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 \( x \rightarrow y \rightarrow z \rightarrow w \). 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 $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n$. 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.

```c
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 $*\text{exp}\_\text{type}$ pointing to the parsing tree for $x + y$, then we can write an indirect reference chain like $e \rightarrow \text{lhs} \rightarrow \text{atom} \rightarrow \text{name}$ to reference to the string of “$x$”. /label1: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 \( x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n = 3; \) 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.

```c
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

effect 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 $y \rightarrow x = 3y \rightarrow x + z$; 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 available for free at

  \[
  \text{http://www.cc.ee.ntu.edu.tw/~farn/red/}
  \]

  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 equivalence 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\cite{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
certainty 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 \emph{discrete variables} of pre-
declared finite integer value ranges. For each declared variable \( x \), \( \text{lb}(x) \) and \( \text{ub}(x) \) 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, \texttt{FALSE} is interpreted as 0 while \texttt{TRUE}
as 1. The second type of variables are \emph{pointers (address variables)} which point to processes
(data structures). The values of pointers range from zero (\texttt{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 \texttt{NULL}, which in C’s tradition is equal to zero.
Or in the same notations as of discrete variables, \( \text{lb}(x) = \texttt{NULL} \) and \( \text{ub}(x) \) is the number of
processes for all declared pointers \( x \).

Variables which are declared to be \emph{global} are accessible to all processes. \emph{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 $(G^d, G^p, I^d, I^p, A(P))$ where $G^d$ and $I^d$ are respectively the sets of global and local discrete variables, $G^p$ and $I^p$ 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 $\eta$ of $X^d$ and $X^p$ can be used to describe the triggering condition of state transitions and has the following syntax.

$$\eta ::= \epsilon_1 \sim \epsilon_2 \mid \neg \eta \mid \eta_1 \lor \eta_2$$
$$\epsilon ::= c \mid \text{NULL} \mid P \mid x \mid y \rightarrow \epsilon \mid x[p] \mid y[p] \rightarrow \epsilon \mid \epsilon_1 \oplus \epsilon_2$$

where $\sim \in \{"\leq", "\lt", "\geq", "\gt", "\geq"\}$ 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 $\oplus \in \{+,-,\ast,/\}$.

Here the notations $y \rightarrow \epsilon$ and $y[p] \rightarrow \epsilon$ for indirect references should be self-explaining to C-programmers. In the expression of $y \rightarrow \epsilon$, $y$ is either a local or global pointer variable that points to a structure and $\epsilon$ 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 $y[p] \rightarrow \epsilon$, we specifically refer to the local pointer $y$ of process $p$.

Parenthesis can be used for disambiguation. Traditional shorthands are $\epsilon_1 !\leq \epsilon_2 \equiv \neg(\epsilon_1 = \epsilon_2)$, $\eta_1 \land \eta_2 \equiv \neg((\neg \eta_1) \lor (\neg \eta_2))$, and $\eta_1 \Rightarrow \eta_2 \equiv (\neg \eta_1) \lor \eta_2$. 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:

$$y_1 \rightarrow y_2 \rightarrow \ldots \rightarrow y_n \rightarrow x = \epsilon;$$
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 $(Q, Q_0, E, \tau, \pi)$ 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 $\text{mode} \in L^d$ 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 $\text{mode}$ 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 $\nu$ of $S^\mathcal{M}$ is a mapping from

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

such that

- $\nu(\text{null}, x) = \perp$ (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 $\nu$, 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 $x \in G^p \cup L^p$, $\nu(p, x) \in \{\text{null}\} \cup \{1, \ldots, \mathcal{M}\}$ such that for all $1 \leq p' \leq \mathcal{M}$, $\nu(p, x) = \nu(p', x)$.
Given a state \( \nu \), 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 \( \eta \) at \( \nu \) to \( \{ \text{TRUE}, \text{FALSE}, \bot \} \) as follows.\(^1\)

