$
TRUE	TRUE	$\perp$	TRUE

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

- When $\nu(p, y_1\texttt{->}\ldots\texttt{->}y_n\texttt{->}x) \neq \perp$ and $\nu(p, \epsilon) \neq \perp$, $\mathbf{next}(p, \nu, y_1\texttt{->}\ldots\texttt{->}y_n\texttt{->}x := \epsilon;)$ is exactly ν except $\mathbf{next}(p, \nu, y_1\texttt{->}\ldots\texttt{->}y_n\texttt{->}x := \epsilon;)(\nu(p, y_1\texttt{->}\ldots\texttt{->}y_n), x) = \nu(p, \epsilon)$.
- When either $\nu(p, y_1\texttt{->}\ldots\texttt{->}y_n\texttt{->}x) = \perp$ or $\nu(p, \epsilon) = \perp$, $\mathbf{next}(p, \nu, y_1\texttt{->}\ldots\texttt{->}y_n\texttt{->}x := \epsilon;)$ is undefined.

Note that the semantics are 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 $\mathbf{next}(p, \nu, \alpha_1\alpha_2\ldots\alpha_n) = \mathbf{next}(\mathbf{next}(p, \nu, \alpha_1), p, \alpha_2\ldots\alpha_n)$. An *initial state* ν_0 of an implementation $S^{\mathcal{M}}$ must satisfy $\bigwedge_{1\leq p\leq\mathcal{M}} \nu_0(p, \texttt{mode}) = 0$. Although there is no initial constraints in our framework, the processes can still set its variables' initial values through the first transition from mode 0. We assume that the system runs with *interleaving*

[1] We confess that this is a little symbol overloading of $\nu()$ since now the second arguments of ν are predicates instead of variables. But for the convenience of presentation, we think it is a natural extension since each predicate should have a value in this three-value logic of $\{\texttt{TRUE}, \texttt{FALSE}, \perp\}$.

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 an 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 $\nu_{k-1}(p, \tau(q, q')) = \texttt{TRUE}$ and $\textbf{next}(p, \nu_{k-1}, \pi(q, q')) = \nu_k$ is defined.

3.3 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.

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

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.[2]

- $\nu(\psi_1 \sim \psi_2) \;=\; \nu(\psi_1) \sim \nu(\psi_2) \in \{\texttt{TRUE}, \texttt{FALSE}\}$
- $\nu(x[p]) = \nu(p, x)$
- $\nu(\neg\phi) = \neg\nu(\phi)$
- $\nu(\phi_1 \vee \phi_2) = \nu(\phi_1) \vee \nu(\phi_2)$

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 if and only if there is a $k \geq 0$ such that either ν_k is undefined or $\nu_k(p, \phi) \neq \texttt{TRUE}$ 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 six through nine. Thus the safety analysis problem for mutual exclusive access to the critical sections of two processes can be formulated as $\mathrm{SAP}(S, 2, \neg(6 \leq \texttt{mode}[1] \leq 9 \wedge 6 \leq \texttt{mode}[2] \leq 9))$. ||

[2] Again, we here overload the meaning of $\nu()$ since now $\nu(\phi)$ has only one parameter when ϕ is global.

4 Framework for Safety Analysis and Reduction

The goal of the framework is to explore and construct a representation of the reachable state-space and analyze if the automaton ever violates the safety property. Our general algorithmic framework for symbolic safety analysis is shown below.

SAP($S, \mathcal{M}, \phi$) {

  reachable := $\bigwedge_{1\leq p\leq \mathcal{M}}$ `mode`$[p] = 0$; /* the initial state-predicate */

  next := `TRUE`;

  while(*next* $\neq$ `FALSE`) {

    next := `FALSE`;

    Sequentially for each $1 \leq p \leq \mathcal{M}$ and for each transition (q, q'), do {

      new := `indirect_condition`($reachable, p, \tau(q, q')$); (1)

      new := `indirect_assignment`($new, p, \pi(q, q')$); (2)

      new := `reduce`(*new*); /* application of reduction techniques */ (3)

      next := *next* $\vee$ (*new* $\wedge \neg$*reachable*);

    }

    reachable := *reachable* $\vee$ *next*;

  }

  if (*reachable* $\wedge\ \phi \neq$ `FALSE`) return "unsafe"; else return "safe";

}

The procedure iterates through the outer loop until *reachable* becomes a fix-point. At line (1), `indirect_condition`(D, p, η) returns a global predicate in BDD representing the subspace of D in which η is true of process p. At line (2), `indirect_assignment`$(D, p, \pi(q, q'))$ calculates 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 5. At line (3), reduce() simplifies reachable state-space representations with various reduction techniques.

¿From its appearance, procedure SAP() 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 6, we shall present two techniques to reduce the complexity of state space representations.

5 Manipulation of Predicates with Indirections

In our presentation of symbolic algorithms using BDD, we shall assume typical Boolean operations, such as conjunctions and negations, are already defined. Details of such BDD operations can be found in [3, 6].

5.1 Symbolic Evaluation of Conditions with Indirect Operands

In a pointer data-structure system, users may write a predicate with indirect reference chains of arbitrary lengths. 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 encoded by a finite number of bits in a BDD-like data structure. This is possible because their value ranges are finite. Specifically, $\texttt{lb}(count) = 0$, and $\texttt{ub}(count) = 5$.

