Efficient and User-Friendly Verification

Farn Wang and Pao-Ann Hsiung, *Member*, *IEEE*

Abstract—A compositional verification method from a high-level resource-management standpoint is presented for dense-time concurrent systems and implemented in the tool of SGM (State-Graph Manipulators) with graphical user interface. SGM packages sophisticated verification technology into state-graph manipulators and provides a user interface which views state-graphs as basic data-objects. Hence, users do not have to be verification theory experts and do not have to trace inside state-graphs to analyze state and path properties to make the best use of verification theory. Instead, users can construct their own verification strategies based on observation on the state-graph complexity changes after experimenting with some combinations of manipulators. Moreover, SGM allows users to control the complexity of state-graphs through iterative state-graphs merging and reductions before they become out of control. Reduction techniques specially designed for the context of state-graph iteration composition and shared variable manipulations are developed and used in SGM. Experiments on different benchmarks to show SGM performance are reported. An algorithm based on group theory to pick a manipulator combination is presented.

Index Terms—Verification, compositional verification, model-checking, real-time systems, timed automata, formal methods, software engineering.

1 Introduction

The general trend in engineering is to package complex technology with simple and friendly interfaces so that more users can benefit. Since the famous Pentium-bug, people have been anticipating wide acceptance of the technology of computer-aided verification. Indeed, with today's powerful hardware and recently reported verification theory breakthroughs [3], [8], it seems that industrial application of verification theory is becoming more and more real. But most verification packages today are developed based on profound, complex theories that take years of graduate study to master. Thus, inevitably, only projects with big budgets can afford the advantage of computer-aided verification. One of the goals of this work is to devise a packaging scheme for verification technologies so that users illiterate in verification technology can still benefit from it.

Here, we give a brief description of our method which works on dense-time concurrent systems. For a system with m concurrent processes, we assume that we are given their m behavior descriptions, called *state-graphs*, stored in an array, $G[1], \ldots, G[m],$, respectively. In the traditional approach, a verification procedure will start by constructing the Cartesian product of $G[1], \ldots, G[m]$, as in Table 1a with eight processes, to verify the given concurrent system. The standard technology now is symbolic manipulation [8], [19], [4], which can be used in both Cartesian product calculation and model-checking. To cope with the resource consumption requirement from different verification tasks, very often ingenious strategies which search through state-graphs for certain state and path properties have to be devised to keep space and CPU-time under control. To this end, users have to be knowledgeable about the theory of computer-aided verification and traverse through the final product state-graph G.

On the contrary, our method treats state-graphs as high-level data-objects and defines and implements many theoretically proven manipulators to merge, reduce, and check them. From the users' standpoint, the goal is to construct a verification procedure, with the many state-graph manipulators in our library, which composes a representation for global state space from all the state-graphs with manageable space and CPU-time consumption. There are three types of manipulators in our method: $\times$ for binary merge, **check**() for model-checking, and *reducers* for reducing the sizes of state-graphs. In between, iterative binary mergings of state-graphs, combinations of reducers can be applied to control the complexity of newly constructed state-graphs. With the many manipulators in our method, users can easily test different combinations of manipulators. In Table 1b, we have another example verification session using our method. Users calculate the binary products of state-graphs and intermittently reduce them with different reducer combinations as the users see fit. One performance advantage of our approach is that the users can control the complexities of state-graphs with on-the-shelf manipulators before those state-graphs become out of control.

Just like a mechanic can buy various on-the-shelf components to build her/his dream car, users of SGM can also enjoy the technology of CAV without deep understanding of the component technologies and construct the verification procedure that better suits them with the manipulations supported in our method. At this moment, we have successfully developed several theoretically sound reducers. From our experiments, different application ordering of different reducers can achieve different complexity reductions. Thus, the performances of the many

• F. Wang is with the Institute of Information Science, Academia Sinica, Taipei, Taiwan 115, ROC. E-mail: farn@iis.sinica.edu.tw.
• P.-A. Hsiung is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 621, ROC.

Manuscript received 28 May 1999; revised 1 Mar. 2001; accepted 12 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109957.

TABLE 1
Verficiation Sessions in Comparison

```
verify(G, 8, φ)
state_graph *G;
int 8; /* number of processes */
CTL φ;
{
  G := G[1] × G[2] × ... × G[8];
  check G against φ;
}
```

(a)

```
verify(G, 8, φ) /* Assume we have reducers R1, ..., R6. */
state_graph *G;
int 8; /* number of processes */
CTL φ;
{
  G := G[1] × G[2];
  G := R1(R4(G));
  G := G × G[3];
  G := R2(R5(R6(G)));
  G := G × G[4];
  ......
  G := G × G[8]
  G := R1(R2(G));
  check G against φ;
}
```

(b)

reduction techniques really depend on each other. This is due to the fact that a reduction technique may need the information derived by another to make further reduction. (See Example 4.) SGM provides a high-level and clean interface for users to experiment with different combinations of manipulators to reduce resource consumptions (memory and CPU time) to fulfill given verification tasks.

For each different verification task, there can be a different reducer combination for it which may cost the least memory space and CPU time. One research issue in our method is how to choose a good reducer combination for a given verification task when the number of reducers is large. In Section 10, we also present an algorithm based on group theory [20], [21] to pick an efficient reducer combination for a given verification task.

Another contribution of the work is in our design of reduction techniques in the context of iterative composition and global shared variable manipulations. For example, we may have m state-graphs $G_1, \ldots, G_m$ constituting a concurrent system and $G_{1:k}$ is generated by composing $G_1, G_2, \ldots, G_k$ with $k < m$. Each of the state-graphs may write specific and unique values to some shared variables. We aim to design reduction techniques on intermediate state-space representations like $G_{1:k}$ by analyzing the distinct values compared with and written to by various parties in the concurrent system. Previous model-checkers like Kronos gain part of their performance by detecting "dead transitions" (i.e., transitions which can never be triggered) after generating a Cartesian product of automata. But, Kronos does not allow natural manipulations of global variables and thus is not able to detect dead transitions whose triggering conditions depend on global variable values. To eliminate dead transitions in the context of iterative composition and global shared variables, we have to research into deeper theory of concurrency and work out new techniques. Instead of using the product calculation technique, we used our Lemma 1 to eliminate "dead transitions" with locality of global read-write operations of concurrent processes. Moreover, we believe that our approach better utilizes the properties among concurrent global operations and has the potential of reducing space-complexity before it becomes uncontrollable in the generation of Cartesian product.

Section 2 defines our problems. Section 3 discusses some related work. Section 4 presents the general framework of our method. Section 5 describes the function and data-structures of our state-graphs. Sections 6 and 7 describe the manipulators we have implemented so far. Section 8 demonstrates SGM in its graphical user interface. Section 9 reports experiments on several benchmarks. Section 10 shows an algorithm which uses group theory to find a local optimal combination of manipulators to counter with state-explosion problem. Section 11 contains the conclusion, our perspective of using this research as a public framework to enhance the application of verification technology and cooperation throughout the world.

We shall adopt the following notations. Given a set or sequence F, $|F|$ is the number of elements in F. For each element e in F, we also write $e \in F$. $\mathcal{N}$ is the set of nonnegative integers, $\mathcal{Z}$ is the set of integers, and $\mathcal{R}^+$ is the set of nonnegative reals.

2 Formal Problem Definition

We formally define our dense-time concurrent systems and their CTL model-checking problem, respectively, in two subsections.

2.1 Timed Mode Transition Systems

A real-time concurrent system is composed of many *processes*. Each process runs autonomously and interacts with others through read-write operations to *global variables* and *timers*. In addition, each process has its own *local variables* and *timers* which no other processes can access. For a system with m processes, we use integer $1, \ldots, m$ to identify the m processes.

Given a timer set H and a variable set F, a *state predicate* η of H and F is a formula constructed according to the following syntax:

$$\eta ::= y = c \mid y = p \mid x + c \sim x' + d \mid x \sim c \mid \neg\eta \mid \eta \vee \eta'.$$

y is a variable in F. c, d are natural numbers. p is the process identifier symbol which represents the local process. x, x' are timers in H. $\sim$ is an inequality operator in $\{\leq, <, =, >, \geq\}$. Common shorthands like $true$, $false$, conjunction ($\wedge$), and implication ($\rightarrow$) can be defined. Notationally, we let B_F^H be the set of all state predicates of H and F.

Definition 1 (PTMTS). *The process timed mode-transition system (PTMTS) is defined to describe behaviors in an atomic process in a* real-time concurrent system. *It is essentially a*

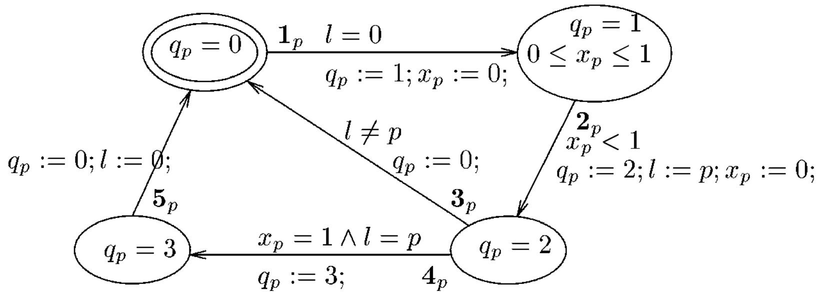


Fig. 1. Fischer's timed mutual exclusion protocol.

timed automata with an identifier and global variables. Notationally, we use symbolic subscript p to denote those transitions, local variables, and timers of process p. When the definitions are instantiated to a particular process with identifier $i \in \{1, \dots, m\}$, all occurrencess of p are to be substituted for i. The identifier of a process can be used in the flexible manipulation of global and local variables.

A PTMTS for process p is a tuple $A_p = \langle X, X_p, Y, Y_p, I_p, T_p \rangle$ with the following restrictions.

- *X is the global timer set, X_p is the local timer set, Y is the global variable set, and Y_p is the local variable set.*
- *$q_p \in Y_p$ is a special local variable which records the current* mode *of process p.*
- *I_p is a state predicate in $B^{X \cup X_p}_{Y \cup Y_p}$ denoting the* invariance condition *of process p. (Thus, the global invariance condition is $\bigwedge_{1 \le p \le m} I_p$.)*
- *T_p is the set of transition rules with the following form:*

  $$(q_p = c \wedge \eta) \rightarrow [q_p := d; \kappa].$$

  Here, c and d are natural numbers. η is a state predicate in $B^{X \cup X_p}_{Y \cup (Y_p - \{q_p\})}$ denoting the transition triggering condition. *κ is a finite sequence of* assignment statements *which is executed on the happening of the transition. Each assignment statement in κ has the following syntax:*

  $$y := c; \mid y := p; \mid x := 0;$$

  Again, $y \in Y \cup (Y_p - \{q_p\})$, $x \in X \cup X_p$, p is the process identifier symbol, and c is a natural number.

Initially, all timers and variables contain zeros. *The processes act by performing transitions in an interleaving fashion, i.e., at any moment, at most one transition can happen. Right before a transition $e = (q_p = i \wedge \eta) \rightarrow [q_p := j; \kappa] \in T_p$ happens, A_p is in mode i and η is satisfied. On the happening of e, which is instantaneous, variables are assigned new values and timers are reset to zeros according to κ and then A_p enters mode j. In between the happenings of transitions, all variable contents stay unchanged and all timer readings increment at a uniform rate.*

Example 1. For each process of Fischer's timed mutual exclusion protocol, we have a PTMTS $\langle X, X_p, Y, Y_p, I_p, T_p \rangle$ with $X = \emptyset$, $X_p = \{x_p\}$, $Y = \{l\}$, $Y_p = \{q_p\}$, $I_p = q_p = 0 \vee (q_p = 1 \wedge 0 \le x_p \wedge x_p \le 1) \vee q_p = 2 \vee q_p = 3$, and

$$T_p = \left\{ \begin{array}{l} (q_p = 0 \wedge l = 0) \rightarrow [q_p := 1; x_p := 0;], \\ (q_p = 1 \wedge x_p < 1) \rightarrow [q_p := 2; l := p; x_p := 0;], \\ (q_p = 2 \wedge l \neq p) \rightarrow [q_p := 0;], \\ (q_p = 2 \wedge x_p = 1 \wedge l = p) \rightarrow [q_p := 3;], \\ (q_p = 3) \rightarrow [q_p := 0; l := 0;] \end{array} \right\}.$$

In Fig. 1, we draw the PTMTS as a timed automaton which is more visually readable. The circles are modes and the starting mode is doubly circled. Inside the circles, we put down the mode names and invariance conditions enforced by I_p in the modes. On each transition, we put down the triggering condition (η), if any, above the assignment statements (κ), if any. For example, in mode $q_p = 1$, $0 \le x_p \le 1$ must be true for process p. In mode $q_p = 1$, when $x_p < 1$, process p may assign p to variable l, reset x_p to zero, and enter mode $q_p = 2$. We also label each transition with a boldface number near its source for later use.

Given a PTMTS $A_p = \langle X, X_p, Y, Y_p, I_p, T_p \rangle$ and a variable y in $Y \cup Y_p$, we let $D_{A_p:y}$ be the union of $\{0\}$ and the set of values assigned to y in T_p. That is, $D_{A_p:y}$ is the domain of y.

Definition 2 (States). *A real-time concurrent system, in our definition, is represented as a set of PTMTSs. Suppose we are given a real-time concurrent system S with m PTMTSs $A_1, \dots, A_m$ such that, for all $1 \le p \le m$, $A_p = \langle X, X_p, Y, Y_p, I_p, T_p \rangle$. A* state *of S is a mapping ν from $X \cup \bigcup_{1 \le p \le m} X_p \cup Y \cup \bigcup_{1 \le p \le m} Y_p$ such that for each $x \in X \cup \bigcup_{1 \le p \le m} X_p$, $\nu(x) \in \mathcal{R}^+$; for each $y \in Y$, $\nu(y) \in \bigcup_{1 \le p \le m} D_{A_p:y}$; and, for $1 \le p \le m$ and $y \in Y_p$, $\nu(y) \in D_{A_p:y}$. A state ν is an* initial state *iff, for all $z \in X \cup \bigcup_{1 \le p \le m} X_p \cup Y \cup \bigcup_{1 \le p \le m} Y_p$, $\nu(z) = 0$.*

Given a state and a state predicate η, we can define $\nu \models \eta$ (*ν satisfies η*) in a traditional inductive way.

- $\nu \models y = c$ iff $\nu(y) = c$,
- $\nu \models y = p$ iff $\nu(y) = p$,
- $\nu \models x + c \sim x' + d$ iff $\nu(x) + c \sim \nu(x') + d$,
- $\nu \models x \sim c$ iff $\nu(x) \sim c$,
- $\nu \models \neg\eta$ iff it is not the case that $\nu \models \eta$,
- $\nu \models \eta \vee \eta'$ iff $\nu \models \eta$ or $\nu \models \eta'$

Given a state ν and $\delta \in \mathcal{R}^+$, we let $\nu + \delta$ be a mapping identical to ν except that, for each $x \in X \cup \bigcup_{1 \le p \le m} X_p$, $(\nu + \delta)(x) = \nu(x) + \delta$. Given a sequence of assignment statements κ, we let $\nu\kappa$ be a new mapping identical to ν except that variables are assigned new values and timers are reset to zero according to κ.