- \( \nu(p, y \rightarrow \epsilon) = \nu(\nu(p, y), \epsilon) \) if \( p \neq \text{NULL} \).
- \( \nu(p, y|c| \rightarrow \epsilon) = \nu(c, y|\rightarrow \epsilon) \) if \( 1 \leq c \leq \mathcal{M} \); otherwise, \( \nu(p, y|c| \rightarrow \epsilon) = \bot \).
- \( \nu(p, \epsilon_1 \oplus \epsilon_2) = \bot \) if \( \oplus = '+' \lor \nu(p, \epsilon_2) = 0 \).
- \( \nu(p, \epsilon_1 \oplus \epsilon_2) = \nu(p, \epsilon_1) \oplus \nu(p, \epsilon_2) \) if either \( \oplus \in \{ +, -, \ast \} \) or \( \oplus = '+' \land \nu(p, \epsilon_2) \neq 0 \).

Integer-division is assumed, that is \( x/y \) is defined as \( \frac{x \text{ mod } y}{|x/y|} \).
- \( \nu(p, \epsilon_1 \oplus \epsilon_2) = \bot \) if \( \epsilon_1 = \bot \lor \epsilon_2 = \bot \).
- \( \nu(p, \epsilon_1 \sim \epsilon_2) = \nu(p, \epsilon_1) \sim \nu(p, \epsilon_2) \)
- “\( \sim \)” equals \( \bot \), and “\( \epsilon \sim \bot \)” equals \( \bot \).
- The negation of the satisfaction mapping is defined as

\[
\begin{array}{|c|c|c|}
\hline
\nu(p, \eta) & \text{FALSE} & \bot & \text{TRUE} \\
\hline
\nu(p, \neg \eta) & \text{FALSE} & \bot & \text{TRUE} \\
\hline
\end{array}
\]

- The disjunction of the satisfaction mapping is defined as

\[
\begin{array}{|c|c|c|}
\hline
\nu(p, \eta_1 \lor \eta_2) & \text{FALSE} & \bot & \text{TRUE} \\
\hline
\text{FALSE} & \text{FALSE} & \bot & \text{TRUE} \\
\bot & \bot & \bot & \bot \\
\text{TRUE} & \text{TRUE} & \bot & \text{TRUE} \\
\hline
\end{array}
\]

Given an action \( \alpha \) 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 \( \nu \), written \( \text{next}(p, \nu, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon) \), is defined as follows.

- When \( \nu(p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x) \neq \bot \) and \( \nu(p, \epsilon) \neq \bot \), \( \text{next}(p, \nu, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon) \) is exactly \( \nu \) except \( \text{next}(p, \nu, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon)(\nu(p, y_1 \rightarrow \ldots \rightarrow y_n), x) = \nu(p, \epsilon) \).
- When either \( \nu(p, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x) = \bot \) or \( \nu(p, \epsilon) = \bot \), \( \text{next}(p, \nu, y_1 \rightarrow \ldots \rightarrow y_n \rightarrow x := \epsilon) \) 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 \( \text{next}(p, \nu, \alpha_1 \alpha_2 \ldots \alpha_n) = \text{next}(\text{next}(p, \nu, \alpha_1), p, \alpha_2 \ldots \alpha_n) \). An initial state \( \nu_0 \) of an implementation \( S^M \) must satisfy \( \bigwedge_{1 \leq p \leq \mathcal{M}} \nu_0(p, \text{mode}) = 0 \). 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 \( \nu \) 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 \( \{ \text{TRUE}, \text{FALSE}, \bot \} \).
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 \( S^M \) is a (finite or infinite) sequence \( \rho = \nu_0\nu_1 \ldots \nu_k \ldots \) 
of states such that for all \( k \geq 0 \),

- \( \nu_0 \) is an initial state of \( S^M \); and
- for each \( \nu_k \) with \( k > 0 \), either \( \nu_k = \nu_{k-1} \) or there is a \( p \in \{1, \ldots, M\} \) and a transition 
  from \( q \) to \( q' \) such that \( \nu_{k-1}(p, \tau(q, q')) = \text{TRUE} \) and \( \text{next}(p, \nu_{k-1}, \pi(q, q')) = \nu_k \) 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.

\[
\begin{align*}
\phi & ::= \psi_1 \sim \psi_2 \mid \neg \phi \mid \phi_1 \lor \phi_2 \\
\psi & ::= c \mid \text{NULL} \mid y \mid x[p] \mid z \rightarrow \epsilon \mid w[p] \rightarrow \epsilon \mid \psi_1 \oplus \psi_2
\end{align*}
\]

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 \( 1 \leq p \leq M \).