Suppose a state-space representation D in a BDD-like data structure is given. We would like to compute the maximal subspace representation D' of D where

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

is true. The condition says that the difference of the count of the first process's parent ($\texttt{parent}[1]\texttt{->count}$) and twice the count of the right child of the second process's left child ($2{*}\texttt{leftchild}[2]\texttt{->rightchild->count}$) is less than the count of process L ($L\texttt{->count}$). Since there is no restriction on the length of indirections, we need a flexible algorithm to construct such a D'. Our simplified algorithm is the following function indirect_condition(), which in turn invokes functions indirect_ref() and indirect_arith().

```
indirect_condition(D, p, η) {
  Without loss of generality, rewrite η into the form ϵ ∼ c where c is a constant
  Construct D_ϵ := indirect_arith(p, ϵ);
  return D ∧ var_eliminate(D_ϵ ∧ VALUE ∼ c, VALUE);
}
```

Procedure var_eliminate(D, x) filters x out of D. For a local discrete variable $x[p]$, var_eliminate$(D, x[p]) = \bigvee_{v \in [lb(x), ub(x)]}(D \wedge x[p] == v)$. For a local pointer $x[p]$, var_eliminate$(D, x[p]) = \bigvee_{v \in \{\mathrm{NULL}, 1, \ldots, \mathcal{M}\}}(D \wedge x[p] == v)$.

Procedure indirect_condition() is simplified with the omission of codes to deal with problems like divide-by-zero and imprecision caused by integer division. In our implementation, the algorithm is more involved and takes care of many special cases. To focus on the algorithms, we only simplify the presentation. The algorithm uses auxiliary variables, VALUE and DPI (for *Destination Process Identifier*). VALUE has the value of an arithmetic expression. DPI stores the destination process identifier of the indirection.

Function indirect_ref(p, 1, l_1->...->l_k) constructs the condition that the value of l_1->...->l_k at process p equals the process identifier recorded in variable DPI.

```
indirect_ref(p, i, l_1->l_2->...->l_k) {
  if i > k, return(DPI == p);
  else if l_i is a local pointer l_i[j] with specific process reference j, then
    return ⋁_{1≤f≤M}(l_i[j] == f ∧ indirect_ref(f, i+1, l_1->...->l_k));
  else if l_i is a local pointer l_i with no specific process reference, then
      return ⋁_{1≤f≤M}(l_i[p] == f ∧ indirect_ref(f, i+1, l_1->...->l_k));
  else if l_i is a global pointer g_i, then
    return ⋁_{1≤f≤M}(g_i == f ∧ indirect_ref(f, i+1, l_1->...->l_k));
```

}

Function `indirect_arith`(p, ϵ) uses the auxiliary variable `VALUE` to symbolically record the value of expression ϵ at process p. It returns the predicate asserting that `VALUE` equals the value of expression ϵ for process p.

`indirect_arith`(p, ϵ) {

  $R :=$ `FALSE`;

  if ϵ is l_1`->`l_2`->`$\ldots$`->`l_k`->`x with $k > 0$, then {

    $H :=$ `indirect_ref`$(p, 1, l_1$`->`l_2`->`$\ldots$`->`$l_k)$;

    for $j := 1$ to $\mathcal{M}$, $\mathtt{lb}(x) \leq v \leq \mathtt{ub}(x)$, do

      $R := R \vee (H \wedge \mathtt{var\_eliminate}(\mathtt{DPI} == \mathtt{j} \wedge \mathtt{x}[\mathtt{j}] == \mathtt{v} \wedge \mathtt{VALUE} == \mathtt{v}, \mathtt{DPI}))$;

  }

  else if ϵ is c, then

    $R := R \vee (\mathtt{VALUE} == c)$;

  else if ϵ is $x[i]$ with local variable x and specific process reference i, then

    for $\mathtt{lb}(x) \leq v \leq \mathtt{ub}(x)$, do $R := R \vee (x[i] == v \wedge \mathtt{VALUE} == v)$;

  else if ϵ is local variable x with no specific process reference, then

    for $\mathtt{lb}(x) \leq v \leq \mathtt{ub}(x)$, do $R := R \vee (x[p] == v \wedge \mathtt{VALUE} == v)$;

  else if ϵ is a global variable x, then

    for $\mathtt{lb}(x) \leq v \leq \mathtt{ub}(x)$, do $R := R \vee (x == v \wedge \mathtt{VALUE} == v)$;

  else if ϵ is $\epsilon_1 \oplus \epsilon_2$ where $\oplus \in \{+, -, *, /\}$, then {

    $R_1 :=$ `indirect_arith`(p, ϵ_1);

    $R_2 :=$ `indirect_arith`(p, ϵ_2);

    for every possible combination of values v_1, v_2 of variable `VALUE`, do {

      $H_1 := \mathtt{var\_eliminate}(R_1 \wedge \mathtt{VALUE} == v_1, \mathtt{VALUE})$;

      $H_2 := \mathtt{var\_eliminate}(R_2 \wedge \mathtt{VALUE} == v_2, \mathtt{VALUE})$;

      $R := R \vee (H_1 \wedge H_2 \wedge \mathtt{VALUE} == v_1 \oplus v_2)$; (1)

  } }

  return R;

}

As an example, let us execute the just-mentioned procedures with $\eta = l\texttt{->}l\texttt{->}x > l\texttt{->}x + 3$ on a state-space described by $D = (l[1] == 2) \wedge (l[2] == 2) \wedge (x[1] == 4) \wedge (x[2] == 3)$. We first rewrite η to $l\texttt{->}l\texttt{->}x - l\texttt{->}x > 3$. Suppose the current process identifier $p = 1$. Further, let $\texttt{lb}(\texttt{x}) = \texttt{3}$ and $\texttt{ub}(\texttt{x}) = \texttt{4}$. Then

$$\texttt{indirect\_ref}(\texttt{p}, \texttt{1}, \texttt{l}) = (\texttt{l}[\texttt{1}] == \texttt{1} \wedge \texttt{DPI} == \texttt{1}) \vee (\texttt{l}[\texttt{1}] == \texttt{2} \wedge \texttt{DPI} == \texttt{2}).$$