Definition 3 (runs). *Suppose we are given a real-time concurrent system $A_1, \ldots, A_m$ such that $A_p = \langle X, X_p, Y, Y_p, I_p, T_p \rangle$ for all $1 \leq p \leq m$. A ν-run is an infinite sequence of state-time pair $(\nu_0, t_0)(\nu_1, t_1) \ldots (\nu_k, t_k) \ldots \ldots$ such that $\nu = \nu_0$, $t_0 t_1 \ldots t_k \ldots \ldots$ is a monotonically increasing real-number (time) divergent sequence and, for all $k \geq 0$,*

- *for all $t \in [0, t_{k+1} - t_k]$, $\nu_k + t \models \bigwedge_{1 \leq p \leq m} I_p$; and*
- *either*
  - $\nu_k + (t_{k+1} - t_k) = \nu_{k+1}$*; or*
  - *there are $p \in \{1, \ldots, m\}$ and $(q_p = \nu_k(q_p) \wedge \eta) \rightarrow [q_p := \nu_{k+1}(q_p); \kappa] \in T_p$ such that $\nu_k + (t_{k+1} - t_k) \models \eta$ and $(\nu_k + (t_{k+1} - t_k))\kappa = \nu_{k+1}$.*

If a ν-run describes the behavior of the system from the beginning of computation, then we require ν to be an initial state.

2.2 CTL and Model-Checking

Our verification framework is model-checking. That is, the system description is given in PTMTSs and the specification is given in CTL formulas [3], [19] and a verification problem instance asks if a given concurrent system with PTMTS processes satisfies a given CTL formula. A CTL formula has the following syntax:

$$\phi ::= \eta \mid \exists \square \phi' \mid \exists \phi' \mathcal{U} \phi'' \mid \neg \phi' \mid \phi' \vee \phi''$$

Here, η is a state predicate in

$$B^{X \cup \bigcup_{1 \leq p \leq m} X_p}_{Y \cup \bigcup_{1 \leq p \leq m} Y_p}.$$

$\exists \square \phi'$ means there exists a run, from the current state, along which ϕ' is always true. $\exists \phi' \mathcal{U} \phi''$ means there exists a run, from the current state, along which ϕ' is true until ϕ'' becomes true. Traditional shorthands, like $\exists \diamond$, $\forall \square$, $\forall \diamond$, $\forall \mathcal{U}$, $\wedge$ (conjuntion), big disjunction ($\bigvee_{1 \leq p \leq m} \cdots$), big conjunction ($\bigwedge_{1 \leq p \leq m} \cdots$), and $\rightarrow$, can all be defined.

The satisfaction of a CTL formula ϕ by a state ν in a real-time concurrent system S, written $S, \nu \models \phi$, can be defined in a standard way.

- $S, \nu \models \eta$ iff ν satisfies η as a state predicate.
- $S, \nu \models \exists \square \phi$ iff there is a ν-run
  $$(\nu_0, t_0)(\nu_1, t_1) \ldots (\nu_k, t_k) \ldots \ldots$$
  such that, for all $k \geq 0$ and $t \in [0, t_{k+1} - t_k]$, $S, \nu_k + t \models \phi$.
- $S, \nu \models \exists \phi \mathcal{U} \phi'$ iff there is a ν-run
  $$(\nu_0, t_0)(\nu_1, t_1) \ldots (\nu_k, t_k) \ldots \ldots,$$
  a $k \geq 0$, and a $t \in [0, t_{k+1} - t_k]$ such that
  - $S, \nu_k + t \models \phi'$; and
  - for all $0 \leq h \leq k$ and $t' \in [0, t_{h+1} - t_h]$, if $t_h + t' < t_k + t$, then $S, \nu_h + t' \models \phi$.
- $S, \nu \models \neg \phi$ iff it is not the case that $S, \nu \models \phi$.
- $S, \nu \models \phi \vee \phi'$ iff $S, \nu \models \phi$ or $S, \nu \models \phi'$

Also, we write $S \models \phi$ (S satisfies ϕ) iff, for the initial state ν of S, $S, \nu \models \phi$.

Example 2. The mutual exclusion specification of Fischer's protocol in Fig. 1 is $\forall \square \neg \bigvee_{1 \leq p < p' \leq m} (q_p = 3 \wedge q_{p'} = 3)$.

3 Related Work

This section gives a brief account of some mature tools that are widely known and of the verification techniques they have used. It is worth noticing that, although such tools are numerous, very few tools have provided high-level verification perceptions yet. The following are some tools and techniques used for formal verification.

SMV [34] is the first model-checking tool for CTL (*Computation Tree Logic*) specifications with BDD (Binary Decision Diagram) technology. NuSMV [9], a newer version of SMV, is one tool that will be adopting a high-level user interface. But, they still do not allow users to easily construct verification strategies best fitting their verification tasks.

SPIN [23] is an automated protocol validation system using PROMELA with refining dependencies for efficient partial-order verification [10], [18].

Murϕ Verification System [13] is a language-based verification tool for finite-state concurrent systems with symmetry-based reduction [27].

Aldebaran [16] is a component of the CADP protocol engineering toolbox developed at INRIA/Verimag with strong/weak (bi)simulation, safety preorder/equivalence, on-the-fly verification techniques [17], time-abstracting bisimulation implemented [40]. Yet another component in the CADP toolset is **XTL Model Checker** [33] with states, transitions, and labels handling labeled transition systems.

CAML Prototype [29] has implemented a quotienting technique of moving a parallel system description into a specification formula, trivial equation elimination, and equivalence reduction. **KRONOS** [11] is a well-known verification tool with minimization algorithms [11], inactive clock reduction [12], and clock equality [12] reduction techniques implemented. Kronos does not allow natural manipulations of global variables. We believe that, in the context of iterative composition, Kronos is not able to detect dead transitions whose triggering conditions depend on global variable values. Moreover, Kronos detects dead transitions by generating a Cartesian product of automata, which can be very space-inefficient.

UPPAAL [31], [30], [7] is a widely used verification tool for real-time systems with a graphical interface, simulation, quotient construction, minimizations, trivial equation elimination, and equivalence reduction implemented [31]. Memory space explosion was tactically handled by compact data structures [30] and optimized constraint manipulations [7].

In Section 7.4, we discuss how to reduce state-graph sizes through bypassing internal transitions. Similar concepts may be dated back to the "internal action" of process algebra [22], [35]. A recent similar concept is the "invisible transition" by Miller and Katz [36]. However, there is both a similarity and a distinction between the concepts of Miller and Katz's "invisible" and our "internal." In the similarity part, both concepts try to hide information which is not interesting to the outside world. In the distinction part, Miller and Katz focus on the observation (i.e., specification)

TABLE 2
An Example Verification Procedure

```
verify($G, m, \phi$) /* Assume we have reducers $\mathbf{R}_1, \ldots, \mathbf{R}_6$. */
state_graph $*G$;
int $m$;
CTL $\phi$;
{
  state_graph $H, H_1, H_2, H_3$;                                        (1)
  int $i$;                                                               (2)

  $H := G[1]$;                                                           (3)
  for $i := 2$ to $m$, do {                                              (4)
    $H := H \times G[i]$;                                                (5)
    $H_1 := \mathbf{R}_1(\mathbf{R}_2(H))$;                              (6)
    $H_2 := \mathbf{R}_3(\mathbf{R}_4(\mathbf{R}_5(\mathbf{R}_6(H))))$;  (7)
    if Size($H_1$) $<$ Size($H_2$) then $H := H_1$;                      (8)
    else if Size($H_1$) $>$ Size($H_2$) then $H := H_2$;                 (9)
    else if Time($H_1$) $<$ Time($H_2$) then $H := H_1$;                 (10)
    else $H := H_2$;                                                     (11)
  }
  return check($H, \phi$);                                               (12)
}
```

of the users, while we focus on the interaction among peer processes. This distinction drove us to design new techniques with Lemmas 1 and 2. More precisely, the property of "invisibility of transitions" proposed by Miller and Katz depends on the specification, whereas our property of "internal" is independent of any specification. A transition may become internal only when verification is performed compositionally.

From the above, we notice that, although different techniques have been implemented in the various well-known tools, yet if a user needs to apply two or more different techniques to a verification task and if those techniques were implemented in different tools, then the user will have to expend great effort trying to either translate the output results of one tool to the input format of another tool or make some strong assumptions of technique application that may be invalid. Even with one verification tool, users still face a delimma. On one side, without a user-friendly interface for users to construct high-level verification strategy, it may become difficult for the tools to figure out the right combination of existing reduction techniques for a given verification task. On the other side, for new users, to acquaint themselves with the reasoning structures of various tools in order to maximize verification efficiency can just be too costly and too time-consuming for their jobs. Such painstaking efforts could be avoided if various compatible techniques could be collected, implemented, and integrated into a single environment in which users can flexibly fine-tune their verification strategy. SGM is proposed with this motivation in mind.

4 General Framework of Verification

Our method can be embodied in a simple language of verification procedure. The language looks like Pascal or C, but supports high-level objects with types of integer, state-graphs, and CTL formula. In the following, we give an example to illustrate how to define a verification procedure in the language.

Merging of state-graphs is performed by the binary $\times$. Then, we have a set of theoretically proven reducers which can be designed by anyone. The control of verification procedure can be achieved with nested **for**-loops indexed on integers and **if**-statements with conditions on state-graph sizes, graph construction times, and index variables. In Table 2, we have an example verification procedure written in the language. Verification procedures always take three arguments. The first, here G, is an array of state-graphs; the second, here m, the number of state-graphs (processes) in the concurrent system; and the third, here ϕ, the CTL formula to check with. The array is declared in C-language style. Lines (1) and (2) declare variables of state-graph type and integer type, respectively. Line (4) iterates the **for**-loop to merge and reduce the state-graphs. Lines (6) and (7) calculate two alternative combinations of reducers. The **if**-structure starting at line (8) chooses the reduction result with, first, the least size and, then, the least CPU-time for each iteration. For any state-graph H, we let $\mathbf{size}(H) = \mathbf{NodeCount}(H) + \mathbf{ArcCount}(H)$. **NodeCount()** is a system-defined function which returns the number of nodes in the argument state-graph. Similarly, another function, **ArcCount()**, returns the number of arcs in the argument state-graph. **Time()** is the CPU time used to construct the state-graph in its argument. After the loop from lines (4) to (11) is over, H is a reduced representation for the global state space. Then, at line (12), we check H against ϕ.

Since our method allows reasoning on state-graphs as whole objects instead of searching for paths in them, users can test different combinations of reducers to achieve their verification tasks from a resource management point of

TABLE 3
Manipulator $\times$

```
$\times(G_1, G_2)$ /* $G_1 = \langle V_1, v_{1:0}, XState_1, E_1, (XPair_1, XProc_1, Xtion_1, XPerm_1)\rangle$
        and $G_2 = \langle V_2, v_{2:0}, XState_2, E_2, (XPair_2, XProc_2, Xtion_2, XPerm_2)\rangle$ */

  Let $G := \langle V, v_0, XState, E, (XPair, XProc, Xtion, XPerm)\rangle$
  Let $XState(v_0) := XState_1(v_{1:0}) \wedge XState_2(v_{2:0})$;
  If $XState(v_0)$ is false, return NULL.
  Let $U := \{(v_0, v_{1:0}, v_{2:0})\}$; $V := \{v_0\}$; $E := \emptyset$;
  While $U$ is not empty, do {
    Choose $s = (v, v_1, v_2) \in U$;
    For each $i \in \{1, 2\}$ and $e \in E_i$ with $XPair(e) = (v_i, \bar{v}_i)$ for some $\bar{v}_i$, do {
      Let $v'$ be the destination node from $v$ by transition $e$.
      Calculate $XState(v')$ as the postcondition of transition $e$ from $v$
        according to the algorithm in [19].
      If $XState(v')$ is FALSE, then break;
      If there is a $\dot{v} \in V$ such that $XState(v') = XState(\dot{v})$, let $v' := \dot{v}$; else {
        Let $V := V \cup \{v'\}$;
        if $i = 1$, then construct a new $(v', \bar{v}_1, v_2)$ in $U$;
              else construct a new $(v', v_1, \bar{v}_2)$ in $U$;
      }
      Let $E := E \cup \{e'\}$; $XPair(e') := (v, v')$; $XProc(e') := XProc_i(e)$;
        $Xtion(e') := Xtion_i(e)$; $XPerm(e') := XPerm_i(e)$;
    } Let $U := U - \{s\}$;
  }
  return $A$;
}
```

view without deep knowledge of the technology and theory inside the manipulators.

5 Implementation of State Graphs

A state-graph G for process set $H = \{i_1, i_2, \ldots, i_h\}$ is a multigraph (allowing more than one arc from one node to the other) and is conceptually implemented as a tuple,

$$G = \langle V, v_0, XState, E, (XPair, XProc, Xtion, XPerm)\rangle.$$

Each element in V is a *region* which represents a set of states. v_0 is the initial region. For each $v \in V$, $XState(v)$ is a condition true for all states represented by v. Specifically, in $XState(v)$, we record the following information:

- The values of all local and global discrete variables (including q_p) in H.
- The Difference-Bound Matrix [2], [14] which records the differences among all local and global timers in H up to the biggest timing constant used in the input PTMTS.
- Some propositions prepared by manipulators in SGM for the use of other manipulators. For example, we may use Lemma 1 in Section 7.1 to infer that, at some states, $l = p$ must be false.

Thus, when we say two states are in the same regions, we mean they are identical with respect to their information recordings.

E is the set of arcs among nodes in V. For each $e \in E$,

- $XPair(e) = (v, v')$ describes the source and destination of arc e.
- $XProc(e) \in \{1, \ldots, m\}$ defines the index of the process which makes the transition corresponding to e.
- $Xtion(e)$ is an index representing the transition which corresponds to e.
- $XPerm(e)$ is a permutation of process identifiers 1 through m and is needed because we implement reduction with symmetry [15].

State-graphs are multigraphs because there can be more than one transition leading from one node to the other. Reduction by symmetry also adds multiplicity to arcs between pairs of nodes. From $XProc()$ and $Xtion()$, we can reach informations about triggering conditions and assignment statements.