Given a state \( \nu \) and a global predicate \( \phi \), we define the valuation of \( \nu \) on \( \phi \), written \( \nu(\phi) \), 
in the following inductive way.\(^2\)

- \( \nu(\psi_1 \sim \psi_2) = \nu(\psi_1) \sim \nu(\psi_2) \in \{\text{TRUE}, \text{FALSE}\} \)
- \( \nu(x[p]) = \nu(p, x) \)
- \( \nu(\neg \phi) = \neg \nu(\phi) \)
- \( \nu(\phi_1 \lor \phi_2) = \nu(\phi_1) \lor \nu(\phi_2) \)

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

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

**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 \( \text{SAP}(S, 2, \neg(6 \leq \text{mode}[1] \leq 9 \land 6 \leq \text{mode}[2] \leq 9)) \).

\(^2\)Again, we here overload the meaning of \( \nu() \) since now \( \nu(\phi) \) has only one parameter when \( \phi \) 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.

\[
\text{SAP}(S, \mathcal{M}, \phi) \{ \\
\quad \text{reachable} := \land_{1 \leq p \leq \mathcal{M}} \text{mode}[p] = 0; /* \text{the initial state-predicate} */ \\
\quad \text{next} := \text{TRUE}; \\
\quad \text{while}(\text{next} \neq \text{FALSE}) \{ \\
\quad \quad \text{next} := \text{FALSE}; \\
\quad \quad \text{Sequentially for each} \ 1 \leq p \leq \mathcal{M} \ \text{and for each transition} \ (q, q'), \ \text{do} \{ \\
\quad \quad \quad \text{new} := \text{indirect\_condition}(\text{reachable}, p, \tau(q, q')); \\
\quad \quad \quad \text{new} := \text{indirect\_assignment}(\text{new}, p, \pi(q, q')); \\
\quad \quad \quad \text{new} := \text{reduce}(\text{new}); /* \text{application of reduction techniques} */ \\
\quad \quad \quad \text{next} := \text{next} \lor (\text{new} \land \neg \text{reachable}); \\
\quad \quad \} \\
\quad \text{reachable} := \text{reachable} \lor \text{next}; \\
\quad \} \\
\quad \text{if} (\text{reachable} \land \phi \neq \text{FALSE}) \ \text{return} \ "\text{unsafe}"; \ \text{else return} \ "\text{safe}"; \\
\text{\} }
\]

The procedure iterates through the outer loop until \text{reachable} becomes a fix-point. At line (1), \text{indirect\_condition}(D, p, \eta) returns a global predicate in BDD representing the subspace of \(D\) in which \(\eta\) is true of process \(p\). At line (2), \text{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 \text{indirect\_condition()} and \text{indirect\_assignment()} will be discussed in section 5. At line (3), \text{reduce()} simplifies reachable state-space representations with various reduction techniques.

From its appearance, procedure \text{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, $1b(count) = 0$, and $ub(count) = 5$.

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

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

is true. The condition says that the difference of the count of the first process’s parent ($\text{parent}[1]\rightarrow\text{count}$) and twice the count of the right child of the second process’s left child ($2*\text{leftchild}[2]\rightarrow\text{rightchild}\rightarrow\text{count}$) is less than the count of process L ($L\rightarrow\text{count}$). 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()`.

```c
indirect_condition(D, p, \eta) {
    Without loss of generality, rewrite \eta into the form \epsilon \sim c where c is a constant
    Construct D_c := indirect_arith(p, \epsilon);
    return D \land \text{var_eliminate}(D_c \land \text{VALUE} \sim c, \text{VALUE});
}
```