Hence $\texttt{indirect\_arith}(\texttt{p}, \texttt{l->x})$

$$\begin{array}{rl} = & (l[1] == 1 \wedge x[1] == 3 \wedge \texttt{VALUE} == \texttt{3}) \;. \\ \vee & (l[1] == 1 \wedge x[1] == 4 \wedge \texttt{VALUE} == \texttt{4}) \\ \vee & (l[1] == 2 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{3}) \\ \vee & (l[1] == 2 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{4}) \end{array}$$

Similarly, since $\texttt{indirect\_ref}(\texttt{p}, \texttt{1}, \texttt{l->l})$

$$\begin{array}{rl} = & (l[1] == 1 \wedge l[1] == 1 \wedge \texttt{DPI} == \texttt{1}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge \texttt{DPI} == \texttt{2}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge \texttt{DPI} == \texttt{1}) \\ \vee & (l[1] == 2 \wedge l[2] == 2 \wedge \texttt{DPI} == \texttt{2}) \end{array}$$

we have $\texttt{indirect\_arith}(\texttt{p}, \texttt{l->l->x})$

$$\begin{array}{rl} = & (l[1] == 1 \wedge l[1] == 1 \wedge x[1] == 3 \wedge \texttt{VALUE} == \texttt{3}) \\ \vee & (l[1] == 1 \wedge l[1] == 1 \wedge x[1] == 4 \wedge \texttt{VALUE} == \texttt{4}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{3}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{4}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 3 \wedge \texttt{VALUE} == \texttt{3}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 4 \wedge \texttt{VALUE} == \texttt{4}) \\ \vee & (l[1] == 2 \wedge l[2] == 2 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{3}) \\ \vee & (l[1] == 2 \wedge l[2] == 2 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{4}) \end{array}$$

Thus, $\texttt{indirect\_arith}(\texttt{p}, \texttt{1}, \texttt{l->l->x} - \texttt{l->x})$

$$\begin{array}{rl} = & (l[1] == 1 \wedge l[1] == 1 \wedge x[1] == 3 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 1 \wedge l[1] == 1 \wedge x[1] == 4 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[1] == 3 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[1] == 4 \wedge x[2] == 3 \wedge \texttt{VALUE} == -\texttt{1}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[1] == 3 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{1}) \\ \vee & (l[1] == 1 \wedge l[1] == 2 \wedge x[1] == 4 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 3 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 3 \wedge x[2] == 4 \wedge \texttt{VALUE} == -\texttt{1}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 4 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{1}) \\ \vee & (l[1] == 2 \wedge l[2] == 1 \wedge x[1] == 4 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 2 \wedge l[2] == 2 \wedge x[2] == 3 \wedge \texttt{VALUE} == \texttt{0}) \\ \vee & (l[1] == 2 \wedge l[2] == 2 \wedge x[2] == 4 \wedge \texttt{VALUE} == \texttt{0}) \end{array}$$

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 pairwisely compared and corresponding state-predicate conjuncted, as in line (1) in procedure `indirect_arith()`.

5.2 Symbolic Assignments with Indirect Operands

Given a state-space predicate D and an assignment statement $\omega := \epsilon;$, one may think its symbolic postcondition in process p would be

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

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

Our algorithm solves the recurrence assignment problem by auxiliary variable `VALUE` as a temporary recorder for the expression value. The destination variables are eliminated from the symbolic predicate by procedure `var_eliminate`() before being assigned by procedure `condition_effect`(). Our algorithm is given as follows.

$\texttt{indirect\_assignment}(D, p, \omega := \epsilon;)$ {

Construct $D_\epsilon := D \wedge \texttt{indirect\_arith}(\epsilon, p)$;

if ω is l_1->l_2->$\ldots$->l_k->x with $k > 0$, then {

Let $R :=$ `FALSE`;

Construct $D_\epsilon := D_\epsilon \wedge \texttt{indirect\_ref}(p, 1, l_1\texttt{->}l_2\texttt{->}\ldots\texttt{->}l_k)$;

for $j := 1$ to $\mathcal{M}$, do {

Let $H := \texttt{var\_eliminate}(\texttt{var\_eliminate}(\texttt{D}_\epsilon \wedge \texttt{DPI} == \texttt{j}, \texttt{DPI}), \texttt{x[j]})$; (2)

Let $H := \texttt{condition\_effect}(x[j], \sim, H)$;

Let $R := R \vee H$;

} }

else if ω_0 is $x[i]$ with local variable x with specific process reference i, then

Let $R := \texttt{condition\_effect}(x[i], \sim, \texttt{var\_eliminate}(D_\epsilon, x[i]))$; (3)

else if ω_0 is local variable x with no specific process reference, then

Let $R := \texttt{condition\_effect}(x[p], \sim, \texttt{var\_eliminate}(D_\epsilon, x[p]))$; (4)

else if ω_0 is a global variable x, then

Let $R := \texttt{condition\_effect}(x, \sim, \texttt{var\_eliminate}(D_\epsilon, x))$; (5)

return R;

}

`condition_effect`$(x, \sim, D)$ {

  $R :=$ `FALSE`;

  for every possible value v of variable `VALUE`, do

    $R := R \vee (x \sim v \wedge \texttt{var\_eliminate}(D \wedge \texttt{VALUE} == v, \texttt{VALUE}))$;

  return $R \wedge \texttt{lb}(x) \leq x \leq \texttt{ub}(x)$;

}

6 Reduction Techniques

We rely on two reduction techniques to alleviate the state-space explosion problem. They are respectively discussed in the following two subsections.