6 On-the-Fly Merging of State-Graphs

In this section, we shall first briefly describe our merge manipulator, $(\times)$, which merges two state-graphs into a new one which represents a finer description of the global state-space. Suppose we are given two state-graphs, with $i \in \{1, 2\}$,

$$G_i = \langle V_i, v_{i:0}, XState_i, E_i, (XPair_i, XProc_i, Xtion_i, XPerm_i)\rangle$$

for process set $H_i = \{p_{i:1}, p_{i:2}, \ldots, p_{i:m_i}\}$ such that $H_1 \cap H_2 = \emptyset$. $\times(G_1, G_2)$ (or $G_1 \times G_2$ in infix notations), computed with the procedure in Table 3, is a new state-graph constructed with on-the-fly technique from G_1 and G_2 for process set $H_1 \cup H_2$. The reader should be mindful that the output of a merge operation is not a plain Cartesian product. It also contains information for reduction information needed for further composition and reduction (like symmetry reduction). Since the merge is in an on-the-fly style, some inconsistent state representations can be discarded before being generated.

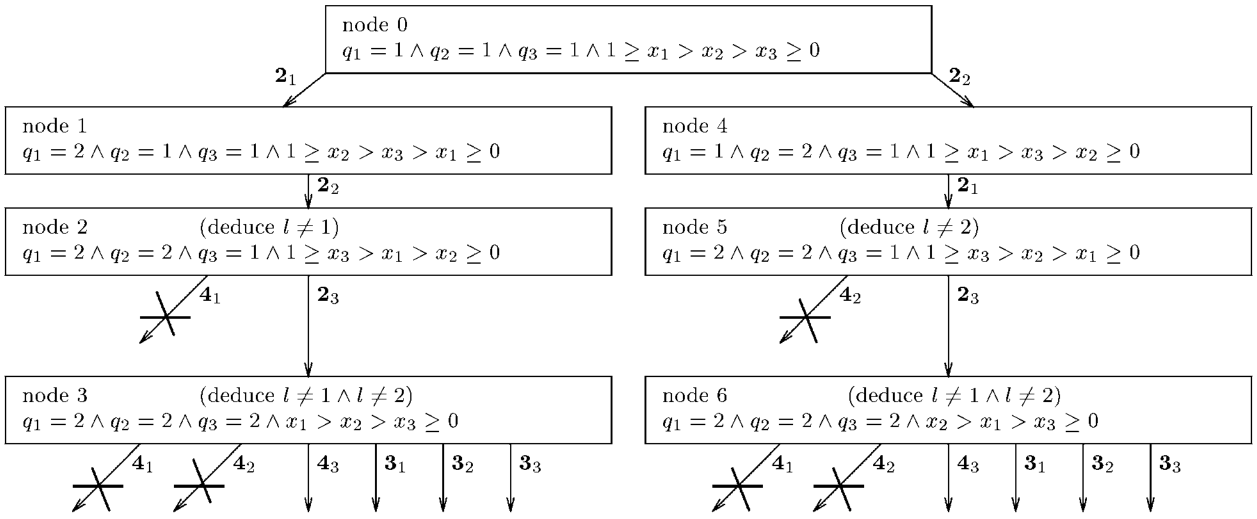

node 0
$q_1 = 1 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_2 > x_3 \geq 0$
$\mathbf{2}_1$
$\mathbf{2}_2$
node 1
$q_1 = 2 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_2 > x_3 > x_1 \geq 0$
node 4
$q_1 = 1 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_3 > x_2 \geq 0$
node 2 (deduce $l \neq 1$)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_1 > x_2 \geq 0$
node 5 (deduce $l \neq 2$)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_2 > x_1 \geq 0$
$\mathbf{4}_1$ $\mathbf{2}_3$ $\mathbf{4}_2$ $\mathbf{2}_3$
node 3 (deduce $l \neq 1 \wedge l \neq 2$)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 2 \wedge x_1 > x_2 > x_3 \geq 0$
node 6 (deduce $l \neq 1 \wedge l \neq 2$)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 2 \wedge x_2 > x_1 > x_3 \geq 0$
$\mathbf{4}_1$ $\mathbf{4}_2$ $\mathbf{4}_3$ $\mathbf{3}_1$ $\mathbf{3}_2$ $\mathbf{3}_3$

Fig. 2. Illustration of Lemma 1.

7 Implemented Reducers

In this section, we discuss the techniques we used to reduce state-graphs. Sections 7.1, 7.2, and 7.4 discuss our newly developed techniques in details with lemmas and examples. Note that our reduction techniques work in the context of iterative composition and shared variable manipulations. Suppose we have merged k state-graphs, out of m given state-graphs, into $G_{1:k}$. We shall establish lemmas which will help us avoid generating unreachable state preresentations in those intermediate state-graphs like $G_{1:k}$ in the first place. Lemma 1 in Section 7.1 helps us to analyze the values written to shared variables by different parties in a concurrent system and to predict the shared variable value ranges in the future. This lemma is very handy in our derivation of value-stability properties of state subspaces of those intermediate state-graphs and can be used to eliminate state subspaces and (dead) transitions, whose triggering conditions depend on global variables and can never be satisfied.

Lemma 2 in Section 7.2 derives a path-based property for timer-elimination and can be used to deduce the behavior-equivalence among states. The lemma is established in the context of iterative composition of state-graphs. Lemma 2 predicts, in a given state, whether the value of a clock will be used again before being reset by considering the behavior structure in $G_{1:k}$ and the read-write values used in the other $m - k$ state-graphs for the clock.

Section 7.3 discusses how we adapt the technique of verification by symmetry [15] to dense-time systems.

Section 7.4 discusses how we can exploit the fact that some transitions becomes internal only after an intermediate state-graph, like $G_{1:k}$, is generated.

7.1 Variable Value Stability under Concurrent Read/Write

Suppose we have an intermediate state-graph composed from state-graphs for processes with identifiers in set H. For a given global variable y, we let $D_{H:y}$ be the set of values written to y by processes with identifiers in H, but NOT by processes without identifiers in H. We present the following lemma, which governs the stability of global variable values, derivable from intermediate state-graphs, in concurrent read/write systems.

Lemma 1. *Suppose we are given a variable y and a finite run segment $(\nu_h, t_h)(\nu_{h+1}, t_{h+1})\ldots(\nu_k, t_k)$ such that, for all $h \leq i < k$, ν_i goes to ν_{i+1} without making an assignment to y on a transition from a process with an identifier in H.*

- *If we enter state ν_h with an assignment $y := a;$, then, for all $h \leq i < k, t_i \leq t \leq t_{i+1}$,*
$$\nu_i + t \models \bigwedge_{b \in (D_{H:y} - \{a\})} y \neq b.$$
- *If we enter state ν_h without an assignment to y but with a triggering condition $y = a$, then, for all $h \leq i < k, t_i \leq t \leq t_{i+1}$, $\nu_i + t \models \bigwedge_{b \in (D_{H:y} - \{a\})} y \neq b$.*
- *If we enter state ν_h without an assignment to y but with a triggering condition $y \neq a$ with $a \in D_{H:y}$, then, for all $h \leq i < k, t_i \leq t \leq t_{i+1}$, $\nu_i + t \models y \neq a$.*

Proof. If the values in $D_{H:y}$ are not going to be written to y by other processes with identifiers not in H, since we have the full knowledge that processes with identifiers in H will neither write values in $D_{H:y}$ to y along the segment, thus the lemma must hold. □

Our SGM takes advantage of Lemma 1 in the following way: In a given state-graph, we have a set of nodes each representing a set (region) of states. Lemma 1 is first applied to deduce the truth values of propositions about valuations of discrete variables in the nodes. Certain nodes can be eliminated because of invalidated invariance conditions. Certain transitions can also be eliminated due to invalidated triggering conditions. After such elimination, the state-graph size is, hopefully, reduced.

Example 3. In the state-graph in Fig. 2, which is part of the state-graph generated for Fischer's timed mutual exclusion algorithm in Fig. 1, the square boxes represent nodes (regions) while the arcs represent transitions with transition indices and process identifiers labeled by their sides. The crossed-out arcs represent transitions detected as untriggerible by Lemma 1. For example, in node 2, we can deduce $l \neq 1$ and, thus, conclude transition 4_1 will never be triggered from node 2. Thus, Lemma 1 can be used to early delete arcs and nodes which eventually are unreachable in the final state-graphs.

7.2 Irrelevance of Shielded Timers

A *timing atom* is an atom of the syntax: $x - c \sim x' - d \mid x \sim c$, where x, x' are clocks and c, d are natural numbers. A set Γ of timing atoms is *shielded* in a state ν with respect to a state predicate η iff the truth values of the timing atoms in Γ together do not affect the truth value of η. To explain this concept, we define function $\Omega()$, which, given a timing atom set Γ, state ν, and a state-predicate η, evaluate all atoms in η except those in Γ. Formally speaking, we define $\Omega(\Gamma, \nu, \eta)$ inductively as follows:

- $\Omega(\Gamma, \nu, \chi) = \{true, false\}$ if $\chi \in \Gamma$.
- $\Omega(\Gamma, \nu, \chi) = \{\nu \models \chi\}$ if χ is atomic and $\chi \notin \Gamma$.
- $\Omega(\Gamma, \nu, \neg\eta) = \{\neg b \mid b \in \Omega(\Gamma, \nu, \eta)\}$.
-
$$\Omega(\Gamma, \nu, \eta \vee \eta') = \{b \vee b' \mid b \in \Omega(\Gamma, \nu, \eta); \\ b' \in \Omega(\Gamma, \nu, \eta')\}.$$

Here, we assume that

$$\neg true \equiv false,$$
$$\neg false \equiv true,$$
$$true \vee true \equiv true,$$
$$true \vee false \equiv true,$$
$$false \vee false \equiv false,$$

and $false \vee true \equiv true$. Then, Γ is *shielded* in ν w.r.t. η iff $|\Omega(\Gamma, \nu, \eta)| = 1$, which means the values of η are independent of atoms in Γ in state ν. For instance, in Example 1, timing atom set $\{0 \leq x_p, x_p \leq 1\}$ is shielded in any state which satisfies $q_p = 0$ w.r.t. $I_p = q_p = 0 \vee (q_p = 1 \wedge 0 \leq x_p \wedge x_p \leq 1) \vee q_p = 2 \vee q_p = 3$.

Definition 4 (Timer shieldedness). *Suppose we are given a model-checking problem instance with system S and CTL formula ϕ. A timing atom in one of the following forms: $x \sim c$, $x + c \sim x' + d$, and $x' + c \sim x + d$ is called an x's timing atom. Let U_x be the set of all timing atoms of x used in the model-checking problem instance. A timer x is* shielded *in a state ν for the problem instance iff*

1. *x is not used in the specification ϕ and*
2. *for any $k \in \mathcal{N}$ and ν-run*

$$(\nu_0, t_0)(\nu_1, t_1) \ldots (\nu_k, t_k) \ldots\ldots,$$

if either

a. *$k > 0$ and ν_{k-1} goes to ν_k on a transition $\eta \rightarrow [\kappa]$ such that $|\Omega(U_x, \nu_{k-1} + t_{k-1}, \eta)| > 1$ or*

b. *$k \geq 0$ and, for some $t \in [0, t_{k+1} - t_k]$,*

$$|\Omega(U_x, \nu_k + t, \bigwedge_{1 \leq p \leq m} I_p)| > 1,$$

then there is a $0 < h < k$ such that ν_{h-1} goes to ν_h on a transition $\eta' \rightarrow [\ldots; x := 0; \ldots]$.

Case 2a is a state property which says a transition can be either allowed to or forbidden from happening due to different valuations of timing atoms in U_x. Case 2b is a state property which says a region can be entered or not due to different valuations of timing atoms in U_x. Case 2 is a path property which intuitively says that if the current reading of timer x is not tested (with invariance conditions or triggered transitions) before it is reset, then the current reading of timer x is of no influence to system behavior in the future.

Lemma 2. *For every subformula ψ of the specification formula and every two states ν, ν' such that ν and ν' are identical except for the readings of x, which is shielded in both ν and ν', $S, \nu \models \psi$ iff $S, \nu' \models \psi$.*

Proof. Suppose we have a ν-run

$$(\nu_0, t_0)(\nu_1, t_1) \ldots (\nu_k, t_k) \ldots\ldots$$

We want to establish that we can construct another ν'*-run*

$$(\nu'_0, t_0)(\nu'_1, t_1) \ldots (\nu'_k, t_k) \ldots\ldots$$

such that, for all $k > 0$, ν_{k-1} goes to ν_k and ν'_{k-1} goes to ν'_k on the same transition $e_k = \eta \rightarrow [\kappa]$ of the same process. Note that ν, ν' only differ at readings of x.

Assume the construction is impossible because there is at least $k > 0$ such that e_k can be triggered in the ν-run, but cannot be triggered in the ν'-run due to either unsatisfied triggering condition (Case 2a in Lemma 2) or unsatisfied invariance condition (Case 2b). Since the only difference between the two runs is caused by the different readings of x in ν and ν', this implies either $\Omega(U_x, \nu_{k-1} + t_k - t_{k-1}, \eta) > 1$ or $\Omega(U_x, \nu_k, \bigwedge_{1 \leq p \leq m} I_p) > 1$, which again implies, by our shieldedness condition, that, before e_k, there is another transition which has reset x to zero in both runs. This means $(\nu_{k-1} + t_k - t_{k-1})(x) = (\nu'_{k-1} + t_k - t_{k-1})(x)$, which is a contradiction to our assumption. Thus, we know the construction can be made and, by induction on the subformula structure, we can show the "iff" relation. □

Here is how we plan to use Lemma 2. In the set of nodes in a state-graph, after application of Lemma 2, we can eliminate some pieces of information (actually columns and rows in zone matrices and timing atoms) from the recordings of nodes. After such elimination, we may hopefully find out some nodes are indistinguishable now and we only have to keep only one representative node in each of such indistinguishable groups. Thus, the state-graph size can be reduced.

