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

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

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

Abstract

We formally define the model of software with pointer data structures. We developed symbolic algorithms for the manipulation of conditions and assignments with indirect operands for verification with BDD-like data-structures. We rely on two techniques, including inactive variable elimination and process-symmetry reduction in the data-structure configuration, to contain the time and memory complexity. We use binary permutation for efficiency but also identify the possibility of anomaly of image false reachability. We implemented the techniques in tool `red` and compare performance with Murϕ and SMC against several other benchmarks.

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

1 Introduction

Verification of networks with special topologies like rings and buses has been widely studied. In real-world software, arbitrary and dynamic network configuration is, however, often constructed using pointers. An action like "$x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n := \ldots;$" can stretch through a network and change the local memory of a peer process in the network. Such indirect references are not only

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

†supported by DARPA/ITOP within the MoBIES project

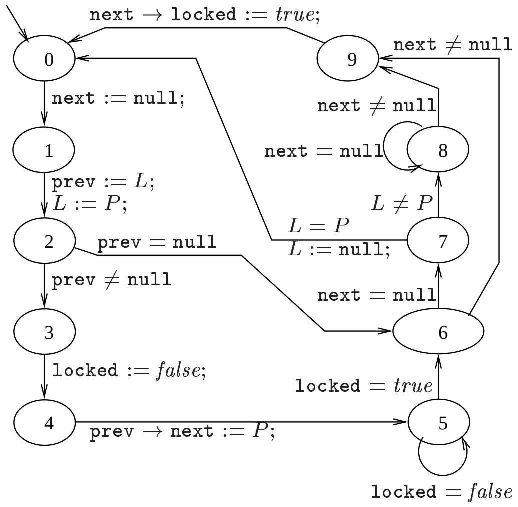

Figure 1: MCS locking algorithm

very common in practice, but also extremely important in both hardware and software engineering. For example, most CPUs now support hardware indirect addressing to facilitate virtual memory management. This important hardware indirect referencing mechanism is transparent to softwares and runs silently. For another example, dynamic data-structures like linear lists, trees, and graphs are constructed with pointers and used intensively in most nontrivial softwares. In example 1, we have a locking algorithm[12] which uses pointers to maintain a queue for critical section mutual exclusion.

Example 1 *:* **MCS (Mellor-Crummy & Scott's) locking algorithm** The algorithm[12] is an example protocol in which a global waiting queue of processes is explicitly used to insure mutual exclusive access to the critical section in a concurrent system. In figure 1, The MCS locking algorithm for a process is drawn as a finite-state automaton. The critical section is from modes 6 to 9. The queue is constructed with one global pointer `L` (to the tail of the queue), and two local pointers of each process: `next` and `prev` (respectively to the successor and predecessor processes of the local process in the queue). P is a special symbol for the data structure address of the running process. Each process also has a local Boolean variable `locked` which is set to false when the process is permitted to the critical section by its predecessor in the queue. We want to guarantee that at any moment, at most one process is in its critical section. $\|$

Due to their dynamic nature, software with pointer data-structures has been known to be extremely difficult to maintain and to debug. For example, any experienced software engineers will agree that bugs caused by dirty pointers to freed data-structures are extremely difficult to detect and remove. Such bugs, whose effect usually does not emerge until long after a data-structure is corrupted through a dirty pointer, is very difficult to trace backward.

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

- We define a formal model for concurrent software with pointer data structures for rigorous research on solution for the problem. Especially note that the framework is defined in such a way to explicitly allow all processes to share the same automaton template but at the same time allow them to use their local variables. This is extremely important in identifying process-symmetry in a convincing way. Most other model-checkers[3, 4, 10, 16] accept that each process be described with its own automaton and usually create difficulty in efficiently identifying symmetric behaviors among the process automata. (Note that asymmetric systems can also be modeled in our framework with processes running mutually disjoint parts of the program.)
- We present algorithms for the symbolic manipulation of conditions and assignments with indirect operands for verification with BDD-like data-structures. Algorithms for both forward analysis and backward analysis have been developed and implemented with tuning for verification performance. Special care is taken to allow for recurrence assignments, like $y \rightarrow x := 3y \rightarrow x + z$; where the left-hand-side may also occur in the right-hand-side.
- We adapt two reduction techniques for model checking such systems.
  - *Reduction by inactive variables eliminations,* which helps the construction of concise state-space representation through the elimination of variable valuations that do not affect the system behaviors[4, 10, 14, 15, 16]. Due to the implicit reading of pointer values in the indirection of operand references, the adaptation is not so trivial.
  - *Reduction by process-symmetry.* The idea of symmetry reduction [6, 8, 16, 13, 14] is to keep only one state if two states turn out to be symmetric. We shall follow the approach of process-symmetry in [8, 14, 16] since process represents a typical basic unit for behavioral equivalence in symmetry. For efficiency, we use binary permutation of process identifiers to transform data-structures to their isomorphic ones with process re-indexing. How to design a symbolic predicate to detect the necessary condition for permutation is also discussed.
- We also implemented our modelling and verification techniques for pointer data-structure systems in our model-checker `red` version 3.1, which is