6.1 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. Similar techniques have been used heavily in tools like Spin[18], UPPAAL[5], SGM[32], and **Red**[29, 30]. 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 over-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 down 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 over-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 over-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 over-approximation of the active condition.

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

- an indirect reference like $y_1\texttt{->}\ldots\texttt{->}y_m\texttt{->}y$ occurs in ω, $p = p'$, and $x = y_1$; or
- an indirect reference like $y_1\texttt{->}\ldots\texttt{->}y_m\texttt{->}y$ occurs in ω and $x \in \{y_2, \ldots, y_m\}$; or
- an indirect reference like $y_1\texttt{->}\ldots\texttt{->}y_m$ occurs in ϵ, $p = p'$, and $x = y_1$; or
- an indirect reference like $y_1\texttt{->}\ldots\texttt{->}y_m$ occurs in ϵ and $x \in \{y_2, \ldots, y_m\}$.

Given a local variable $x[p]$, an over-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.

$\texttt{base\_oapprox\_active}(x)$

let $\eta_x := \texttt{FALSE}$;

for each transition (q, q') in $A(P)$,

if x is possibly read in actions in $\pi(q, q')$,

then $\eta_x := \eta_x \vee (\texttt{mode} == q \wedge \texttt{var\_eliminate}(\tau(q, q'), x))$.

else {

break $\tau(q, q')$ into DNF $\Delta_1 \vee \Delta_2 \vee \ldots \vee \Delta_k$;

for each Δ_i, if x appears in Δ_i, then $\eta_x := \eta_x \vee (\texttt{mode} == q \wedge \texttt{var\_eliminate}(\Delta_i, x))$.

} }

return η_x;

}

We need the following concept. A transition is *disrupting* to local variable x iff it assigns a value to x which is not computed from x in the transition. For example, a transition with assignment sequence $x = 3; y = x;$ is disrupting for both x and y. For another example, a transition with assignment sequence $x = y; y = x + 3;$ is disrupting for x but not for y. A disrupting transition for a variable x marks an event that the value of x before the event does not affect the behavior of the system from the event on.

In the second step, from the base approximation, we calculate an over-approximation of the backward weakest precondition through each transition until a least fix-point is reached. This is done in the following procedure.

$\texttt{oapprox\_active}(x)$ {

  $\eta_x := \texttt{base\_oapprox\_active}(x); \eta'_x := \texttt{FALSE};$

  while($\eta'_x \neq \eta_x$) {

    $\eta'_x := \eta_x;$

    for each transition (q, q') that is non-disrupting to x in $A(P)$, {

      let $\eta'_x := \eta'_x \vee \delta$, where δ is an over-approximation of

        the weakest precondition of η_x before applying (q, q'). (6)

  } }

  return η_x.

}

The function $\texttt{oapprox\_active}_{\texttt{x}}(\texttt{s})$ computes the predicate that x can be read in some actions along a path from state s before some of its disrupting transitions takes place. The over-approximation technique that we use in statement (6) discards (i.e., existentially quantifies) any local variables of peer processes. It can be proven that the over-approximation local state predicate is indeed independent of x. By applying our technique to the MCS algorithm in figure 1, we find that

$$\begin{array}{rcl} \text{active}_{\texttt{locked}} & = & 4 <= \texttt{mode} <= 5 \\ \text{active}_{\texttt{next}} & = & \texttt{mode} == 1 \vee (2 <= \texttt{mode} <= 4 \wedge \texttt{prev!=}P) \vee \texttt{mode} >= 5 \\ \text{active}_{\texttt{prev}} & = & 1 <= \texttt{mode} <= 4 \end{array}$$

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.

6.2 Graph automorphism symmetry reduction (GASR)

We follow the reduction framework in [15] to permute process identifiers to take advantage of the symmetry among processes running different copies of the same program. Our idea is to use the pointing-to relations of the global and local pointers to define a precedence relation among processes in a state, then permute the processes according to the precedence relation. We view the pointer data structure as a directed graph. Each global pointer and each process is viewed as a node while the pointing-to relation is viewed as arcs from nodes to nodes. Thus the symmetry reduction for pointer data structures has the flavor of graph isomorphism problem with node renaming, which is not yet known to be in PTIME. Intuitively, we want to keep as few data structures, which are isomorphic, as possible.

Efficient binary permutation

There are two challenges here. The first is to design an efficient symmetry reduction strategy. For m processes, we have $m!$ different permutations and obviously we do not want to try them all to find the best permutation. Our technique is to use binary permutations, which permute two processes each time, to compose full permutation. In theory, we know that all permutations can be constructed with a sequence of binary permutations. This is to say that bubble-sort works for any sequence. However, binary permutations can create some data-structure configurations which are not reachable. 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 binary permutation $\sigma = (132)$ 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 an initial state if the cycle is a static one. We call this problem the *anomaly of image false reachability.* Although this is a possible cause for imprecision, we choose to live with it knowing that graph isomorphism problem can be too complex to solve.