Example 4. In Fig. 3, part of the state-graph generated for Fischer's timed mutual exclusion algorithm in Fig. 1, we illustrate how Lemma 2 works. Fig. 3 can be thought as a reduction from Fig. 2. Let us examine how this reduction is possible. By analyzing the intermediate state-graphs G after merging the state-graphs for processes 1, 2, and 3, we can deduce that, in Fig. 2, x_1 is shielded in node 2, x_2

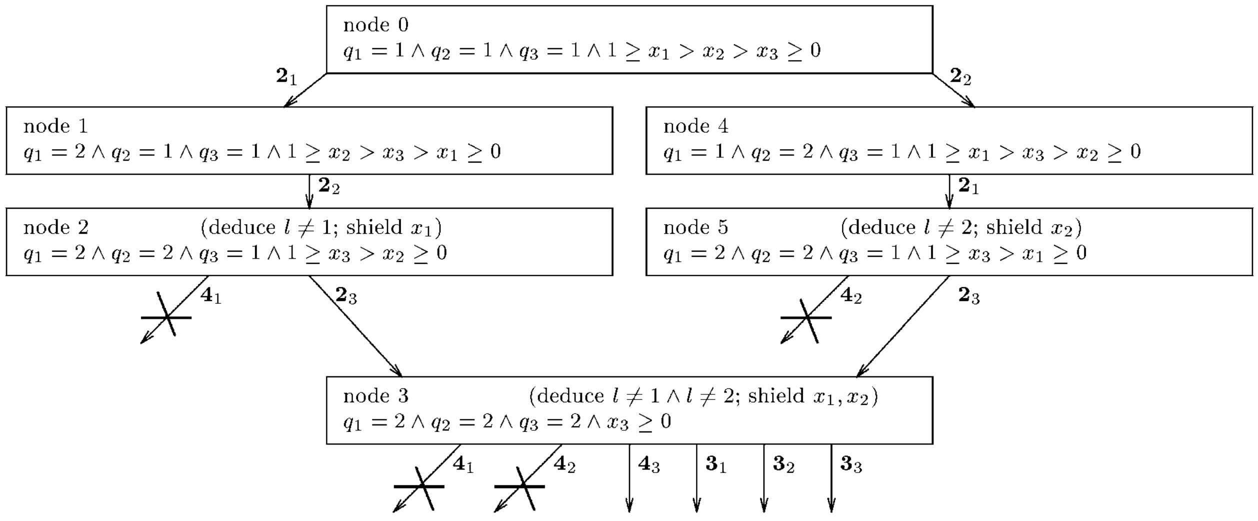

node 0
$q_1 = 1 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_2 > x_3 \geq 0$
node 1
$q_1 = 2 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_2 > x_3 > x_1 \geq 0$
node 4
$q_1 = 1 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_3 > x_2 \geq 0$
node 2 (deduce $l \neq 1$; shield x_1)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_2 \geq 0$
node 5 (deduce $l \neq 2$; shield x_2)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_1 \geq 0$
node 3 (deduce $l \neq 1 \wedge l \neq 2$; shield x_1, x_2)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 2 \wedge x_3 \geq 0$
$\mathbf{2}_1$ $\mathbf{2}_2$ $\mathbf{2}_2$ $\mathbf{2}_1$ $\mathbf{4}_1$ $\mathbf{2}_3$ $\mathbf{4}_2$ $\mathbf{2}_3$ $\mathbf{4}_1$ $\mathbf{4}_2$ $\mathbf{4}_3$ $\mathbf{3}_1$ $\mathbf{3}_2$ $\mathbf{3}_3$

Fig. 3. Illustration of Lemma 2.

is shielded in node 5, and x_1, x_2 are shielded in both nodes 3 and 6. To convince ourselves of these, let us look at node 2 and timer x_1. Condition 1 is obviously satisfied (check Example 2). Condition 2 can be established as follows: In node 2, process 1 is in mode $q_1 = 2$ from which x_1 is read only by process 1 with transition 4_1 before any reset. But, from Example 3, we already knew that, from mode $q_1 = 2$, transition 4_1 is not triggerible. Thus, we deduce Conditions 1 and 2 are both satisfied.

After shielding the clocks in different nodes, we find that, according to Lemma 2, nodes 3 and 6 are equivalent. Thus, we can just keep one of them. However, we note for readers that the shieldedness of the timers in this example is detected only after we have eliminated arcs using Lemma 1. This implies that application ordering among reducers can affect verification performance.

7.3 Symmetry through Index Permutation

We extend the technique of reduction by symmetry [15] to dense-time systems. The idea is to attach process identifer permutation to arcs in state-graphs. Only one node among those equivalents under process identifier permutation will be kept. This usually results in factorial reduction in state-graph sizes. By applying the reduction by symmetry technique, we can further reduce the structure in Fig. 3 to the one in Fig. 4, which is again part of the state-graph generated for Fischer's timed mutual exclusion algorithm in Fig. 1. Note that there is a new arc from node 0 labeled with 2_2 now. On the same arc, we also label the permutation of $(2, 1)$.

As we see, reduction by symmetry transforms a state-graph into a multigraph. In fact, a lot of the arcs can also be eliminated to save memory space. We adopt a two-fold approach to this aspect.

- For CTL specification which is symmetric to all process identifiers, we shall also reduce those arcs which are equivalent under permutation by process

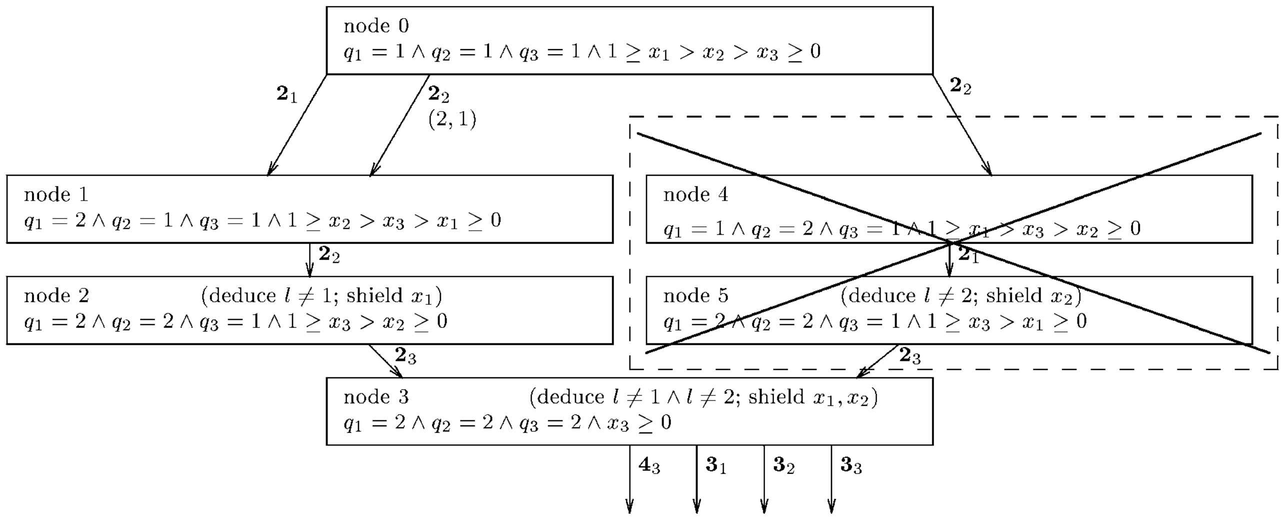

node 0
$q_1 = 1 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_2 > x_3 \geq 0$
$\mathbf{2}_1$
$\mathbf{2}_2$ $(2,1)$
$\mathbf{2}_2$
node 1
$q_1 = 2 \wedge q_2 = 1 \wedge q_3 = 1 \wedge 1 \geq x_2 > x_3 > x_1 \geq 0$
node 4
$q_1 = 1 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_1 > x_3 > x_2 \geq 0$
$\mathbf{2}_2$ $\mathbf{2}_1$
node 2 (deduce $l \neq 1$; shield x_1)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_2 \geq 0$
node 5 (deduce $l \neq 2$; shield x_2)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 1 \wedge 1 \geq x_3 > x_1 \geq 0$
$\mathbf{2}_3$ $\mathbf{2}_3$
node 3 (deduce $l \neq 1 \wedge l \neq 2$; shield x_1, x_2)
$q_1 = 2 \wedge q_2 = 2 \wedge q_3 = 2 \wedge x_3 \geq 0$
$\mathbf{4}_3$ $\mathbf{3}_1$ $\mathbf{3}_2$ $\mathbf{3}_3$

Fig. 4. Illustration of reduction by symmetry.

TABLE 4
Bypass Internal Transition

```
BIT($G, \phi$) /* $G = \langle V, v_0, XState, E, (XPair, XProc, Xtion, XPerm)\rangle$ for process set $H$ */ {
  $E' = \emptyset$;
  For every $v_1, v_2 \in V$ and $e \in E$ with $XPair(e) = (v_1, v_2)$ and $Xtion(e) = \eta \rightarrow [\kappa]$, if
    • $XState(v_1)/I$ implies $XState(v_2)/I$; and
    • $XState(v_1)$ implies $\eta$ is true; and
    • $e$ is internal to $H$,
   then {
    For all $e' \in E$ with $XPair(e') = (v_2, v_3)$ for some $v_3 \in V$, do {
      create $e''$ such that $XPair(e'') := (v_1, v_3)$, $Xtion(e'') := Xtion(e')$,
        $XProc(e'') := XProc(e')$, and $XPerm(e'') := XPerm(e')$.
      $E' := E' \cup \{e''\}$;
    }
    $E := E - \{e\}$;
  }
  $E := E \cup E'$;
  Delete all unreachable nodes in $G$;
}
```

identifiers. It can be shown that such reduction will not affect the answer of model-checking.

- For asymmetric CTL specification, we still do the arc reduction with symmetry technique. But, when actually model-checking a CTL specification against a state-graph whose arcs have been reduced with symmetry technique, SGM will dynamically reconstruct the arcs which were reduced. Once the evaluation from one node to another is done, the arcs dynamically constructed will then be discarded. Such dynamic reconstruction and discarding can increase the efficiency of memory space management.

7.4 Internal Transition Bypassing

Suppose we are given a model-checking problem with dense-time concurrent system S, of m processes, and CTL formula ϕ. Given a set H of processes in S, we let Y^H be the set of discrete variables either read or written to by the processes in H. A discrete variable y is said to be *internal* to H iff

- $(y \in Y^H) \wedge (y \notin Y^{\{1,\dots,m\}-H})$;
- y is not used in ϕ.

In other words, an internal variable of a process set H is neither accessed by process not in H nor used by the specification. For example, all local variables of a process are internal variables of any state-graph. A variable which is not internal is called a *visible variable*. For example, all global variables of a system are visible variables.

Reading and writing of variables occurs on transitions in their triggering conditions and assignment statements, respectively. A transition $\eta \rightarrow [\kappa]$ is said to be *internal* to process H iff its triggering condition η and assignment statements κ only access internal variables of H. Note that κ does not reset any timer variables, either local or global. Otherwise, the progress of time is not continuous and outside processes may thus speculate on the occurrence time of the transition.

Given a region v, we let $XState(v)/I$ be the new region recording obtained from v by discarding all conditions on variables from I. Specifically, $XState(v)/I$ is obtained from v by deleting those values recordings for variables in I and those literals with variables in I. Our BIT reduction procedure is shown in Table 4.

Example 5. We have a token ring in Fig. 5 of four processes with template state-graph A_p. t_1, t_2, t_3, t_4 are the tokens. For each $1 \leq p \leq m$, token t_p is internal to process set $\{(p\%4)+1, p\}$ where $\%$ is the modulo operator. Mode $q_p = 1$ is the critical section. To check for mutual exclusion, it is enough to model-check on $\forall\Box\neg(q_1 = 1 \wedge q_2 = 1)$, which we assume to be the specification. After merging the state-graphs for A_3 and A_4, we find the path in Fig. 6. Note that, in $A_3 \times A_4$, q_3, q_4, and t_4 are all internal. Thus, according to our algorithm in Table 4, the arc from node 0 to node 1 can be bypassed.

8 User Interface

We have implemented a text-based user-interface and a window-based user-interface for SGM. They all support high-level verification in which users can manipulate state-graphs as basic data-objects.

8.1 Verification Procedure Language

The input language of SGM is called *Verification Procedure Language* (VPL), which consists of three parts for system description, specification, and state-graph manipulation. System description corresponds to the timed automata model of real-time systems as defined later in this section. Specification corresponds to a TCTL specification. Manipulation corresponds to a list of actions on state-graphs. The following denotes a basic input file structure, where – starts a comment line.

```
– (1)  System Description Part
       automata Client : i = 1..11;
       clock ...;
```

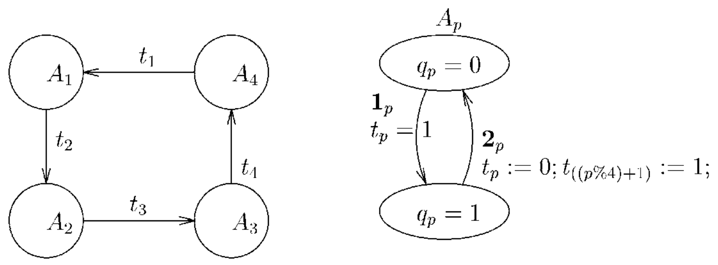
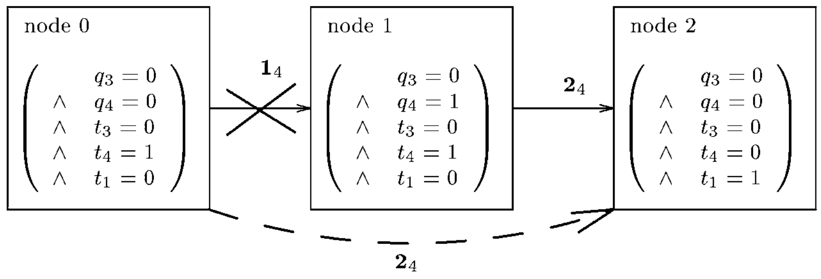


Fig. 5. A ring of four processes.



node 0
$q_3 = 0 \wedge q_4 = 0 \wedge t_3 = 0 \wedge t_4 = 1 \wedge t_1 = 0$
$\mathbf{1}_4$
node 1
$q_3 = 0 \wedge q_4 = 1 \wedge t_3 = 0 \wedge t_4 = 1 \wedge t_1 = 0$
$\mathbf{2}_4$
node 2
$q_3 = 0 \wedge q_4 = 0 \wedge t_3 = 0 \wedge t_4 = 0 \wedge t_1 = 1$
$\mathbf{2}_4$

Fig. 6. Part of a region graph for the ring network.

```
        register ...;
        automaton Client[i] { ........ }
- (2)   TCTL Specification Part
        verify ......;
- (3)   Manipulation Part
        manipulation
          graph G[1], G[2];
          merge_graph(Client[1], Client[2],
                      G[1]);
          shield_clock(G[1]); ..........
```

The above SGM input allows easy change of a system's degree of concurrency by simply changing the first line. SGM strives to maintain a parametric input description so that users do not have to write 11 automata descriptions or rewrite everything whenever a process is added or removed. In Table 5, we have the VPL description of Fischer's timed mutual exclusion protocol.