available freely at

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

for timed automata. The implementation not only support pointer data-structures but also support complex arithmetics on process identifiers.

- We carried out experiments on several benchmarks to show the usefulness of our techniques and compare performance with SMC[9] and Murϕ[7, 11].

In section 2, we shall define the formal framework of this research. In section 3, we shall present the algorithmic framework to integrate safety-analysis software with various reduction techniques. In section 4, we shall present the algorithm for the manipulation of symbolic predicates and symbolic assignment statements with BDD-like data-structures. In section 5, we shall discuss our reduction techniques. In section 6, we discuss our implememntations and experiments. Section 7 is the conclusion.

2 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. We shall have the convention that if p is the address of a process's data-structure, then the process is also named p. But our model and techniques can be easily adapted for the modelling and verification of systems with data-structure addresses not bound to process identifiers.

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

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

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

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

2.1 Syntax of algorithm descriptions

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

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

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

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

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

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

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

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

2.2 Computation of systems

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

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

such that

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

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

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

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

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

$\nu(p, \eta_1 \vee \eta_2)$	*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 $y_1 \to \ldots \to y_n \to x := \epsilon;$, with $n \geq 0$, to p at ν, written $\mathrm{next\_state}(p, \nu, y_1 \to \ldots \to y_n \to x := \epsilon;)$, is defined as follows:

- When $\nu(p, y_1 \to \ldots \to y_n \to x) \neq\perp$ and $\nu(p, \epsilon) \neq\perp$, $\mathrm{next\_state}(\nu, p, y_1 \to \ldots \to y_n \to x := \epsilon;)$ is identical to ν except that $\mathrm{next\_state}(\nu, p, y_1 \to \ldots \to y_n \to x := \epsilon;)(\nu(p, y_1 \to \ldots \to y_n), x) = \nu(p, \epsilon)$.

- When either $\nu(p, y_1 \to \ldots \to y_n \to x) = \perp$ or $\nu(p, \epsilon) = \perp$, $\text{next\_state}(\nu, p, y_1 \to \ldots \to y_n \to x := \epsilon;)$ is undefined.

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

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

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

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

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

2.3 Safety analysis problem

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

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

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 $1 \le p \le \mathcal{M}$.

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

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

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

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

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

3 Framework for safety analysis and reductions

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

SAP($S, \mathcal{M}, \phi$) {
  $reachable := \bigwedge_{1 \leq p \leq \mathcal{M}} (\nu_0(p, \texttt{mode}) = 0)$; /* the initial state-predicate */
  $next := true$;
  Loop until $next = false$, do {
    $next := false$;
    Sequentially for each $1 \leq p \leq \mathcal{M}$ and for each transition (q, q'), do {
      $new :=$ `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)
      $next := next \cup (new - reachable)$;
    }
    $reachable := reachable \cup next$;
  }
  if ($reachable \wedge \phi \neq false$) return "unsafe"; else return "safe";
}

The procedure iterates through the outer loop until *reachable* becomes a fixpoint. At line (1), `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 4. At line (3), reduce() is about application of various reduction techniques to control the complexity of reachable state-space representations.