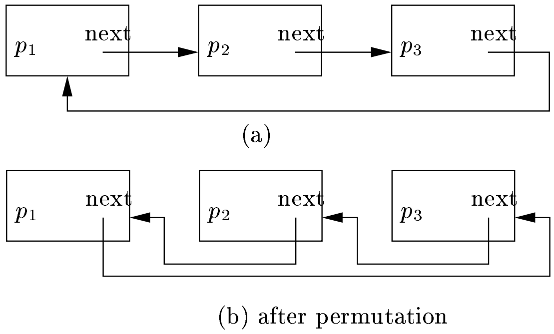

Figure 2: Anomaly of image false reachability

How to handle the anomaly ?

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

LEMMA 1 *Assume a reduced transition system that contains one member per equivalence class of states w.r.t. the group of binary permutations 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 $\sigma\nu$ (the application of permutation σ to ν) 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 without any loss in precision caused by the anomaly of image false reachability.

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 binary permutations $\sigma_{p_i p_j}$ 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 reduced ordered BDD to check whether this exchange of roles between p_i and p_j leads to the same initial condition. If this is the case then $\sigma_{p_i p_j}$ leaves the initial condition invariant.

Symmetry reduction as sorting

In particular, set Σ is closed under composition and inversion. Moreover, such a group Σ has a well-structured generating set—the set of all binary permutations $\sigma_{p_i p_j}$ where $p_i \neq p_j$, neither p_i nor p_j are among the "forbidden" process identifiers, $\sigma_{p_i p_j}(p_i) = p_j$, $\sigma_{p_i p_j}(p_j) = p_i$, and $\sigma_{p_i p_j}(p_k) = p_k$ 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 binary permutations until some kind of "lexicographically" smallest state is achieved.

With binary permutation, we have to construct a predicate $\texttt{reverse}(i, j)$ in BDD, for each $1 \leq i < j \leq m$ which characterizes those data-structure configurations in which processes i and j have to be permuted. For the efficient computation of symmetry group, $\texttt{reverse}()$ may actually lead to an over-approximation of the symmetry group Σ. Once predicate $\texttt{reverse}(i, j)$ is ready for each $1 \leq i < j \leq \mathcal{M}$, the following procedure implements our GASR. Given a $\sigma \in \Sigma$, the following procedure uses $\texttt{reverse}()$ to iteratively permute σ to a "normalized" image in Σ.

$\texttt{reduce\_symmetry}(\eta)$ {

  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 η;

}

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.

Since this process resembles conventional sorting procedures, it yields a unique, minimal 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 automorphic representatives.

Criterion for binary permutation

The second challenge is to design a criterion to determine when we need to permute two process identifiers. In other words, how do we design predicate $\texttt{reverse}()$? Our technique is to define an artificial distinct significance to each global and local pointer. For example, in the MCS algorithm, in our significance scale, the process pointed to by L is much more significant than the others. Thus the process pointed to by L should precede all other processes after the permutation. Basically, we assign the significance to global pointers according to their declaration order. The same is true among local pointers. Suppose local pointer `next` is declared before `prev` in MCS algorithm. Thus if neither process i or j is pointed to by L and process i's local pointer `next` points to process j, then we know process i cannot be preceded by process j after the permutation. We have to consider the pointing-to relation of local pointer `prev` to decide the precedence between two processes only when we cannot decide their precedence with L and `next`. In figure 3, we have drawn the four-process network constructed by the MCS algorithm respectively before and after our permutation in figure 1. After the permutation, the network nodes are reordered in a linear sequence according to the queue formation.

Note that in our implementation, predicate `reverse`(i,j) does not use information on indirection paths of length > 1 between processes i and j. This is for the complexity of the BDD for $\texttt{reverse}(i, j)$. In our experiment, if we consider indirection path lengths > 1 in the construction of $\texttt{reverse}(i, j)$, the sizes of BDDs become too big to represent and manipulate

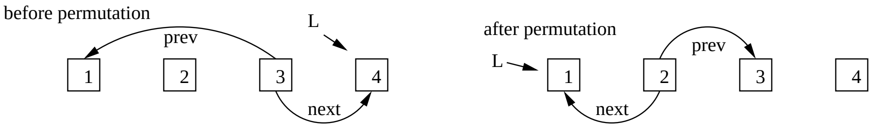

Figure 3: permutation of process identifiers

efficiently. When `reverse`(i, j) are FALSE, it only means we have no rule to break the tie between processes i and j. Procedure `reduce_symmetry`() does not break any ties either.

7 Implementation and Experiments

We implemented our techniques in a symbolic verification tool called **Red**[29, 30], which supports verification of timed automata [1] with a new BDD-like data structure for dense-time state-space representation. The reduction by inactive variable elimination is automatic since in almost all previous work, it has been shown to be indispensible for verification performance. Instead, our focus in the experiment is on symmetry reduction and our BDD implementation. The symmetry reduction for pointer data structure is invoked by option "Sp." We compared the performance of **Red**, both with and without the symmetry reduction technique, with that of SMC[12] and Murϕ[20] running in various options. Since neither SMC nor Murϕ supports pointers, we use arrays to encode the pointers instead.

There are six benchmarks in our experiments. In the following, we describe each benchmark and report its experiment in a subsection. Together, in table 1, we list the sizes of the benchmarks. The last three benchmarks are all extracted from the classic textbook *Operating System Concepts* by Silberschatz, Galvin, Gagne[28]. Due to the popularity of this textbook, we believe these three benchmarks objectively help in proving the value of our techniques.

All the experiments were carried out on a Pentium 4 2.1GHz/256MB PC running cygwin. All data related to **Red** are collected with forward analysis (option -f). In each entry of the rows, the CPU times and memory consumptions (in kilobytes) are shown. The memory complexity for **Red** is collected only for BDDs and their management.