Procedure `var_eliminate(D, x)` filters x out of D. For a local discrete variable x[p],
`var_eliminate(D, x[p]) = \nexists v \in \{x, y\} (D \land x[p] = v).` For a local pointer x[p],
`var_eliminate(D, x[p]) = \nexists v \in \{\text{NULL}, 1, \ldots, M\} (D \land 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, l_1\rightarrow \ldots \rightarrow l_k)` constructs the condition that the value of $l_1 \rightarrow \ldots \rightarrow l_k$ at process p equals the process identifier recorded in variable DPI.

```c
indirect_ref(p, i, l_1\rightarrow l_2\rightarrow \ldots \rightarrow 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 \nexists f \in M (l_i[j] = f \land \text{indirect_ref}(f, i + 1, l_1\rightarrow \ldots \rightarrow l_k));
    else if l_i is a local pointer $l_i$ with no specific process reference, then
        return \nexists f \in M (l_i[p] = f \land \text{indirect_ref}(f, i + 1, l_1\rightarrow \ldots \rightarrow l_k));
    else if l_i is a global pointer $g_i$, then
        return \nexists f \in M (g_i = f \land \text{indirect_ref}(f, i + 1, l_1\rightarrow \ldots \rightarrow 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 l₁→l₂→⋯→lₖ→x with k > 0, then {
        H := indirect_ref(p, 1, l₁→l₂→⋯→lₖ);
        for j := 1 to M, lb(x) ≤ v ≤ ub(x), do
            R := R ∨ (H ∧ var_eliminate(DPI == j ∧ x[j] == v ∧ VALUE == v, DPI));
    }
    else if ε is c, then
        R := R ∨ (VALUE == c);
    else if ε is x[i] with local variable x and specific process reference i, then
        for lb(x) ≤ v ≤ ub(x), do R := R ∨ (x[i] == v ∧ VALUE == v);
    else if ε is local variable x with no specific process reference, then
        for lb(x) ≤ v ≤ ub(x), do R := R ∨ (x[p] == v ∧ VALUE == v);
    else if ε is a global variable x, then
        for lb(x) ≤ v ≤ ub(x), do R := R ∨ (x == v ∧ VALUE == v);
    else if ε is ε₁ ⊕ ε₂ where ⊕ ∈ {+,-,*,/}, then {
        R₁ := indirect_arith(p, ε₁);
        R₂ := indirect_arith(p, ε₂);
        for every possible combination of values v₁, v₂ of variable VALUE, do {
            H₁ := var_eliminate(R₁ ∧ VALUE == v₁, VALUE);
            H₂ := var_eliminate(R₂ ∧ VALUE == v₂, VALUE);
            R := R ∨ (H₁ ∧ H₂ ∧ VALUE == v₁ ⊕ v₂); \hspace{1cm} (1)
        }
    }
    return R;
}
As an example, let us execute the just-mentioned procedures with \( \eta = l \rightarrow l \rightarrow x > l \rightarrow x + 3 \) on a state-space described by \( D = (l[1] == 2) \land (l[2] == 2) \land (x[1] == 4) \land (x[2] == 3) \). We first rewrite \( \eta \) to \( l \rightarrow l \rightarrow x - l \rightarrow x > 3 \). Suppose the current process identifier \( p = 1 \). Further, let \( lb(x) = 3 \) and \( ub(x) = 4 \). Then

\[
\text{indirect_ref}(p, 1, 1) = (l[1] == 1 \land \text{DPI} == 1) \lor (l[1] == 2 \land \text{DPI} == 2).
\]

Hence \( \text{indirect_arith}(p, l \rightarrow x) = (l[1] == 1 \land x[1] == 3 \land \text{VALUE} == 3) \lor (l[1] == 1 \land x[1] == 4 \land \text{VALUE} == 4) \lor (l[1] == 2 \land x[2] == 3 \land \text{VALUE} == 3) \lor (l[1] == 2 \land x[2] == 4 \land \text{VALUE} == 4) \).

Similarly, since \( \text{indirect_ref}(p, 1, l \rightarrow 1) = (l[1] == 1 \land l[1] == 1 \land \text{DPI} == 1) \lor (l[1] == 1 \land l[1] == 2 \land \text{DPI} == 2) \lor (l[1] == 2 \land l[2] == 1 \land \text{DPI} == 1) \lor (l[1] == 2 \land l[2] == 2 \land \text{DPI} == 2) \), we have