In VPL, we support the full syntax of TCTL formulas through various keywords, such as `all_paths`($\forall$), `exists_path`($\exists$), `henceforth`($\Box$), `eventually`($\Diamond$), etc. The specification begins with the keyword `verify`. Users can easily specify a TCTL formula using the user-friendly keywords. For example, `all_paths` means *for all paths in the state-graph starting from the initial mode*. For our running example of 3-automata Fischer's timed mutual exclusion protocol (FMEP) [28], [1], [32], the TCTL specification is as follows:

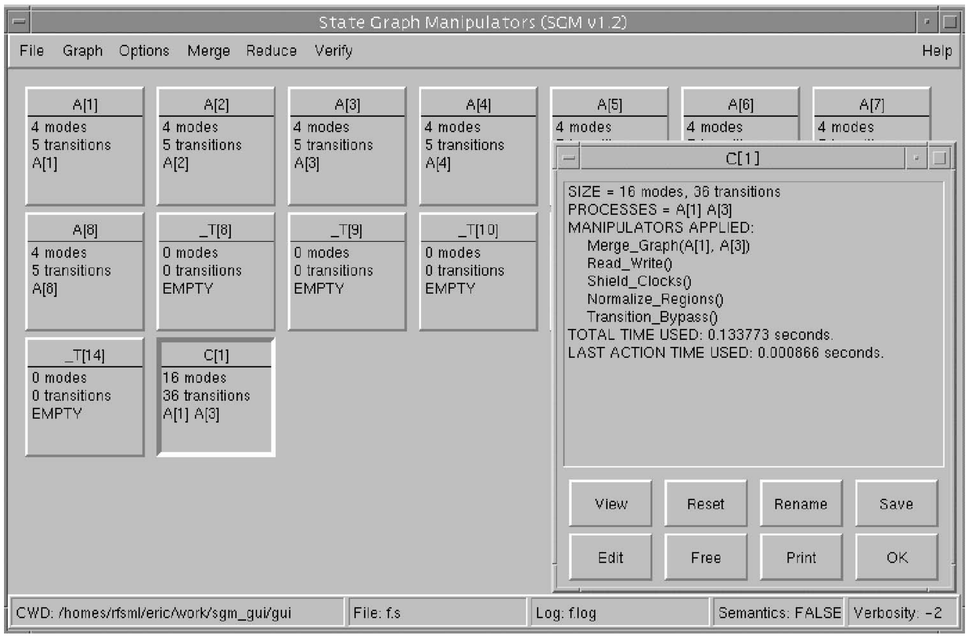
```
verify all_paths
   (henceforth not{(mode(A[1]) = M4) and
   (mode(A[2]) = M4)}
   and not{(mode(A[1]) = M4) and
   (mode(A[3]) = M4)}
   and not{(mode(A[2]) = M4) and
   (mode(A[3]) = M4)});
```

Coming to the final part of a VPL input, *state-graph manipulation*, this part is required only in the batch mode of SGM. It allows the user flexibility in choosing *which* manipulators to apply, *when* to apply, and *in what sequence* to apply. Arguments of manipulators are state-graphs, which are defined in Sections 6 and 7.

In the last part of SGM input, state-graphs are variables which have to be declared first. Then, a list of manipulations follows the declaration. A *simple* manipulation can be either the merging of two state-graphs such as `g[3] := merge_graph(g[1], g[2]);` or the application of a single reduction technique on some state-graph such as `shield_clock(g[3]);` or model-checking a state-graph such as `model_check(g[3]);` or printing a state-graph such as `print_graph(g[3]);`, where `g[1], g[2], g[3]` are all state-graph variables. More complex programming constructs are also provided, such as *for-loops* and *if-then-else* statements, for more dynamic selections of manipulator sequences. These are omitted due to page limits.

TABLE 5
VPL for Fischer's Timed Mutual Exclusion Protocols

```
automata A : i = 1..11;
clock x[1..total(A)]; register lock;
automaton A[i] {
  initially M1 and lock = 0 and x[i] = 0;
  mode M1 { invariant True;
    when lock = 0 may goto M2; x[i] := 0;
  }
  mode M2 { invariant x[i] < 1;
    when x[i] < 1 may goto M3; x[i] := 0; lock := i;
  }
  mode M3 { invariant True;
    when x[i] >= 1 and lock = i may goto M4;
    when ~(lock = i) may goto M1;
  }
  mode M4 { invariant True;
    when True may goto M1; lock := 0;
  }
}
```


State Graph Manipulators (SGM v1.2)
File Graph Options Merge Reduce Verify Help
C[1]
View Reset Rename Save
Edit Free Print OK

```
A[1]          A[2]          A[3]          A[4]          A[5]          A[6]          A[7]
4 modes       4 modes       4 modes       4 modes       4 modes       4 modes       4 modes
5 transitions 5 transitions 5 transitions 5 transitions
A[1]          A[2]          A[3]          A[4]

A[8]          _T[8]         _T[9]         _T[10]
4 modes       0 modes       0 modes       0 modes
5 transitions 0 transitions 0 transitions 0 transitions
A[8]          EMPTY         EMPTY         EMPTY

_T[14]        C[1]
0 modes       16 modes
0 transitions 36 transitions
EMPTY         A[1] A[3]
```

```
SIZE = 16 modes, 36 transitions
PROCESSES = A[1] A[3]
MANIPULATORS APPLIED:
    Merge_Graph(A[1], A[3])
    Read_Write()
    Shield_Clocks()
    Normalize_Regions()
    Transition_Bypass()
TOTAL TIME USED: 0.133773 seconds.
LAST ACTION TIME USED: 0.000866 seconds.
```

```
CWD: /homes/rfsml/eric/work/sgm_gui/gui | File: f.s | Log: f.log | Semantics: FALSE | Verbosity: -2
```

Fig. 7. SGM Graphical User Interface.

Continuing with Fischer's mutual exclusion protocol example, the state-graph manipulation is as follows:

```
manipulation
 graph g[1..total(A)-1];
    - new graph declarations
 for(i:=1; i<total(A); i++) {
    - for each declared graph g[i]
  if(i = 1) { g[1] := merge_graph(A[1], A[2]); }
  else { g[i] := merge_graph(g[i-1], A[i+1]); }
  shield_clock(g[i]);
  symmetry(g[i]); }
 model_check(g[total(A)-1]);
    - model-check global graph
 print_graph(g[total(A)-1]);
    - print global graph
```

8.2 Graphical User Interface

Besides a human-readable input language, user-friendly verification is achieved by SGM through: 1) three user interfaces, including graphical, interactive, and batch, 2) two manipulation modes. A user has to first create a text input file using VPL (Section 8.1).

In the graphical mode, as shown in Fig. 7, after a user loads an input file, SGM displays the system as a set of boxes, where each box represents a state-graph. Each box has some basic visible information, including its size and component processes. Detailed information of each state-graph (box) can be obtained by opening the boxes. After selecting two state-graphs (boxes), one can merge them through a *Merge Graph* command in the SGM menu and a new state-graph (box) is created which represents their merger. A state-graph (box) can be selected for applying any manipulator in the SGM Reduce menu. The manipulators will be described in detail in the next section.

9 Experiments

We perform two experiments on SGM. First, we run SGM against several academic as well as industrial examples, with various reducer sequences, to show that it is possible for nonexpert users to easily come up with their own verification strategy with SGM. Second, we compare SGM's performance with two other popular model-checkers for real-time systems, that is, UPPAAL and Kronos, to show that, even for nonexperts, reasonable performance can be achieved in SGM.

9.1 Application Examples

We present the results of five application examples which illustrate the benefit of allowing user-flexibility in manipulating state-graphs and of the high-level perception of system verification. On experimentation, we observe that different verification tasks require different manipulator sequences to best reduce the intermediate state-graphs and to help in saving the most computing resources, such as processor time and memory space. While, for some verification tasks, a manipulator may not be applicable or compatible, there are also tasks for which a manipulator not only provides no reduction, but also consumes computing resources without any benefit. Since SGM allows a user complete control over what kind of manipulators to apply or not to apply, the user can find the best manipulator sequences suited for a particular verification task.

The five application examples presented here include:

1. Fischer's timed mutual exclusion protocol (FMEP) [28], [1], [32],
2. a graphical user interface of a calculator,
3. Carrier Sense, Multiple Access with Collision Detection (CSMA/CD) network communication protocol [39], [26], [37],

TABLE 6
Performance Data for Fischer's Mutual Exclusion Protocol Example (11 Processes)

	#Modes / #Transitions / Construction Time (sec)							
n	2	3	4	5	6	8	10	11
A	70 / 160 / 0.06	1191 / 3988 / 3	22781 / 105125 / 1343	O/M	O/M	O/M	O/M	O/M
B	31 / 70 / 0.07	184 / 594 / 1.1	1033 / 4316 / 19.1	5804 / 28451 / -	O/M	O/M	O/M	O/M
C	36 / 82 / 0.07	224 / 859 / 6.1	1509 / 8107 / 229	O/M	O/M	O/M	O/M	O/M
D*	16 / 35 / 0.07	39 / 109 / 0.6	76 / 247 / 3	130 / 472 / 10.7	204 / 810 / 35	424 / 1944 / 276	760 / 3937 / -	219 / 1063 / -
E	16 / 35 / 0.08	39 / 109 / 0.9	76 / 247 / 4.7	130 / 472 / 18	204 / 810 / 61	424 / 1944 / 487	760 / 4786 / -	219 / 1174 / -

A: {mg, rw}, **B**: {mg, rw, sm}, **C**: {mg, sm}, **D**: {mg, rw, sc, sm}, **E**: {mg, rw, sm, sc}
O/M: Out of Memory, mg = merge_graph(), rw = read_write(), sc = shield_clock(), sm = symmetry()

The "*" at row **D** means combination **D** is with the best performance in terms of the smallest state-space size and time.

4. priority-based task execution control mechanism in the PATHO real-time operating system [38], [6], [5], and
5. ring network token passing.

9.1.1 Fischer's Timed Mutual Exclusion Protocol

This protocol [28], [1], [32] was used for illustration throughout the article. (Check Fig. 1.) We applied SGM to the protocol with 11 processes. In Table 6, we show the experiment results with three different reduction sequences applied after each intermediate state-graph is generated from merging. Each column represents the state-graph sizes (in #Modes: number of modes and #Transitions: number of transitions) and CPU time used to generate the state-graphs for the corresponding number of processes. Note, in all rows, the numbers in the last column drop drastically from those in the column next to last. This is because, once all processes are merged, the causality can be fully determined and a lot of assumptions of variable values in the regions of the global state-graph can then be determined as unfounded.

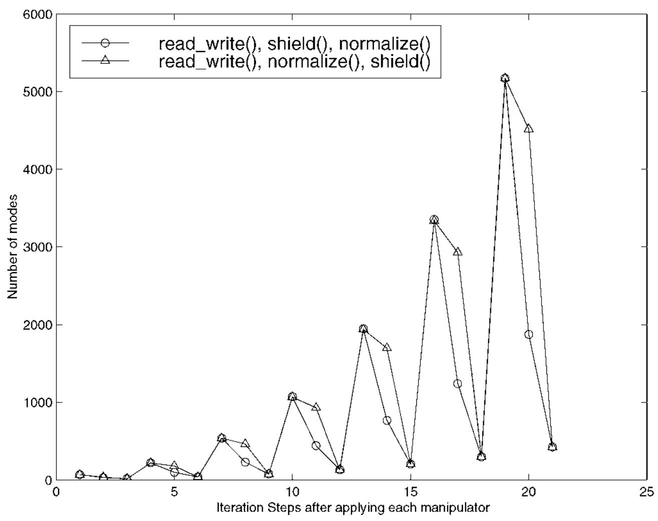


Fig. 8. Performance chart for FMEP example (mode comparison).

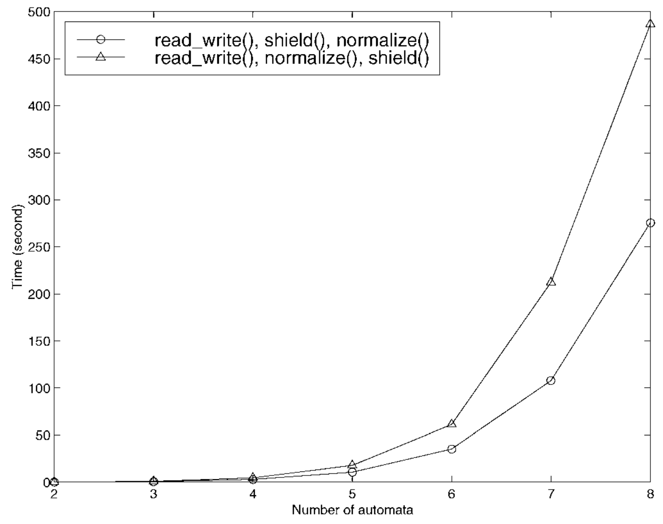


Fig. 9. Performance chart for FMEP example (time comparison).

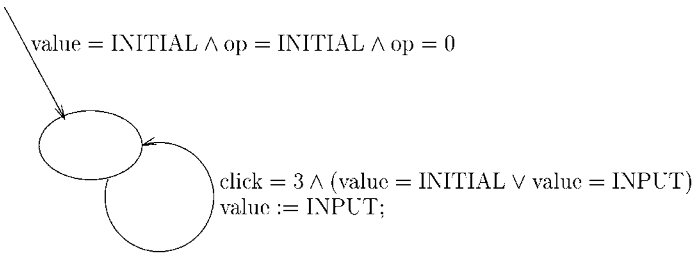

value = INITIAL ∧ op = INITIAL ∧ op = 0
click = 3 ∧ (value = INITIAL ∨ value = INPUT)
value := INPUT;

Fig. 10. Automaton for rule "3.Click; value=INITIAL,INPUT;⇒ value=INPUT;".