The verification algorithm of `Mur`ϕ `3.1` is breadth first search with various symmetry algorithms including exhaustive canonicalization(-sym1), heuristic fast canonicalization(-sym2),

Sizes of process program template $A(P)$	modes	transitions	global p'ters	global disc.	local p'ters	local disc.
MCS locking algorithm	10	15	1	0	2	1:[0,1]
Leader election	2	3	1	0	1	0
Doubly linked cycle	2	5	1	0	2	0
Bounded buffer	10	26	6	3:[0,10],1:[0,5]	2	1:[0,1]
Reader-writer	14	36	4	2:[0,15],1:[0,r]	2	1:[0,1]
Critical region	19	47	6	2:[0,m],3:[0,m+1]	2	1:[0,1]

For discrete variables, $h : [i, j]$ means h variables with value range $[i, j]$.
r: # readers; m: # processes;

Table 1: Sizes of the benchmarks

heuristic small memory normalization with permutation trial limit 10(-sym3) and heuristic fast normalization(-sym4). The memory allocated for the hash table and state queue is 202 Mbytes. The benchmarks run without deadlock detection and all data are composed of both compile and execution time.

For `SMC`, we tried various combinations of its built-in symmetry reduction heuristics. `SMC` option `-s`$\langle num \rangle$ is the symmetry option. If the value of $\langle num \rangle$ is even, only process symmetry is employed. Otherwise, both process symmetry and state symmetry are employed. The higher the value, the more sophisticated is the equivalence checking algorithm employed.

7.1 MCS locking algorithm

This is the MCS locking algorithm shown in example 1[23]. We want to verify that at most one process can be in the critical section at all times. The result of the experiment is shown in table 2. We can see that our tool is able to verify the system of ten processes. Both `SMC` and `Mur`ϕ have difficulties for systems with more than five processes.

7.2 Leader election with dynamic forest configurations

A version of this algorithm is used in the IEEE 1394 Firewire protocol. This benchmark is unique in that the network configurations are forests instead linear lists. We choose this benchmark to observe how our techniques perform against nonlinear dynamic network configurations. In this benchmark, each process has a local pointer `parent` which is set to `NULL`

Tools	Options	3	4	5	6	7	8	9	10
red	-fSp	0.30s	2.14s	11.41s	64.43s	461.71s	2933.40s	15137.08s	71676.51s
		25k	68k	168k	460k	1311k	3894k	11107k	31226k
	-f	3.37s	47.77s	1095.77s	16393.27s	Out of Memory			
		121k	504k	3937k	28995k				
SMC	-s1	145.0s	Not Available						
		C/D							
	-s2	596.4s	>17h						
		13472k	Unfinished						
	-s3	600.3s	>17h						
		13477k	Unfinished						
	-s4	1601.8s	20252.6s						
		13460k	C/D						
	-s5	1624.0s	>17h						
		13457k	Unfinished						
	-s6	1600.3s	>17h						
		13459k	Unfinished						
	-s7	1620.8s	>17h						
		13457k	Unfinished						
Murphi	-sym1	3.93s	28.51s	Internal Error	Not Available				
	-sym2	2.77s	25.40s						
	-sym3	2.77s	25.37s						
	-sym4	2.89s	25.30s						

Table 2: MCS Locking Algorithm

initially. Processes send random requests to become a child of another process until pointer `parent` is set to the parent process's identifier. A process responds to a request by writing its identifier to a global pointer `respond_id`. The requesting process then updates its local pointer `parent` according to the content of `respond_id`. A group of symmetric processes thus form a dynamic forest structure by the pointer `parent`. Our task is to check that there exists at least one root in the forest at any time. The result of the experiment is shown in table 3. Our technique is able to verify a system with nine processes in under one second. With its best reduction scheme, `SMC` is able to finish the task in 590.0 seconds. As for `Mur`ϕ, an internal error occurs for the eight-process system. But for the seven-process system, `Mur`ϕ uses more than 40 seconds for the verification.

7.3 Doubly linked cycle insertion and deletion

The third benchmark is the insertion and deletion algorithms for a dynamic double-link cycle. It is adapted from our source program for **Red**. The cycle consists of a set of symmetric

Tools	Options	3	4	5	6	7	8	9	10
red	-fSp	0.01s	0.01s	0.02s	0.01s	0.02s	0.02s	0.04s	0.05s
		7k	13k	23k	36k	54k	77k	108k	145k
	-f	0.04s	0.06s	0.22s	1.20s	10.11s	94.60s	1055.34s	Out of
		15k	51k	192k	803k	4022k	20785k	110974k	Memory
SMC	-s1	0.3s	0.5s	0.3s	0.7s	4.8s	62.7s	2096.4s	Not
		1k	7k	34k	193k	1224k	8444k	68240k	Available
	-s2	0.2s	0.2s	0.4s	2.4s	42.7s	1097.2s	34511.2s	
		1k	7k	29k	135k	681k	3707k	21799k	
	-s3	0.2s	0.2s	0.4s	2.3s	29.5s	944.1s	16335.1s	
		1k	7k	28k	134k	567k	3451k	14964k	
	-s4	0.2s	0.4s	0.4s	1.4s	9.2s	73.7s	619.0s	
		1k	6k	19k	62k	196k	604k	1857k	
	-s5	0.3s	0.3s	0.4s	1.3s	8.9s	70.4s	591.0s	
		1k	6k	19k	62k	195k	602k	1851k	
	-s6	0.3s	0.3s	0.5s	1.4s	9.1s	73.3s	621.0s	
		1k	6k	19k	62k	196k	604k	1857k	
	-s7	0.2s	0.3s	0.4s	1.3s	8.8s	70.7s	592.6s	
		1k	6k	19k	62k	195k	602k	1851k	
Murphi	-sym1	2.69s	2.43s	2.49s	3.79s	46.00s	Internal	Not Available	
	-sym2	2.68s	2.50s	2.49s	3.58s	41.86s	Error		
	-sym3	2.69s	2.37s	2.52s	3.58s	41.86s			
	-sym4	2.71s	2.50s	2.57s	3.57s	42.20s			