\[
\]

Thus, \( \text{indirect_arith}(p, 1, 1 \rightarrow l \rightarrow x - l \rightarrow x) \)

\[
\]

To evaluate an expression like \( \epsilon_1 \oplus \epsilon_2 \), the values recorded in the \text{VALUE} variable respectively in the symbolic predicates of \( \epsilon_1 \) and \( \epsilon_2 \) are pairwise compared and corresponding state-predicate conjuncted, as in line (1) in procedure \text{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

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

But this fails in two ways. First, there can be indirections in $\omega$. Second, the destination of $\omega$ can occur in $\epsilon$ 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.

\[
\begin{align*}
\text{indirect\_assignment}(D, p, \omega := \epsilon) \{ \\
\quad \text{Construct } D_\epsilon := D \land \text{indirect\_arith}(\epsilon, p); \\
\quad \text{if } \omega \text{ is } l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k \rightarrow x \text{ with } k > 0, \text{ then } \{ \\
\qquad \text{Let } R := \text{FALSE}; \\
\qquad \text{Construct } D'_\epsilon := D_\epsilon \land \text{indirect\_ref}(p, 1, l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k); \\
\qquad \text{for } j := 1 \text{ to } M, \text{ do } \{ \\
\qquad\qquad \text{Let } H := \text{var\_eliminate(var\_eliminate}(D'_\epsilon \land \text{DPI} == j, \text{DPI}), x[j]); \quad \text{(2)} \\
\qquad\qquad \text{Let } H := \text{condition\_effect}(x[j], \sim, H); \\
\qquad\qquad \text{Let } R := R \lor H; \\
\qquad \} \} \\
\quad \text{else if } \omega_0 \text{ is } x[i] \text{ with local variable } x \text{ with specific process reference } i, \text{ then} \\
\qquad \text{Let } R := \text{condition\_effect}(x[i], \sim, \text{var\_eliminate}(D_\epsilon, x[i])); \quad \text{(3)} \\
\quad \text{else if } \omega_0 \text{ is local variable } x \text{ with no specific process reference, then} \\
\qquad \text{Let } R := \text{condition\_effect}(x[p], \sim, \text{var\_eliminate}(D_\epsilon, x[p])); \quad \text{(4)} \\
\quad \text{else if } \omega_0 \text{ is a global variable } x, \text{ then} \\
\qquad \text{Let } R := \text{condition\_effect}(x, \sim, \text{var\_eliminate}(D_\epsilon, x)); \quad \text{(5)} \\
\quad \text{return } R;
\end{align*}
\]
condition_effect\(x, \sim, D\) {
    \(R := \text{FALSE};\)
    for every possible value \(v\) of variable \(\text{VALUE}\), do
    \(R := R \lor (x \sim v \land \text{var\_eliminate}(D \land \text{VALUE} = v, \text{VALUE}));\)
    return \(R \land \text{lub}(x) \leq x \leq \text{ub}(x)\);
}

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 \( \omega = \epsilon \) if either

- an indirect reference like \( y_1\rightarrow \ldots \rightarrow y_m\rightarrow y \) occurs in \( \omega \), \( p = p' \), and \( x = y_1 \); or
- an indirect reference like \( y_1\rightarrow \ldots \rightarrow y_m\rightarrow y \) occurs in \( \omega \) and \( x \in \{y_2, \ldots, y_m\} \); or
- an indirect reference like \( y_1\rightarrow \ldots \rightarrow y_m \) occurs in \( \epsilon \), \( p = p' \), and \( x = y_1 \); or
- an indirect reference like \( y_1\rightarrow \ldots \rightarrow y_m \) occurs in \( \epsilon \) 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.