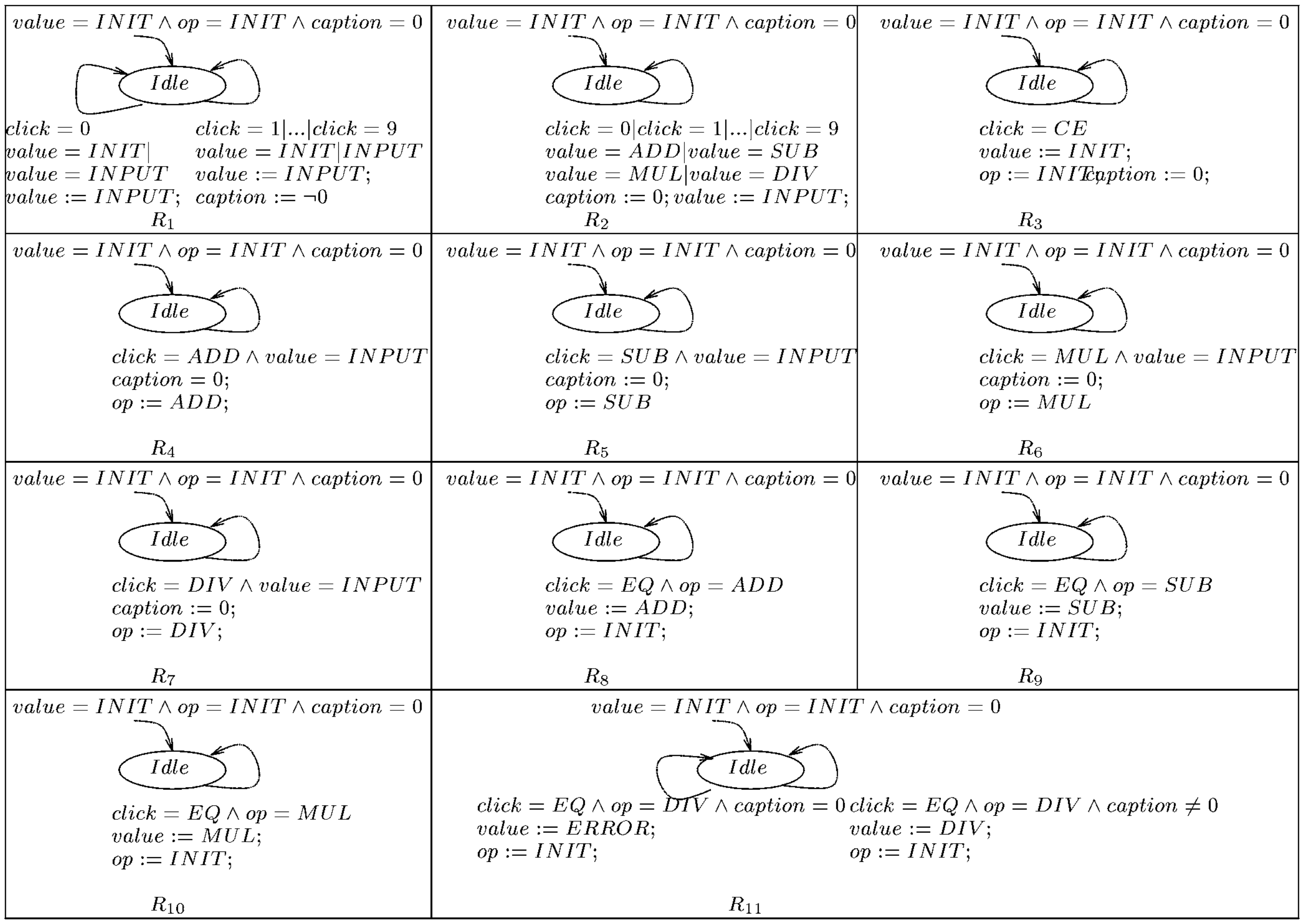


Fig. 11. Calculator GUI example.

Note the little difference between reduction sequences (D) and (E) due to the alteration of ordering of reducer application. Although both sequences result in the same final effect, that is, they reduce the intermediate state-graphs to the same size, the *decrease rate* is not the same. The first sequence decreases the state-graph sizes more quickly than the second sequence. This is observable from Fig. 8. Further, comparing the time taken by the two sequences for state-graph reductions, we see from Fig. 9 that it is also the first sequence that uses a shorter time. Thus, we conclude the first sequence is a better manipulation of the state-graphs.

9.1.2 Graphical User Interface for a Simple Calculator

This is a real project example from the Institute of Information Science, Academia Sinica, Taiwan. The project goal was to develop a generator of graphical user interfaces (GUI). The example considered here is a GUI for a simple calculator. The generator created a set of condition/action rules governing the behavior of a calculator GUI. Due to the large number of rules, it was difficult to verify if a resulting GUI behaved in the same way as a real calculator. It was also difficult to comprehend how large the state space would be and how the state space could be reduced. Thus, SGM came in handy in such a situation. We collaborated with the project members to verify the GUI rules created by their generator.

The set of rules was transformed into a corresponding set of timed automata and input to SGM. Each rule was modeled by a single timed automaton with one mode and one or more looping transitions. The rule condition was mapped to a triggering condition of the transitions. The

TABLE 7
Performance Data for Rules of Calculator GUI (Version 1)

n	$\frac{\#\textbf{modes}}{\#\textbf{transitions}}$										**time**
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	(sec)
A	$\frac{11}{660}$	$\frac{11}{671}$	$\frac{22}{1364}$	$\frac{33}{2079}$	$\frac{44}{2816}$	$\frac{55}{3575}$	$\frac{69}{4554}$	$\frac{83}{5561}$	$\frac{97}{6596}$	$\frac{90}{1229}$	12.9
B	$\frac{11}{650}$	$\frac{11}{561}$	$\frac{21}{1091}$	$\frac{31}{1641}$	$\frac{41}{2211}$	$\frac{51}{2801}$	$\frac{61}{2711}$	$\frac{71}{2421}$	$\frac{81}{1931}$	$\frac{90}{1229}$	27.5
C	$\frac{11}{650}$	$\frac{11}{561}$	$\frac{21}{1091}$	$\frac{31}{1641}$	$\frac{41}{2211}$	$\frac{46}{2526}$	$\frac{55}{2444}$	$\frac{64}{2182}$	$\frac{73}{1740}$	$\frac{83}{1133}$	32.6

A: {mg}, **B**: {mg, rw}, **C**: {mg, rw, sv}, mg = merge_graph(), rw = read_write(), sv = shield_variables()

TABLE 8
Performance Data for Rules of Calculator GUI (Version 2)

n	$\frac{\#\textbf{modes}}{\#\textbf{transitions}}$										**time**
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	(sec)
A	$\frac{3}{36}$	$\frac{3}{39}$	$\frac{6}{84}$	$\frac{9}{135}$	$\frac{12}{192}$	$\frac{15}{255}$	$\frac{21}{378}$	$\frac{27}{513}$	$\frac{33}{660}$	$\frac{18}{101}$	0.68
B	$\frac{3}{34}$	$\frac{3}{33}$	$\frac{5}{59}$	$\frac{7}{89}$	$\frac{9}{123}$	$\frac{11}{161}$	$\frac{13}{159}$	$\frac{15}{149}$	$\frac{17}{131}$	$\frac{18}{101}$	0.94
C	$\frac{3}{34}$	$\frac{2}{22}$	$\frac{2}{23}$	$\frac{3}{37}$	$\frac{4}{53}$	$\frac{5}{71}$	$\frac{7}{84}$	$\frac{8}{78}$	$\frac{9}{68}$	$\frac{11}{61}$	1.00

A: {mg}, **B**: {mg, rw}, **C**: {mg, rw, sv}, mg = merge_graph(), rw = read_write(), sv = shield_variables()

action part was mapped to a set of transition assignment statements. An example automaton for rule

3.Click; value=INITIAL,INPUT;⇒ value=INPUT;

is shown in Fig. 10. The rule says that if button "3" is clicked and "value" equals to "INITIAL" or "INPUT," then value is assigned "INPUT."

The set of automata obtained from the rules are shown in Fig. 11. We experimented with two versions of the GUI rule set: one with a distinction between each number input $(0, 1, \ldots, 9)$ and another with a distinction between only zero and nonzero numbers. The second version is a correct model because only one of the rules (a division rule) required the operand (number input) to be nonzero, while the other rules made no distinction among the input numbers.

On applying the manipulators **read_write()** and **shield_ variables()** after each **merge()**, we found a significant reduction in state-graph sizes. For the first version, the reduction was as much as 73.6 percent for transitions and 24.7 percent for modes. For the second version, the reduction was as much as 89.7 percent for transitions and 72.7 percent for modes. More detailed readings are tabulated in Tables 7 and 8 for versions 1 and 2, respectively.

9.1.3 CSMA/CD Network Communication Protocol

This protocol [39], [26], [37] resolves the competition between several message senders using a multiaccess channel. As shown in Fig. 12, whenever two or more senders send their messages about the same time, they all detect a collision, wait a random amount of time, and retransmit their messages. If σ is the largest propagation delay, then a sender can be sure that there will be no collision if none is detected within 2σ. For example, on a 10Mbps Ethernet, the worst case round trip propagation delay is $2\sigma = 51.2\mu s$. We need to verify that, when one sender begins transmitting, there always exists a computation that leads to a successful transmission. As shown in Table 9, each of the manipulators implemented in SGM was applied to this protocol example.

We observed that there were intriguing interactions among the manipulators. For example, if **read_write()** was *applied before* **symmetry()** (sequences (A), (B), (F), and (I)), the final state-graph sizes were significantly smaller than if **read_write()** was *not applied before* **symmetry()** (sequences (C), (D), (E), (G), (H), (J), and (N)). This is due to **read_write()** creating an invariant literal set associated with each node in a state-graph and these literal sets can be used by **symmetry()** to achieve a greater reduction. Furthermore, on one hand, if **read_write()** was applied before other manipulators (sequences (A) and (B)), it turned out that applying **symmetry()** before applying **shield_clock()** produced better results in terms of greater reductions obtained. On the other hand, if **read_write()** was not applied (sequences (G) and (H)) or was applied last (sequences (C) and (D)), then the opposite holds true, that is, applying

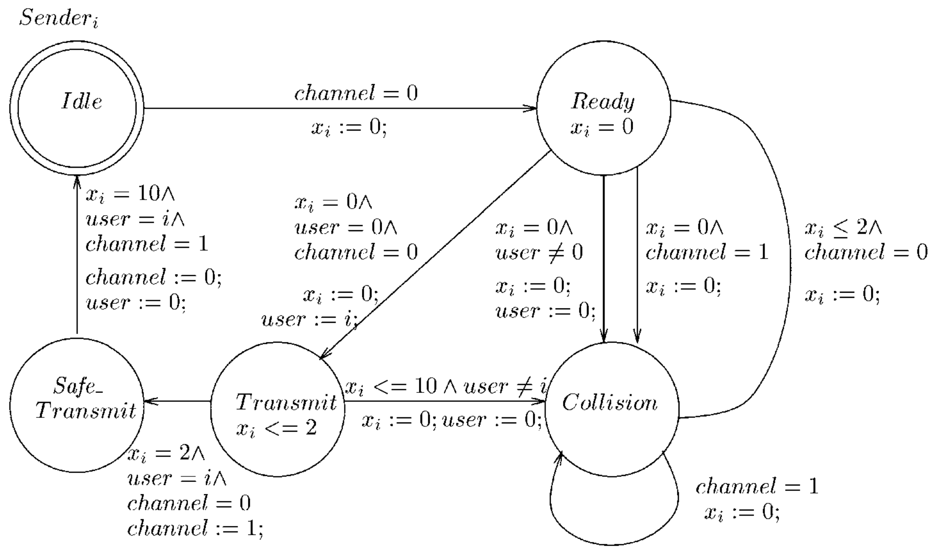


Fig. 12. CSMA/CD network protocol (ith sender).

shield_clock() before applying **symmetry()** gave greater reduction results.

After experimenting with all possible sequences of manipulators, we found that the best sequence (as evident from Table 9) is sequence (B), that is, {**merge_graph(), read_write(), symmetry(), shield_clock()**}, which is different from that for FMEP.

9.1.4 PATHO Task Execution

In contrast to the above two examples, this is an *asymmetric* system since task execution in the real-time operating system PATHO [38], [6], [5] is based on task *priority*. Each task has an index i, with the smallest index having the highest priority. As shown in Fig. 13, a task executes (enters *Run* mode) only if no other higher priority tasks are pending and no task is currently running. Otherwise, the task is said to be pending (enters the *Pend* mode). Each task needs one time unit for execution and two instances of the task are separated by at least 20 time units. A task is said to be *dead* if there is not enough time for it to finish execution before another instance of the same task starts. We need to verify that, for less than 20 tasks, all tasks can be executed in a timely fashion.

Due to asymmetricity, **symmetry()** is not applicable here. As shown in Table 10, on applying the two manipulators, **read_write()** and **shield_clock()**, we observed that their order

TABLE 9
Performance Data for CSMA/CD Communication Protocol

n	#modes/#transitions			time
	2	**3**	**4**	(sec)
A	56/192	582/2999	1238/3813	170
B*	56/192	579/2996	1219/3751	175
C	57/216	743/4770	4370/24993	594
D	57/216	749/4792	4361/24981	488
E	57/216	756/4788	4417/25400	618
F	56/193	565/2717	1240/3824	157
G	80/342	1128/7796	4312/24147	432
H	80/341	1132/7890	4335/24417	550
I	73/248	807/4057	1841/5889	238
J	75/281	1115/6985	5660/32136	701
K	111/386	2940/14255	26563/79758	1069
L	111/384	3000/14637	26907/81097	1301
M	145/496	4555/21817	39272/120709	1200
N	101/416	1865/12256	5676/32663	702
O	137/480	3798/18946	26636/80253	602

A: {mg, rw, sc, sm}, **B**: {mg, rw, sm, sc}, **C**: {mg, sm, sc, rw},
D: {mg, sc, sm, rw}, **E**: {mg, sm, rw, sc}, **F**: {mg, sc, rw, sm}, **G**: {mg, sc, sm},
H: {mg, sm, sc}, **I**: {mg, rw, sm}, **J**: {mg, sm, rw}, **K**: {mg, sc, rw},
L: {mg, rw, sc}, **M**: {mg, rw}, **N**: {mg, sm}, and **O**: {mg, sc}
The "*" at row **B** means combination **B** is with the best performance in terms of the smallest state-space size and time.

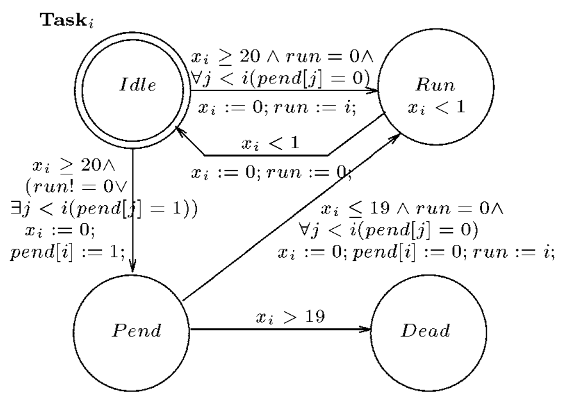

Task$_i$
Idle
Run
$x_i < 1$
Pend
Dead
$x_i \geq 20 \wedge run = 0 \wedge$
$\forall j < i(pend[j] = 0)$
$x_i := 0; run := i;$
$x_i < 1$
$x_i := 0; run := 0;$
$x_i \geq 20 \wedge$
$(run! = 0 \vee$
$\exists j < i(pend[j] = 1))$
$x_i := 0;$
$pend[i] := 1;$
$x_i \leq 19 \wedge run = 0 \wedge$
$\forall j < i(pend[j] = 0)$
$x_i := 0; pend[i] := 0; run := i;$
$x_i > 19$

Fig. 13. PATHO task execution (ith task).

has little consequence as the reductions obtained were approximately the same (sequences (D) and (E)). Further, we also applied **bypass_internal_transition()** and **shield_variables()**, both of which gave no reductions at all in this case. This is intuitive as there are no internal variables in this example, whereas the reductions by those two manipulators depended on the existence of internal variables. Finally, the best sequence obtained for this example was sequence (D), namely, {**merge_graph(), shield_clock(), read_write()**}.

9.1.5 Ring Network

Here, we apply SGM to the timed version of the ring network in Fig. 5. From Table 11, we can observe that the **read_write()** and BIT manipulators in sequence (C) reduce the intermediate state-graph sizes and achieve a greater reduction compared to that in sequence (B) with only **read_write()** applied.