Table 3: Leader Election Algorithm

processes connected by two local pointers `next` and `prev`. Each process tries to insert and delete itself from the cycle randomly. A global pointer `L` points to the tail of the cycle. If the cycle is empty, pointer `L` is equal to `NULL`. In the experiment, we would verify that pointer `L` is not `NULL` when a process thinks itself is in the cycle. The result of the experiment is shown in table 4. Our tool is able to verify the system with ten processes in 10 seconds. In contrast, all tests on `SMC` run out of time (over 17 hours), while `Mur`ϕ produces an internal error for the nine-process system.

7.4 Bounded buffer algorithm

A group of producers generate goods and put them in the bounded buffer. When the buffer is full, producers are put in the waiting queue and stop generating messages. Consumers, on the other hand, take goods from the bounded buffer. If the buffer is empty, they will be put in the waiting queue. Our goal is to verify that the bounded buffer can never overflow. The performance data of experiment is in table 5. Our tool scales much better than peer tools

Tools	Options	3	4	5	6	7	8	9	10
red	-fSp	0.03s	0.11s	0.30s	0.66s	1.42s	2.80s	5.20s	9.23s
		66k	200k	471k	953k	1746k	2983k	4861k	7634k
	-f	0.07s	0.26s	0.91s	5.00s	56.79s	Out of Memory		
		51k	213k	948k	4996k	30712k			
SMC	No termination								
Murphi	-sym1	2.53s	2.63s	2.39s	2.40s	3.51s	14.94s	Internal Error	Not Available
	-sym2	2.62s	2.61s	2.37s	2.37s	3.25s	11.98s		
	-sym3	2.62s	2.61s	2.37s	2.37s	3.22s	11.97s		
	-sym4	2.59s	2.60s	2.38s	2.49s	3.25s	12.51s		

Table 4: Doubly Linked Cycle Operations

w.r.t. concurrency complexity.

7.5 Reader-writer algorithm

This experiment models groups of readers and writers trying to access a shared object. Several readers can read the same object simultaneously. But only one writer accesses an object exclusively. In our benchmark, we model the variant called the *first* reader-writers problem where no reader will be kept waiting unless a writer has obtained permission to use the shared object. We want to check whether the shared object will be read and written at the same time. The performance data of experiment is in table 6. Again, our tool scales much better than peer tools w.r.t. concurrency complexity.

7.6 Conditional critical region

Our last experiment models a high-level synchronization construct called *conditional critical region*. Suppose variable v is to be shared by several processes. The programmer can use the following construct to access v exclusively: "`region v when B do S`", where B is a Boolean expression and S is a (compound) statement. In our model, each process repeatedly executes the above region statement with nondeterministic Boolean condition B. And the safety property checks the mutual exclusiveness of the critical region S. The performance data of experiment is in table 7. And again, our tool scales much better than peer tools w.r.t. concurrency complexity.

Tools	Options	3	4	5	6	7	8	9	10	11	12	13
red	-fSp	1.84s	6.93s	26.48s	65.62s	254.07s	417.63s	1059.98s	1538.85s	3151.59s	4321.84s	7777.07s
		121k	210k	391k	622k	1018k	1195k	1750k	1939k	2688k	2918k	3884k
	-f	4.22s	31.15s	246.43s	1451.21s	5622.91s	19924.80s	68785.24s	Out of Memory			
		143k	502k	1398k	4323k	11623k	30836k	85984k				
SMC	-s1	0s	2s	18s	201s	Out of Memory						
		109k	833k	5647k	45008k							
	-s2	0s	2s	23s	260s							
		109k	833k	5648k	45008k							
	-s3	0s	2s	23s	264s							
		109k	833k	5647k	45008k							
	-s4	0s	2s	26s	289s							
		109k	833k	5648k	45008k							
	-s5	0s	3s	27s	290s							
		109k	833k	5647k	45008k							
	-s6	0s	2s	25s	279s							
		109k	833k	5648k	45008k							
	-s7	0s	2s	27s	299s							
		109k	833k	5674k	45008k							
Murphi	-sym1	3.12s	3.06s	3.32s	5.85s	21.57s	150.95s	Internal Error	Not Available			
	-sym2	3.14s	3.03s	3.89s	5.63s	20.02s	140.57s					
	-sym3	3.09s	3.00s	3.81s	5.53s	20.04s	140.57s					
	-sym4	3.12s	3.10s	3.91s	5.58s	20.16s	141.51s					

Table 5: Bounded Buffer Problem

7.7 Discussion of the experiments

The first three benchmarks represent three types of dynamic data structures: a double-linked queue, a forest and a double-linked cycle. Each may contain an arbitrary number of symmetric processes. For this three, our tool performs better with respect to concurrency complexity.

The remaining benchmarks use semaphores to solve the synchronization problem. For each semaphore, a doubly linked queue is used to record the blocked processes. Each blocked process is removed from the queue For these three, our tool does not perform as well as `SMC` or `Mur`ϕ in small systems. However, our reduction technique is able to verify systems with many processes successfully. In contrast, we either run out of memory (for `SMC`) or encounter internal errors (for `Mur`ϕ) in large systems. This may suggest our technique is more scalabel than those deployed in the other tools.