```plaintext
base_oapprox_active(x)

let \( \eta_x := \text{FALSE} \);

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

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

then \( \eta_x := \eta_x \lor (\text{mode} == q \land \text{var_eliminate}(\tau(q, q'), x)) \).

else {

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

for each \( \Delta_i \), if \( x \) appears in \( \Delta_i \), then \( \eta_x := \eta_x \lor (\text{mode} == q \land \text{var_eliminate}(\Delta_i, x)) \).

}

return \( \eta_x \);
```

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.

\[
\text{oapprox\_active}(x) \{ \\
\eta_x := \text{base\_oapprox\_active}(x); \eta_x' := \text{FALSE}; \\
\text{while}(\eta_x' \neq \eta_x) \{ \\
\quad \eta_x' := \eta_x; \\
\quad \text{for each transition } (q, q') \text{ that is non-disrupting to } x \text{ in } A(P), \{ \\
\quad \quad \text{let } \eta_x' := \eta_x' \lor \delta, \text{ where } \delta \text{ is an over-approximation of} \\
\quad \quad \text{the weakest precondition of } \eta_x \text{ before applying } (q, q'). \quad (6) \\
\quad \} \} \\
\text{return } \eta_x. 
\}
\]

The function \( \text{oapprox\_active}_x(s) \) 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

\[
\begin{align*}
\text{active\_locked} &= 4 \leq \text{mode} \leq 5 \\
\text{active\_next} &= \text{mode} = 1 \lor (2 \leq \text{mode} \leq 4 \land \text{prev!}=P) \lor \text{mode} \geq 5 \\
\text{active\_prev} &= 1 \leq \text{mode} \leq 4
\end{align*}
\]

It shows that local variable \( \text{locked} \), for example, will not be read and thus affect the system behaviors outside local modes 4 and 5. The elimination of values of \( \text{locked} \) 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 $M = 3$, 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 $\Sigma$. If

- all $\sigma \in \Sigma$ satisfy $\sigma(p) = p$ on all process identifier $p$ that are mentioned anywhere in the program, and
- if $\nu$ 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 $\Sigma$ satisfies additionally that for all $\sigma \in \Sigma$ and all states $\nu$, $\nu$ satisfies the goal condition iff $\sigma\nu$ (the application of permutation $\sigma$ to $\nu$) 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 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 $\Sigma$ also having binary permutations $\sigma_{p_ip_j}$ 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 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 $\sigma_{p_ip_j}$ leaves the initial condition invariant.

**Symmetry reduction as sorting**

In particular, set $\Sigma$ is closed under composition and inversion. Moreover, such a group $\Sigma$ has a well-structured generating set—the set of all binary permutations $\sigma_{p_ip_j}$ 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 $\Sigma$ 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 $\text{reverse}(i, j)$ 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, $\text{reverse}()$ may actually lead to an over-approximation of the symmetry group $\Sigma$. Once predicate $\text{reverse}(i, j)$ 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 $\text{reverse}()$ to iteratively permute $\sigma$ to a “normalized” image in $\Sigma$.