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

4 Manipulation of predicates with indirections

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

4.1 Symbolic evaluation of conditions with indirect operands

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

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

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

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

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

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

$\texttt{indirect\_condition}(D, p, \eta)$ {
  Collect the operands $\omega_0, \ldots, \omega_n$ used in η;
  Rewrite η into η' in the form of $\omega_0 \sim \epsilon$.
  Construct $D_\epsilon := D \wedge \texttt{indirect\_arith}(\epsilon, p)$;
  if ω_0 is $h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k \rightarrow x$ with $k > 0$, then {
    Let $R := \mathit{false}$;
    Construct $D_{\omega_i} := D_\epsilon \wedge \texttt{indirect\_ref}(h_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k, p)$;
    for $j := 1$ to $\mathcal{M}$,
      $R := R \vee \texttt{condition\_effect}(x[j], \sim, \texttt{var\_eliminate}(D_{\omega_i} \wedge (\texttt{PI} = j), \texttt{PI}))$;
  }

```
  else if ω0 is x[i] with local variable x with specific process reference i, then
    Let R := condition_effect(x[i], ∼, D_ε);
  else if ω0 is x with local variable x with no specific process reference, then
    Let R := condition_effect(x[p], ∼, D_ε);
  else if ω0 is a global variable x, then
    Let R := condition_effect(x, ∼, D_ε);
  return R;
}
```

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

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

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

```
indirect_ref(h_i → l_{i+1} → ... → l_k, p) {
  if i ≥ k, return(PI = p);
  Let R := false;
  if h_i is a local pointer l_i[j] with specific process reference j, then
    for f := 1 to M, do
      R := R ∨ (l_i[j] = f ∧ indirect_ref(l_{i+1} → ... → l_k, f));
  else if h_i is a local pointer l_i with no specific process reference, then
    for f := 1 to M, do
      R := R ∨ (l_i[p] = f ∧ indirect_ref(l_{i+1} → ... → l_k, f));
  else if h_i is a global pointer g_i, then
    for f := 1 to M, do R := R ∨ (g_i = f ∧ indirect_ref(l_{i+1} → ... → l_k, f));
  return(R);
```

}

Function `indirect_arith`(ϵ, p) uses the auxiliary variable `VALUE` to symbolically record the value of expression ϵ at process p. It returns the state-predicate with `VALUE` to the value of expression ϵ for process p at every state.

`indirect_arith`(ϵ, p) {
  $R := \mathit{false}$;
  if ϵ is $h_1 \to l_2 \to \ldots \to l_k \to x$ with $k > 0$, then {
    $H :=$ `indirect_ref`$(h_1 \to l_2 \to \ldots \to l_k, p)$;
    for $j := 1$ to $\mathcal{M}$, `lb`$(x) \leq v \leq$ `ub`(x), do
      $R := R \vee ($`var_eliminate`$(H \wedge \texttt{PI} = j, \texttt{PI}) \wedge x[j] = v \wedge \texttt{VALUE} = v)$;
  }
  else if ϵ is $x[i]$ with local variable x and specific process reference i, then
    for `lb`$(x) \leq v \leq$ `ub`(x), do $R := R \vee (x[i] = v \wedge \texttt{VALUE} = v)$;
  else if ϵ is local variable x with no specific process reference, then
    for `lb`$(x) \leq v \leq$ `ub`(x), do $R := R \vee (x[p] = v \wedge \texttt{VALUE} = v)$;
  else if ϵ is a global variable x, then
    for `lb`$(x) \leq v \leq$ `ub`(x), do $R := R \vee (x = v \wedge \texttt{VALUE} = v)$;
  else if ϵ is $\epsilon_1 \oplus \epsilon_2$ where $\oplus \in \{+, -, *, /\}$, then {
    $R_1 :=$ `indirect_arith`(ϵ_1, p);
    $R_2 :=$ `indirect_arith`(ϵ_2, p);
    for every possible value v_1, v_2 of variable `VALUE`, do {
      $H_1 :=$ `var_eliminate`$(R_1 \wedge \texttt{VALUE} = v_1, \texttt{VALUE})$;
      $H_2 :=$ `var_eliminate`$(R_2 \wedge \texttt{VALUE} = v_2, \texttt{VALUE})$;
      $R := R \vee (H_1 \wedge H_2 \wedge \texttt{VALUE} = v_1 \oplus v_2)$; (1)
    }
  }
  return R;
}

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

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

}

4.2 Symbolic assignments with indirect operands

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

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

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

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

$\texttt{indirect\_assignment}(D, p, \omega_0 := \epsilon;)$ {
  Construct $D_\epsilon := D \wedge \texttt{indirect\_arith}(\epsilon, p)$;
  if ω_0 is $h_1 \to l_2 \to \ldots \to l_k \to x$ with $k > 0$, then {
    Let $R :=$ *false*;
    Construct $D_{\omega_i} := D_\epsilon \wedge \texttt{indirect\_ref}(h_1 \to l_2 \to \ldots \to l_k, p)$;
    for $j := 1$ to $\mathcal{M}$, do {
      Let $H := \texttt{var\_eliminate}(\texttt{var\_eliminate}(D_{\omega_i} \wedge (\texttt{PI} = j), \texttt{PI}), x[j])$; (2)
      Let $H := \texttt{condition\_effect}(x[j], \sim, \texttt{var\_eliminate}(H, x[j]))$;
      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;
}

---

5 Reduction techniques

There are two reduction techniques we used to battle the state-space explosion problem. Due to page-limit, we give a brief description of them.