Tools	Options	3	4	5	6	7	8	9	10	11	12	13
red	-fSp	1.25s	2.90s	23.73s	42.71s	276.57s	440.48s	1898.16s	2547.08s	7696.00s	9732.03s	23941.32s
		220k	244k	321k	415k	896k	996k	2000k	2119k	3898k	4037k	6962k
	-f	3.99s	15.11s	250.41s	1034.94s	8187.32s	21179.90s	106175.88s	Out of Memory			
		220k	244k	1106k	2454k	9932k	22906k	71049k				
SMC	-s1	0s	1s	26s	238s	Out of Memory						
		96k	626k	6602k	49549k							
	-s2	0s	2s	31s	304s							
		96k	626k	6602k	49550k							
	-s3	0s	2s	31s	219s							
		96k	626k	6602k	49549k							
	-s4	0s	2s	33s	311s							
		96k	626k	6602k	49550k							
	-s5	0s	2s	35s	331s							
		96k	626k	6602k	49549k							
	-s6	0s	2s	32s	302s							
		96k	626k	6602k	49550k							
	-s7	0s	2s	34s	318s							
		96k	626k	6602k	49549k							
Murphi	-sym1	3.35s	3.08s	3.62s	5.37s	44.76s	287.14s	Internal Error	Not Available			
	-sym2	3.38s	3.08s	3.58s	5.22s	42.24s	276.76s					
	-sym3	3.20s	2.96s	3.57s	5.17s	42.23s	274.04s					
	-sym4	3.34s	3.03s	3.62s	5.23s	42.17s	276.14s					

Table 6: Reader-Writer Problem

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 GASR is an indispensable technique in controlling the complexity of such software systems.

As we have pointed out, full symmetry equivalence classes can be expensive to compute with their factorial complexity. In the future, it will be interesting to see whether we can devise other abstraction techniques for symmetry reduction of software systems.

Tools	Options	3	4	5	6	7
red	-fSp	40.14s	751.14s	5428.41s	23197.71s	75229.25s
		10224k	11065k	11941k	12833k	15618k
	-f	142.34s	4640.21s	66400.22s	Out of Memory	
		10224k	11065k	63655k		
SMC	No termination					
Murphi	-sym1	4.00s	7.02s	49.75s	Internal	Not
	-sym2	4.00s	6.78s	46.09s	Error	Available
	-sym3	4.01s	6.75s	46.01s		
	-sym4	3.97s	6.80s	46.00s		

Table 7: Conditional Region Construct

References

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

[2] M.C. Browne, E.M. Clarke, O. Grumberg. Reasoning about networks with many finite processes. 5th ACM PODC, p.240-248, 1986.

[3] 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.

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

[5] 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.

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

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th Annual Symposium on Principles of Programming Languages. ACM Press, 1977.

[8] 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.

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

[10] E.M. Clarke, O. Grumberg, S. Jha. Verifying parameterized networks using abstraction and regular languages. In Proc. 6th International Conference on Concurrency Theory, p.395-407, August 1995.

[11] D.L. Dill. The Murphi Verification System. CAV 1996, LNCS, Springer-Verlag.

[12] 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

[13] E.A. Emerson, K.S. Namjoshi. Reasoning about rings. 22th ACM POPL, 1995.

[14] E.A. Emerson, A.P. Sistla. Symmetry and Model Checking. Formal Methods in System Design: An International Journal, Vol 9(1), pp 105–131. August, 1996.

[15] 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.

[16] P. Godefroid. Using partial orders to improve automatic verification methods, Proc. of Computer Aided Verification Workshop. 1990.

[17] G. Holzmann, P. Godefroid, D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. 12th Int'l Symposium on Protocol Specification, Testing, and Verification (PSTV), Lake Buena Vista, Florida, North-Holland, June 1992.

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

[19] G.J. Holzmann, D. Peled. An Improvement in Formal Verification, Proc. of the 1994 FORTE Conference, Bern, Switzerland.

[20] C.N. Ip, D.L. Dill. Better Verification through Symmetry. FMSD 9(1/2):41-75, 1996.

[21] R.P. Kurshan, K. McMillan. A structural induction theorem for processes. 8th ACM PODC, p.239-248, 1989.

[22] S. Katz, D.A. Peled. Verification of distributed programs using representative interleaving sequences. Distributed Computing (6), pp 107-120. 1992.

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

[24] Robby, M.B. Dwyer, J. Hatcliff, R. Iosif. Space-Reduction Strategies for Model Checking Dynamic Software. Electronic Notes in Theoretical Computer Science 89(3), 2003.

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

[26] A.P. Sistla, S.M. German. Reasoning about systems with many processes. JACM, 30:675-735, 1992.

[27] A.P. Sistla. Parameterized verification of linear networks using automata as invariants. CAV'97, LNCS, p.412-423, Springer-VERlag, 1997.

[28] A. Silberschatz, P.B. Galvin, G. Gagne. Operating System Concepts. Jhon Wiley & Sons, Inc. 2003.

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

[30] F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram, IFIP FORTE, August 2001, Cheju Island, Korea.

[31] F. Wang. Symmetric Model-Checking of Concurrent Timed Automata with Clock-Restriction Diagram. RTCSA'2002.

[32] F. Wang, P.-A. Hsiung. Efficient and User-Friendly Verification. IEEE Transactions on Computers, Jan. 2002, Vol. 51, Nr.1, ISSN 0018-9340, pp. 61-83. Preliminary materials also appears in proceedings of IEEE HASE'98, RTCSA'98, and IFIP FORTE'99.