```c
reduce_symmetry(\eta) {
    Sequentially for $i := 1$ to $\mathcal{M} - 1$, do
        Sequentially for $j := i + 1$ to $\mathcal{M}$,
            let $\eta := (\eta - \text{reverse}(i, j)) \lor \text{permute}(\eta \land \text{reverse}(i, j), i, j)$;
    return $\eta$;
}
```
Here $\text{permute}(\eta, i, j)$ is obtained from $\eta$ 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.

$\text{permute}(\eta, i, j)$ 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 $\text{reverse}(\cdot)$? 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 $\text{prev}$ 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 $\text{reverse}(i, j)$ does not use information on indirection paths of length $> 1$ between processes $i$ and $j$. This is for the complexity of the BDD for $\text{reverse}(i, j)$. In our experiment, if we consider indirection path lengths $> 1$ in the construction of $\text{reverse}(i, j)$, the sizes of BDDs become too big to represent and manipulate
efficiently. When \( \text{reverse}(i, j) \) are FALSE, it only means we have no rule to break the tie between processes \( i \) and \( j \). Procedure \text{reduce_symmetry}() does not break any ties either.

7 Implementation and Experiments

We implemented our techniques in a symbolic verification tool called \textbf{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 indispensable 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 \textbf{Red}, both with and without the symmetry reduction technique, with that of SMC[12] and Mur\( \phi \)[20] running in various options. Since neither SMC nor Mur\( \phi \) 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 \textit{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 \textbf{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 \textbf{Red} is collected only for BDDs and their management.

The verification algorithm of \textbf{Mur}\( \phi \) 3.1 is breadth first search with various symmetry algorithms including exhaustive canonicalization(-sym1), heuristic fast canonicalization(-sym2),
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 Murϕ 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.
<table>
<thead>
<tr>
<th>Tools</th>
<th>Options</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>-Isp</td>
<td>0.30s</td>
<td>2.14s</td>
<td>11.41s</td>
<td>64.43s</td>
<td>461.71s</td>
<td>2333.40s</td>
<td>5137.08s</td>
<td>1676.51s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2ks</td>
<td>68k</td>
<td>168k</td>
<td>460k</td>
<td>1311k</td>
<td>3894k</td>
<td>11107k</td>
<td>31220k</td>
</tr>
<tr>
<td></td>
<td>-f</td>
<td>3.37s</td>
<td>47.77s</td>
<td>1095.71s</td>
<td>6393.27s</td>
<td>Out of Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>121k</td>
<td>504k</td>
<td>3937k</td>
<td>28995k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMC</td>
<td>-s1</td>
<td>145.08s</td>
<td>C/D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s2</td>
<td>596.4s</td>
<td>&gt;17H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13472k</td>
<td>Unfinished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s3</td>
<td>600.3s</td>
<td>&gt;17H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13477k</td>
<td>Unfinished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s4</td>
<td>1601.8s</td>
<td>20252.6s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13460k</td>
<td>C/D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s5</td>
<td>1624.0s</td>
<td>&gt;17H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13457k</td>
<td>Unfinished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s6</td>
<td>1600.3s</td>
<td>&gt;17H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13459k</td>
<td>Unfinished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s7</td>
<td>1620.8s</td>
<td>&gt;17H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13457k</td>
<td>Unfinished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Murphi</td>
<td>-sym1</td>
<td>3.93s</td>
<td>28.51s</td>
<td>Internal Error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym2</td>
<td>2.77s</td>
<td>25.40s</td>
<td>Not Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym3</td>
<td>2.77s</td>
<td>25.37s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym4</td>
<td>2.89s</td>
<td>25.30s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 Murphi, an internal error occurs for the eight-process system. But for the seven-process system, Murphi 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
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 Murφ 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.
Table 4: Doubly Linked Cycle Operations

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: “\( \text{region } v \text{ when } B \text{ do } S \)”, 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.
### Table 5: Bounded Buffer Problem

<table>
<thead>
<tr>
<th>Tools</th>
<th>Options</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdd</td>
<td>-lp</td>
<td>1.84s</td>
<td>6.93s</td>
<td>26.48s</td>
<td>65.62s</td>
<td>254.07s</td>