5.1 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. Such a technique has been used heavily in tools like Spin[10], UPPAAL[4], SGM[16], and `red`[14, 15]. 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 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. By applying our technique to the MCS algorithm in figure 1, we find that

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

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

5.2 Symmetry reduction

We follow the reduction framework in [8] to permute process identifiers to take advantage of the symmetry among processes running different copies of the

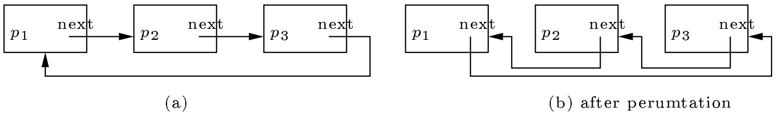

Figure 2: Anomaly of image false reachability

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 how to design an efficient symmetry reduction strategy. For m processes, we have $m!$ different permutations and obvioously we do not want to try them all to find the best permutation. Our technique toward this challenge 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 like 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 the 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.

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 $\overline{\sigma}\nu$ does, then the reduced set of states intersects with the goal condition if and only if the original transition system does. ||

That is, under these conditions we can replace the original transition system with the reduced one for solving a safety analysis problem 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 transpositions $\sigma_{p_i p_j}$ 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 optimal BDD to check whether this exchange of roles between p_i and p_j leads to the same initial condition, leads to the same initial condition. If this is the case then $\sigma_{p_i p_j}$ leaves the initial condition invariant.

Symmetry reduction as sorting

In particular, this set is closed under composition and inversion. Moreover, such a group Σ has a well-structured generating set—the set of all transpositions $\sigma_{p_i p_j}$ 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 transpositions until some kind of "lexicographically" smallest state is achieved. Since this process resembles usual sorting procedures, it yields a unique, smallest member of the equivalence class of the original state in polynomial time. Thus, this procedure can be used to efficiently solve the problem of how to construct the reduced transition system.

Criterion for binary permutation

With binary permutation, we have to design a predicate $\texttt{reverse}(i, j)$, for each $1 \leq i < j \leq m$ which characterizes those data-structure configurations in which processes i and j have to be permuted. This give us the second challenge, namely, how to design a criterion to determine when we need to permute two

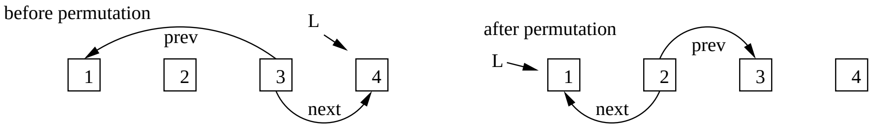

Figure 3: permutation of process identifiers

process identifiers. 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 of processes i and j are 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.

6 Implementation and experiments

We implemented our techniques in a symbolic verification tool called `red`[14, 15], 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. 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[9] and Murϕ[11] running in various options. Since neither SMC and Murϕ supports pointers, we have to use arrays in them to encode the pointers.

We have tested our implementation with three benchmarks. The first is the MCS locking algorithm whose `red` code can be found in our tool website. The safety condition to check is that no two processes are in the critical section at the same time. (Please check example 2.)

The second benchmark is a leader-election alogorithm. Each process has a local pointer `parent`. A set of symmetric processes randomly request to be a child of another process, who is not yet somebody's child. The process responds to a request will write its identifier to a global pointer `respond_id`. Then the requesting process will write the content of `respond_id` to its local variable `parent`. The processes use their variables `parent` to construct a dynamic forest structure. We want to make sure that at any time, at least some process is not somebody's child.

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

The performance data table is in table 1. The first row of columns 3 to 11 are for the numbers of processes in the implementation. All the data is collected on a Pentium III 800MHz running Linux with 256MB. All data of `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 three benchmarks represent three different types of dynamic data-structures: doubly-linked queues, doubly-linked cycles, and forests with arbitrary number of children of each internal nodes. According to the data, only Murϕ outperforms `red` in one benchmark, the double-cycle insertions which generates the simplest data-structure configurations in all three benchmarks.