Two versions of the ring network were experimented with with SGM: one timed and one untimed. From Table 12, we can observe that, in this version, the application of BIT (in sequence (C)) results in a greater final reduction than that with only **read_write()** (in sequence (B)). There was no reduction on the intermediate state-graphs.

9.2 Performance Comparison with UPPAAL and Kronos

We compare the performance of SGM, Kronos (version 2.4.4), and UPPAAL (version 3.0.39) in this section against Balarin's version of Fischer's timed mutual exclusion algorithm. The SGM execution is with reduction sequence

TABLE 10
Performance Data for PATHO Task Execution

n	#modes/#transitions			time
	2	**3**	**4**	(sec)
A	45/139	1060/5004	2309/6643	14
B	45/139	1060/5004	2240/6477	21
C	37/89	443/1470	832/2186	9
D*	37/89	412/1450	708/2108	10
E	34/89	376/1450	711/2124	12

A: {mg}, **B**: {mg, sc}, **C**: {mg, rw}, **D**: {mg, sc, rw}, and **E**: {mg, rw, sc}.
mg = **merge_graph()**, rw = **read_write()**, sc = **shield_clock()**, sm = **symmetry()** The "*" at row **D** means combination **D** is with the best performance in terms of the smallest state-space size and time.

TABLE 11
Performance Data for Ring Network (Timed Version)

n	#modes / #transitions / time (sec)						total time
	2	**3**	**4**	**5**	**6**	**7**	(seconds)
A	32 / 64 / 0.02	221 / 663 / 0.2	1364 / 5450 / 2.3	7842 / 38460 / 21.7	42752 / 24445 / 183	28 / 28 / 0.08	208
B	19 / 33 / 0.04	96 / 225 / 0.2	483 / 1397 / 0.17	2492 / 8155 / 18.3	11965 / 42443 / 258	28 / 28 / 0.09	278
C	14 / 28 / 0.05	63 / 202 / 0.29	303 / 1442 / 1.15	1513 / 9794 / 25.1	7782 / 65036 / 351	18 / 18 / 0.07	378

A = {mg}, **B** = {mg, rw}, **C** = {mg, rw, bit}}
mg = **merge_graph()**, rw = **read_write()**, bit = **bypass_internal_transition()**

TABLE 12
Performance Data for Ring Network (Nontimed Version)

n	#modes / #transitions / time (sec)						total time
	2	**3**	**4**	**5**	**6**	**7**	(seconds)
A	20	88	368	1504	6080	14	6
	40	264	1472	7520	36480	14	
	0.01	0.03	0.15	0.85	4.8	0.01	
B	12	36	108	324	972	14	2.04
	20	84	324	1188	4212	14	
	0.02	0.06	0.18	0.60	1.16	0.02	
C	12	36	108	324	972	9	4.35
	24	120	552	2424	10344	9	
	0.02	0.06	0.21	0.91	3.13	0.02	

A = {mg}, **B** = {mg, rw}, **C** = {mg, rw, bit}}
mg = **merge_graph()**, rw = **read_write()**, bit = **bypass_internal_transition()**

TABLE 13
Performance Data of Scalability W.R.T. Number of Processes

benchmarks	concurrency	Kronos	UPPAAL	SGM {mg, rw, sc, sm}
Fischer's mutual exclusion	3 processes	0.01s†	0.05s*	0.6s/1287k$^{\otimes}$
	4 processes	0.08s†	2s*	3s/3302k$^{\otimes}$
	5 processes	0.21s†	286s*	10.7s/8209k$^{\otimes}$
	6 processes	O/M†	O/M*	35s/20940k$^{\otimes}$
	7 processes	O/M†	O/M*	108s/46311k$^{\otimes}$
	8 processes	O/M†	O/M*	276s/98028k$^{\otimes}$
	9 processes	O/M†	O/M*	190705k$^{\otimes}$

†: data collected on a Pentium III 500MHz with 256MB memory running LINUX;
*: data collected on a Pentium II 333MHz with 192MB memory running LINUX;
$^{\otimes}$: data collected on a SUN SPARC 340MHz running solaris;
s: seconds; k: kilobytes of memory in data-structure; O/M: Out of memory;

read-write analysis, shielded clock, and symmetry. As shown in Table 13, it is possible for nonexpert users to benefit from the user-friendly interface of SGM and come up with reasonable verification performance.

10 Automatic Reducer Combination for Efficient Verification

At this moment, we have four reducers in addition to the merge operator $\times$. As mentioned in Section 3, from the literature, there are many reduction algorithms with the potential to become new reducers in SGM. Different reducers may have different impacts on different verification tasks. Moreover, when the number of reducers is large, sometimes it may become difficult for users to pick a good reducer combination to accomplish their verification tasks. In this section, we shall develop an algorithm based on group theory [20], [21] to automatically pick a "locally optimal" combination for a given task.

Suppose we have n reducers: $\mathbf{R}_1, \mathbf{R}_2, \ldots,$ and $\mathbf{R}_n$. We shall simplify the verification procedure construction problem to the verification procedure template in Table 14. Thus, the goal for the construction of efficient verification procedure is restricted to finding a good sequence $i_1, \ldots, i_k$ from the integer interval $[1, n]$ such that the verification procedure costs less space and time.

Before the presentation of our algorithm, we need to clarify what it aims to achieve. In executing a verification task, there can be a trade-off between space and time requirements. By dynamically deducing information while needed and deleting it while not, we can save a lot of memory space. But, repetitively and dynamically creating the same piece of information will certainly take up a lot of CPU time. However, we believe, for verification tasks, space management is more important than time management because of the state-space explosion phenomenon. Most verification tasks quickly run out of memory instead of taking too long to complete. Thus, our algorithm will pick a reducer sequence with predicted "locally minimal" space requirement.

In the following, we shall first present a structure for reducer sequence groups. Then, based on the structure, we shall define local optimality of reducer sequences and present an algorithm for predicting a locally most efficient reducer sequence.

Structure of Reducer Sequence Groups. Given a set $\{\mathbf{R}_1, \ldots, \mathbf{R}_n\}$ of n reducers, a *reducer sequence* is a sequence like $\mathbf{R}_{i_1}\mathbf{R}_{i_2}\ldots\mathbf{R}_{i_k}$ such that $|\{i_1, \ldots, i_k\}| = k$ and

TABLE 14
A Verification Procedure Template

```
verify(G, m, φ)
/* Assume we use reducers $\mathbf{R}'_1, \ldots, \mathbf{R}'_k$ out of library of $\mathbf{R}_1, \ldots, \mathbf{R}_n$. */
state_graph *G; /* the input state-graph array. */
int m; /* the number of input state-graphs. */
CTL φ; /* the specification */
{
  state_graph H;                                                        (1)
  int i;                                                                (2)

  $H := G[1]$;                                                          (3)
  for $i := 2$ to $m$, do                                               (4)
    $H := \mathbf{R}'_1(\mathbf{R}'_2(\ldots(\mathbf{R}'_i(H \times G[i]))\ldots))$;   (5)
  return check(H, φ);                                                   (6)
}
```

$\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. Suppose we are given a reducer sequence $\gamma = \mathbf{R}_{i_1}\mathbf{R}_{i_2}\ldots\mathbf{R}_{i_k}$. A *binary permutation* (j, j') on γ is a pair of integers such that $1 \leq j < j' \leq k$ and denotes an operation on γ which switches the position of $\mathbf{R}_{i_j}$ and $\mathbf{R}_{i_{j'}}$ in the sequence. Formally speaking, $\gamma(j, j') = \mathbf{R}_{\hat{i}_1}\mathbf{R}_{\hat{i}_2}\ldots\mathbf{R}_{\hat{i}_k}$ such that $\mathbf{R}_{\hat{i}_j} = \mathbf{R}_{i_{j''}}$, $\mathbf{R}_{\hat{i}_{j'}} = \mathbf{R}_{i_j}$, and, for all $1 \leq h \leq k$ with $h \neq j$ and $h \neq j'$, $\mathbf{R}_{\hat{i}_h} = \mathbf{R}_{i_h}$.

By group theory [20], [21], it is known that every permutation can be constructed as a sequential composition of binary permutations. This further implies that, for any two reducer sequences γ, γ' composed of the same set of reducers, there is a finite sequence $\theta_1 \ldots \theta_h$ of binary permutations such that $\gamma\theta_1\theta_2 \ldots \theta_h = \gamma'$. For convenience, we adopt left-associativity to interpret the ordering of permutation operations. Thus, all reducer sequences composed of the same set of reducers form a connected graph.

We now have to define operations between reducer sequences composed of different sets of reducers. This can be done by the *append* operation. Given reducer sequence $\gamma = \mathbf{R}_{i_1}\mathbf{R}_{i_2}\ldots\mathbf{R}_{i_k}$ and a reducer $\mathbf{R} \notin \{\mathbf{R}_{i_1}, \mathbf{R}_{i_2}, \ldots, \mathbf{R}_{i_k}\}$, $\gamma\mathbf{R}$ is exactly the new reducer sequence $\mathbf{R}_{i_1}\mathbf{R}_{i_2}\ldots\mathbf{R}_{i_k}\mathbf{R}$.

The following lemma depicts the structures of reducer sequence groups. Given a sequence F, we let $[F]$ be the set of elements used in F.

lemma 3. *Given a set $\{\mathbf{R}_1, \ldots, \mathbf{R}_n\}$ of reducers, for any two reducer sequences γ, γ' constructed from the set, there is a sequence $\lambda = \gamma_1\gamma_2 \ldots \gamma_k$ of reducer sequences such that*

- $\gamma = \gamma_1$,
- $\gamma' = \gamma_k$, *and*
- *for all $1 \leq i < k$, one of the following three is true:*
  - *for some $1 \leq j < j' \leq k$, $\gamma_i(j, j') = \gamma_{i+1}$ or*
  - *for some $\mathbf{R} \in \{\mathbf{R}_1, \ldots, \mathbf{R}_n\} - [\gamma_i]$, $\gamma_i\mathbf{R} = \gamma_{i+1}$ or*
  - *for some $\mathbf{R} \in \{\mathbf{R}_1, \ldots, \mathbf{R}_n\} - [\gamma_{i+1}]$,*

$$\gamma_i = \gamma_{i+1}\mathbf{R}.$$

With the operations of binary permutations and appending, we know that we can draw an undirected *reducer sequence graph (RSG)* for a given set of reducers. The nodes in an RSG are reducer sequences, while the arcs are determined by whether the two nodes can be related by a binary permutation or an appending operation. In Fig. 14, we have an RSG for three reducers. Lemma 3 says that such a graph is connected.

Two reducer sequences in an RSG are called *neighbors* to each other if we can go from one to the other by a binary permutation or an appending operation. Suppose we have a valuation $\mathcal{V}$ on all reducer sequences in an $\text{RSG} = (\Gamma, \Omega)$ such that Γ is the set of nodes (reducer sequences), Ω is the set of edges, and, for all $\gamma \in \Gamma$, $\mathcal{V}(\gamma) \in \mathbf{R}^+$. A reducer sequence γ in the RSG is called a *local minimum* if $\mathcal{V}(\gamma) \leq \mathcal{V}(\gamma')$ for every neighbor γ' of γ in the RSG.

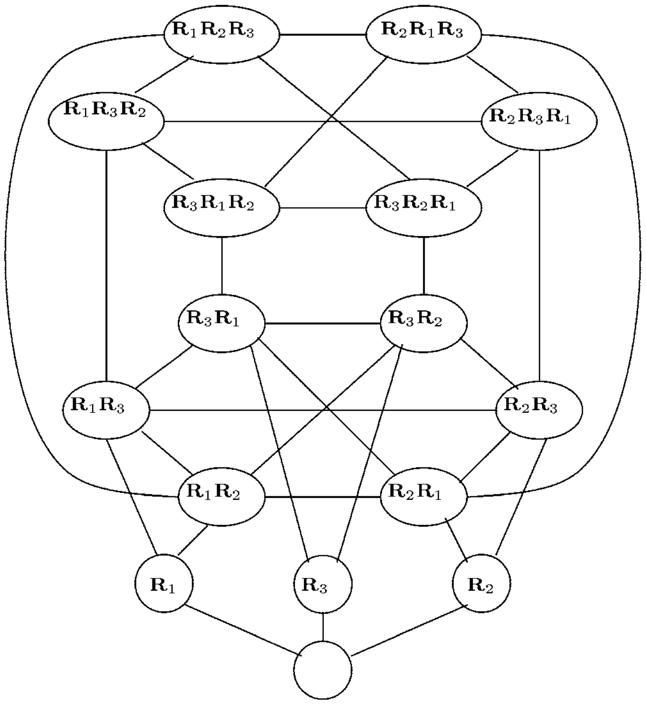

$\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3$
$\mathbf{R}_2\mathbf{R}_1\mathbf{R}_3$
$\mathbf{R}_1\mathbf{R}_3\mathbf{R}_2$
$\mathbf{R}_2\mathbf{R}_3\mathbf{R}_1$
$\mathbf{R}_3\mathbf{R}_1\mathbf{R}_2$
$\mathbf{R}_3\mathbf{R}_2\mathbf{R}_1$
$\mathbf{R}_3\mathbf{R}_1$
$\mathbf{R}_3\mathbf{R}_2$
$\mathbf{R}_1\mathbf{R}_3$
$\mathbf{R}_2\mathbf{R}_3$
$\mathbf{R}_1\mathbf{R}_2$
$\mathbf{R}_2\mathbf{R}_1$
$\mathbf{R}_1$
$\mathbf{R}_3$
$\mathbf{R}_2$
Fig. 14. RSG for three reducers.

TABLE 15
Our Reducer Sequence Picking Algorithm

```
**good_sequence**$(G, m)$ /* Assume we have reduce operators $\mathbf{R}_1, \ldots, \mathbf{R}_n$. */
**state_graph** $*G$; /* the input state-graph array. */
**int** $m$; /* the number of input state-graphs. */
{
  **state_graph** $H1, H2, H3$;
  Randomly pick a reducer sequence $\gamma$ from $\mathbf{R}_1, \ldots, \mathbf{R}_n$.
  Repeat forever {
    $B_\gamma :=$ $\{\gamma(j, j') \mid 1 \leq j < j' \leq |\gamma|\}$ ;
        $\cup$ $\{\gamma' \mid \exists \mathbf{R} \in \{\mathbf{R}_1, \ldots, \mathbf{R}_n\} - [\gamma'](\gamma = \gamma'\mathbf{R})\}$
        $\cup$ $\{\gamma\mathbf{R} \mid \mathbf{R} \in \{\mathbf{R}_1, \ldots, \mathbf{R}_n\} - [\gamma]\}$
    Pick $\gamma' \in B_\gamma$ such that for all $\gamma'' \in B_\gamma$, $\mathcal{V}_s(\gamma') \leq \mathcal{V}_s(\gamma'')$;
    If $\mathcal{V}_s(\gamma) < \mathcal{V}_s(\gamma')$, **then** return $\gamma$;
    else if $\mathcal{V}_s(\gamma) > \mathcal{V}_s(\gamma')$, **then** $\gamma := \gamma'$;
    else if $\mathcal{V}_t(\gamma) \leq \mathcal{V}_t(\gamma')$, **then** return $\gamma$;
    else $\gamma := \gamma'$;
  }
}
```

Our algorithm to pick reducer sequences for efficient verification will use memory-space consumption increase rate with respect to concurrency as our valuation $\mathcal{V}$.