<td>417.63s</td>
<td>1059.98s</td>
<td>538.85s</td>
<td>4151.73s</td>
<td>8321.84s</td>
<td>7777.01s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-f</td>
<td>1.22s</td>
<td>1.15s</td>
<td>24.63s</td>
<td>45.12s</td>
<td>62.29s</td>
<td>622.91s</td>
<td>9924.80s</td>
<td>8785.24s</td>
<td>Out of Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMC</td>
<td>-s1</td>
<td>0s</td>
<td>2s</td>
<td>18s</td>
<td>201s</td>
<td>Out of Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s2</td>
<td>0s</td>
<td>2s</td>
<td>23s</td>
<td>260s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s3</td>
<td>0s</td>
<td>2s</td>
<td>23s</td>
<td>264s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s4</td>
<td>0s</td>
<td>2s</td>
<td>23s</td>
<td>289s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Murphi</td>
<td>-sym1</td>
<td>3.12s</td>
<td>3.00s</td>
<td>3.32s</td>
<td>5.85s</td>
<td>21.57s</td>
<td>150.95s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym2</td>
<td>3.14s</td>
<td>3.03s</td>
<td>3.89s</td>
<td>5.63s</td>
<td>20.04s</td>
<td>140.57s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym3</td>
<td>3.09s</td>
<td>3.00s</td>
<td>3.81s</td>
<td>5.53s</td>
<td>20.04s</td>
<td>140.57s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym4</td>
<td>3.12s</td>
<td>3.10s</td>
<td>3.91s</td>
<td>5.68s</td>
<td>20.16s</td>
<td>141.51s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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 Murphi 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 Murphi) in large systems. This may suggest our technique is more scalable than those deployed in the other tools.
<table>
<thead>
<tr>
<th>Tools</th>
<th>Options</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>-Sp</td>
<td>1.2s</td>
<td>2.9s</td>
<td>23.7s</td>
<td>42.71s</td>
<td>276.5s</td>
<td>440.4s</td>
<td>1898.16s</td>
<td>2547.08s</td>
<td>6006.04s</td>
<td>9732.03s</td>
<td>23941.32s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>220k</td>
<td>244k</td>
<td>321k</td>
<td>415k</td>
<td>896k</td>
<td>996k</td>
<td>200k</td>
<td>2119k</td>
<td>3898k</td>
<td>4037k</td>
<td>6962k</td>
</tr>
<tr>
<td>-f</td>
<td>3.99s</td>
<td>5.11s</td>
<td>25.041s</td>
<td>103.94s</td>
<td>188.732s</td>
<td>21179.906s</td>
<td>106175.88s</td>
<td>Out of Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>220k</td>
<td>244k</td>
<td>1106k</td>
<td>2154k</td>
<td>9932k</td>
<td>22906k</td>
<td>71049k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMC</td>
<td>-s1</td>
<td>0s</td>
<td>1s</td>
<td>26s</td>
<td>238s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td>Out of Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>31s</td>
<td>304s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s2</td>
<td>0s</td>
<td>2s</td>
<td>31s</td>
<td>219s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>33s</td>
<td>311s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49550k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s3</td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49550k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s4</td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49550k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s5</td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49550k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s6</td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49550k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-s7</td>
<td>0s</td>
<td>2s</td>
<td>35s</td>
<td>331s</td>
<td>96k</td>
<td>620k</td>
<td>6602k</td>
<td>49549k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Murph</td>
<td>-sym1</td>
<td>3.3s</td>
<td>3.0s</td>
<td>3.6s</td>
<td>5.37s</td>
<td>44.76s</td>
<td>287.14s</td>
<td>Internal Error</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.38s</td>
<td>3.0s</td>
<td>3.58s</td>
<td>5.22s</td>
<td>42.24s</td>
<td>276.76s</td>
<td>Not Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym2</td>
<td>3.38s</td>
<td>3.0s</td>
<td>3.58s</td>
<td>5.22s</td>
<td>42.24s</td>
<td>276.76s</td>
<td>Not Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.20s</td>
<td>2.9s</td>
<td>3.57s</td>
<td>5.17s</td>
<td>42.23s</td>
<td>274.04s</td>
<td>Not Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-sym3</td>
<td>3.34s</td>
<td>3.0s</td>
<td>3.62s</td>
<td>5.25s</td>
<td>42.17s</td>
<td>276.14s</td>
<td>Not Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Tools</th>
<th>Options</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No termination</td>
</tr>
<tr>
<td>SMC</td>
<td>-sym1</td>
<td>4.00s</td>
<td>7.02s</td>
<td>49.75s</td>
<td>Internal</td>
<td>Not</td>
</tr>
<tr>
<td></td>
<td>sym2</td>
<td>4.00s</td>
<td>6.78s</td>
<td>46.09s</td>
<td>Error</td>
<td>Available</td>
</tr>
<tr>
<td></td>
<td>sym3</td>
<td>4.01s</td>
<td>6.75s</td>
<td>46.01s</td>
<td></td>
<td>Available</td>
</tr>
<tr>
<td></td>
<td>sym4</td>
<td>3.97s</td>
<td>6.80s</td>
<td>46.00s</td>
<td></td>
<td>Available</td>
</tr>
</tbody>
</table>

Table 7: Conditional Region Construct

References