For the leader-election benchmarks, we found that our symbolic techniques is so powerful that the safety property is verified in our quotient-structure BDD without ever having to invoking the symmetry reduction. But for the other two benchmarks, symmetry becomes indispensible.

7 Conclusion

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

References

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

benchmarks	Tools	Options ?	3	4	5	6	7	8	9
MCS	red	-fSp	0.98s	5.25s	27.09s	141.56s	928.26s	8584.44s	115072.26s
			48k	131k	397k	1064k	3299k	22824k	197599k
		-f	2.33s	23.13s	291.21s	5442.33s	Not available		
			139k	1125k	12776k	234497k			
	SMC	-s1	145.0 s	Not available					
			C/D						
		-s2	596.4s	>17h	Not available				
			13472k	N/F					
		-s3	600.3s	>17h	Not available				
			13477k	N/F					
		-s4	1601.8s	20252.6s	Not available				
			13460k	C/D					
		-s5	1624.0s	>17h	Not available				
			13457k	N/F					
		-s6	1600.3s	>17h	Not available				
			13459k	N/F					
		-s7	1620.8s	>17h	Not available				
			13457k	N/F					
	Murphi	-sym1	0.18 s	3.89 s	371.83 s	T/M	Not available		
		-sym2	0.19 s	3.54 s	179.72 s	T/M	Not available		
		-sym3	0.18 s	3.27 s	142.47 s	T/M	Not available		
		-sym4	0.18 s	3.29 s	148.04 s	T/M	Not available		
leader election	red	-fSp	0.02s	0.05s	0.10s	0.16s	0.25s	0.35s	0.54s
			17k	38k	69k	113k	171k	246k	337k
		-f	0.02s	0.05s	0.10s	0.16s	0.25s	0.35s	0.54s
			17k	38k	69k	113k	171k	246k	337k
	SMC	-s1	0.3s	0.5s	0.3s	0.7s	4.8s	62.7s	2096.4s
			1k	7k	34k	193k	1224k	8444k	68240k
		-s2	0.2s	0.2s	0.4s	2.4s	42.7s	1097.2s	34511.2s
			1k	7k	29k	135k	681	3707k	21799k
		-s3	0.2s	0.2s	0.4s	2.3s	29.5s	944.1s	16335.1s
			1k	7k	28k	134k	567	3451k	14964k
		-s4	0.2s	0.4s	0.4s	1.4s	9.2s	73.7s	619.0s
			1k	6k	19k	62k	196	604k	1857
		-s5	0.3s	0.3s	0.4s	1.3s	8.9s	70.4s	591.0s
			1k	6k	19k	62k	195	602k	1851
		-s6	0.3s	0.3s	0.5s	1.4s	9.1s	73.3s	621.0s
			1k	6k	19k	62k	196	604k	1857
		-s7	0.2s	0.3s	0.4s	1.3s	8.8s	70.7s	592.6s
			1k	6k	19k	62k	195	602k	1851
	Murphi	-sym1	0.1s	0.1s	0.2s	2.81s	T/M	Not available	
		-sym2	0.1s	0.11s	0.18s	2.56s	T/M	Not available	
		-sym3	0.1s	0.12s	0.19s	2.57s	T/M	Not available	
		-sym4	0.1s	0.1s	0.2s	2.58s	T/M	Not available	
double cycle insertion	red	-fSp	0.06s	0.26s	0.95s	3.59s	14.86s	59.65s	228.72s
			16k	38k	98k	254k	618k	1462k	3418k
		-f	0.93s	67.89s	>15m	Not available			
			165k	10073k	N/F				
	SMC	No termination							
	Murphi	-sym1	0.1s	0.1s	0.11s	0.29s	2.13s	24.36s	290.26s
		-sym2	0.1s	0.1s	0.11s	0.16s	0.53s	5.99s	66.59s
		-sym3	0.1s	0.1s	0.11s	0.11s	0.12s	0.15s	0.17s
		-sym4	0.1s	0.1s	0.1s	0.11s	0.1	0.13	0.15

s (m, h): CPU time in seconds (minunts, hours); k: Memory in kilobytes;
C/D: core dumped; T/M: internal error, too many states; N/F: not finished; I/E: internal error

Table 1: Performance data table of three benchmarks

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

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

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

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

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

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

[8] 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.

[9] 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

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

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

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

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

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

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

[16] 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 of this paper also appears in proceedings of IEEE HASE'98, RTCSA'98, and IFIP FORTE'99.