10.1 Algorithm to Pick a Locally Most Efficient Sequence

Suppose we are given a concurrent system presented as m state-graphs $G[1], \ldots, G[m]$, with $m > 4$, and reducers $\mathbf{R}_1, \ldots, \mathbf{R}_n$. Our strategy is to predict the space-complexities of reducer sequences by testing the procedure template on four state-graphs ($G[1], G[2], G[3], G[4]$ or some other four picked by users). Since the RSG has size of order n factorial, it is not feasible to test all the reducer sequences. Our algorithm hinges on the definition of valuations on reducer sequences which reflects how fast the memory space rate and CPU-time consumption rate grow. It will randomly generate a reducer sequence and then start searching the RSG. The search stops when it reaches a local minimal reducer sequence.

There will be two valuations in our method: T^he major one is for space consumption and the minor one is for CPU-time consumption. We need the minor one because our reducers do not increase the sizes of its argument state-graphs. Thus, a naive reducer sequence which leads to minimal space consumption is the sequence of all reducers. However, for a given verification tasks, some reducers may be applied with no effect on the state-graph while still consuming a huge amount of CPU-time. Thus, it is better if we can also use CPU-time as a minor valuation.

Both the major and the minor valuations are devised on the same idea. Since verification problems usually exhibit at least singly exponential space complexity with respect to concurrency, our algorithm attempts to use the predictions of how fast the exponent base grows as an indication of the memory consumption. If two reducer sequences have the same prediction, then we choose the one with less CPU-time consumption. We define the reduced state-graph H_h^γ inductively with reducer sequence γ after each iteration.

- $H_2^\gamma = \gamma(G[1] \times G[2])$ and
- for each $h > 2$, $H_h^\gamma = \gamma(H_{h-1} \times G[h])$.

Our algorithm uses the following major valuation $\mathcal{V}^\gamma$ to predict the space consumption complexity of a reducer sequence γ:

$$\mathcal{V}_s(\gamma) = \frac{\frac{\mathbf{size}(H_4^\gamma)}{\mathbf{size}(H_3^\gamma)}}{\frac{\mathbf{size}(H_3^\gamma)}{\mathbf{size}(H_2^\gamma)}} = \frac{(\mathbf{size}(H_4^\gamma))(\mathbf{size}(H_2^\gamma))}{(\mathbf{size}(H_3^\gamma))^2}.$$

The design of $\mathcal{V}_s(\gamma)$ has the following intuitive: In most verification tasks, complexities are super-exponential. Since we want to predict the complexities of reducer sequences for high concurrency with only composition of up to four processes, we decide to use the "multiplication rate of multiplication rate" as the predicting valuation. The denominator $\frac{\mathbf{size}(H_3^\gamma)}{\mathbf{size}(H_2^\gamma)}$ of the first fraction is the multiplication rate of space requirement from concurrency two to three. The numerator $\frac{\mathbf{size}(H_4^\gamma)}{\mathbf{size}(H_3^\gamma)}$ of the first fraction is the multiplication rate of space requirement from concurrency three to four. The fraction is thus an indication of how fast the space consumption rate multiplies. For most verification tasks for concurrent systems, verification on up to four processes usually takes very little memory space and CPU-time. Thus, $\mathcal{V}_s(\gamma)$ should be able to give a good prediction on space complexity.

The minor valuation $\mathcal{V}_t(\gamma)$ is defined similarly in the following way to predict how fast the CPU-time consumption complexity of a reducer sequence γ multiplies:

$$\mathcal{V}_t(\gamma) = \frac{\frac{\mathbf{Time}(H_4^\gamma)}{\mathbf{Time}(H_3^\gamma)}}{\frac{\mathbf{Time}(H_3^\gamma)}{\mathbf{Time}(H_2^\gamma)}} = \frac{(\mathbf{Time}(H_4^\gamma))(\mathbf{Time}(H_2^\gamma))}{(\mathbf{Time}(H_3^\gamma))^2}.$$

Table 15 is our algorithm for picking a reducer sequence for efficient verification. The termination of its execution is guaranteed because the RSG is finite and the search stops when γ is no greater than its neighbors with respect to $\mathcal{V}_s()$ and $\mathcal{V}_t()$. In Section 10.2, we will have experimental data to justify our algorithm.

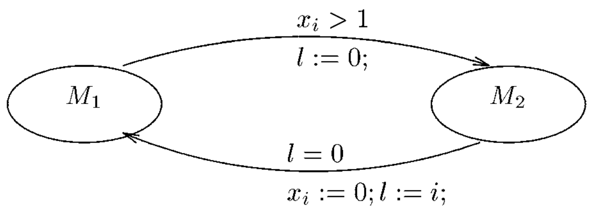


Fig. 15. A small mutual exclusion protocol.

Before we leave this section, there is a legitimate question to ask: *Why do we only use four processes to predict the complexity?* Indeed, we can use a larger number of processes to make more accurate prediction with redefinition of $\mathcal{V}_s(\gamma)$ as, say,

$$\mathcal{V}_s(\gamma) = \frac{\frac{\textbf{size}(H_{10}^{\gamma})}{\textbf{size}(H_9^{\gamma})}}{\frac{\textbf{size}(H_9^{\gamma})}{\textbf{size}(H_8^{\gamma})}} = \frac{(\textbf{size}(H_{10}^{\gamma}))(\textbf{size}(H_8^{\gamma}))}{(\textbf{size}(H_9^{\gamma}))^2}.$$

But, this prediction will cost more space and CPU-time to calculate. And, for a lot of inefficient reducer sequences, they may already run out of memory before the construction of H_8^{γ}. Thus, "four" is a safe minimum which allows the reducer sequences to complete their predictions.

10.2 Experiment

We test our method on Fischer's timed mutual exclusion protocol in Fig. 1 and the simple mutual exclusion protocol in Fig. 15. In Tables 16 and 17, we have shown the graph sizes in mode counts, transition counts, and their sums for small mutual exclusion protocol (Fig. 15) and the Fischer's protocol (Fig. 1), respectively.

Now, we compare the data with our predicting valuation $\mathcal{V}_s()$. In Table 16, for reducer sequence A, B, C,D, and E,

$$\mathcal{V}_s(A) = \frac{42 \times 949}{222^2} \approx 0.809,$$
$$\mathcal{V}_s(B) = \frac{33 \times 1032}{200^2} \approx 0.851,$$
$$\mathcal{V}_s(C) \approx 0.810,$$
$$\mathcal{V}_s(D) \approx 0.789,$$

TABLE 16
Performance Data of Reducer-Sequence Picking Algorithm for Small Mutual Exclusion Protocol Example

	#Modes/#Transitions (#Modes+#Transitions)/(Construction Time (seconds))						
n	2	3	4	5	6	7	8
A	14/28 42/0.01	57/165 222/0.11	203/746 949/0.66	725/3040 3765/3.75	2646/12003 14649/20.81	10469/49411 59880/126.95	O/M
B	11/22 33/0.02	51/149 200/0.11	215/817 1032/1.00	850/3759 4609/8.07	3311/15864 199175/74.91	13430/66567 79997/1025.10	O/M
C	8/14 22/0.02	15/35 50/0.09	24/68 92/0.30	35/116 151/0.73	46/171 217/1.36	60/251 311/2.58	76/357 433/4.64
D	6/11 17/0.02	12/29 41/0.10	20/58 78/0.32	30/100 130/0.84	42/157 199/1.88	56/231 287/3.82	72/324 396/7.24
E	6/11 17/0.02	12/29 41/0.10	20/58 78/0.30	30/100 130/0.78	42/157 199/1.74	56/231 287/3.43	72/324 396/6.61

A: ×, **readwrite**(); B: ×, **readwrite**(), **shield**(); C: ×, **readwrite**(), **symmetry**()
D: ×, **readwrite**(), **shield**(), **symmetry**(); E: ×, **readwrite**(), **symmetry**(), **shield**(); O/M: Out of Memory

TABLE 17
Performance Data of Reducer-Sequence Picking Algorithm for Fischer's Mutual Exclusion Protocol Example

	#Modes/#Transitions (#Modes+#Transitions)/(Construction Time (seconds))						
n	2	3	4	5	6	7	8
A	70/160 230/0.06	1239/4013 5252/3.20	28593/120850 149443/218.65	O/M	O/M	O/M	O/M
B	31/70 101/0.07	182/578 760/0.95	1024/4149 5173/12.85	5804/28451 34255/204.99	O/M	O/M	O/M
C	36/80 116/0.08	224/743 967/3.30	1270/6357 7627/185.80	O/M	O/M	O/M	O/M
D	16/35 51/0.07	39/109 148/0.61	76/247 323/2.92	130/472 602/10.69	204/810 1014/35.25	301/1290 1591/107.84	424/1944 2368/275.61
E	16/35 51/0.08	39/109 148/0.89	76/247 323/4.66	130/472 602/17.87	204/810 1014/61.19	301/1290 1591/212.40	424/1944 2368/486.67

A: ×, **readwrite**(); B: ×, **readwrite**(), **shield**(); C: ×, **readwrite**(), **symmetry**()
D: ×, **readwrite**(), **shield**(), **symmetry**(); E: ×, **readwrite**(), **symmetry**(), **shield**();
O/M: Out of Memory

and $\mathcal{V}_s(E) \approx 0.789$. As we can see, the valuation indeed gives a good prediction of how fast the multiplication rate grows. It is observed that, by permutating the shield() and the symmetry() reducer, there is no difference made with respect to the space complexity prediction. Thus, we can further calculate $\mathcal{V}_t(D) = \frac{0.02\times 0.32}{0.10^2} = 0.64$ and $\mathcal{V}_t(E) = 0.6$. And, as we see from Table 16, reducer sequence E indeed looks likely to have the smallest space and time complexity.

In Table 17, for reducer sequence A, B, C,D, and E

$$\mathcal{V}_s(A) = \frac{230 \times 149443}{5252^2} \approx 1.246,$$
$$\mathcal{V}_s(B) = \frac{101 \times 5173}{760^2} \approx 0.905,$$
$$\mathcal{V}_s(C) \approx 0.946,$$
$$\mathcal{V}_s(D) \approx 0.752,$$

and $\mathcal{V}_s(E) \approx 0.752$. Again, the valuation indeed gives a good prediction of how fast the multiplication rate grows. Since reducer sequence D and E tie in space complexity predication, we further calculate $\mathcal{V}_t(D) = \frac{0.07\times 2.92}{0.61^2} \approx 0.549$ and $\mathcal{V}_t(E) = 0.47$. It seems that we have a contradiction in our predicting valuation $\mathcal{V}_t()$ as we can check from Table 17. At this moment, we do not have experimental data to tell if our predicting valuation will be correct on very large concurrency.

11 Conclusion and Future Work

We propose a method which treats state-graphs as data-objects and allows the definition of state-graph manipulators as a user-friendly way to package the complex technology of computer-aided verification. The success of the method depends on whether many more state-graph manipulators can be developed and proven correct.

In the future, hopefully, our idea of state-graph manipulators can be used as a public framework for the verification of real-world projects. In such a framework, an easy-to-plug-in application program interface (API) should be defined and standardized so that researchers and developers can design, implement, and register their manipulators and everybody else can benefit from their achievements. Such a framework is very much like the Internet model and will certainly help in promoting the application of verification technology and cooperation of researchers throughout the world. However, certification methods for new manipulators will then become important issues.

Acknowledgments

The authors would like to thank the reviewers of the *IEEE Transactions on Computers*. Their comments and criticism have made this paper much better. This work is partially supported by the National Science Council of Taiwan, Republic of China, under grants NSC 87-2213-E-001-007, NSC 88-2213-E-001-002, NSC 89-2213-E-001-004, and NSC 90-2213-E-001-046. Parts of this manuscript were published in the *Proceedings of the 1998 IEEE International Symposium on High Asurance Systems Engineering* (HASE) [41], the *Proceedings of the 1998 International Conference on Real-Time Computing Systems and Applications* (RTCSA) [24], and the *Proceedings of the 1999 International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols/Protocol Specification, Testing, and Verification* (FORTE/PSTV) [25].

References

[1] M. Abadi and L. Lamport, "An Old-Fashioned Recipe for Real Time," *Proc. REX Workshop, Real-Time Theory in Practice,* pp. 1-27, June 1991.

[2] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi, "An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness," *Proc. IEEE Int'l Conf. Real-Time Systems Symp. (RTSS '92),* 1992.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill, "Modeling Checking for Real-Time Systems," *Proc. IEEE Logics in Computer Science,* 1990.

[4] R. Alur, T.A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems," *Proc. IEEE Real-Time Systems Symp.,* 1993.

[5] F. Balarin, "Approximate Reachability Analysis of Timed Automata," *Proc. Real-Time Systems Symp. (RTSS '96),* pp. 52-61, Dec. 1996.

[6] F. Balarin, K. Petty, A.L. Sangiovanni-Vincentelli, and P. Varaiya, "Formal Verification of the PATHO Real-Time Operating System," *Procs. 33rd Conf. Decision and Control (CDC '94),* Dec. 1994.

[7] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, Y. Wang, and C. Weise, "New Generation of UPPAAL," *Proc. Int'l Workshop Software Tools for Technology Transfer (STTT '98),* July 1998.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proc. Fifth Ann. Symp. Logic in Computer Science,* June 1990.

[9] A. Cimatti, F. Clarke, E. Giunchiglia, and M. Roveri, "NuSmv: A Reimplementation of Smv," *Proc. Int'l Workshop Software Tools for Technology Transfer (STTT '98),* July 1998.

[10] E. Clarke, O. Grumberg, M. Minea, and D. Peled, "State-Space Reduction Using Partial-Ordering Techniques," *Int'l J. Software Tools for Technology,* vol. 2, no. 3, pp. 279-287, 1999.

[11] C. Daws, A. Olivers, S. Tripakis, and S. Yovine, "The Tools KRONOS," *Hybrid System III,* pp. 208-219, 1996.

[12] C. Daws and S. Yovine, "Reducing the Number of Clock Variables of Timed Automata," *Proc. Real-Time Systems Symp.,* pp. 73-81, Dec. 1996.

[13] A.J. Dill, D.L. Drexler, A.J. Hu, and C.H. Yang, "Protocol Verification as a Hardware Design Aid," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors,* 1992.

[14] D. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *Proc. Int'l Conf. Computer-Aided Verification,* 1989.

[15] E.A. Emerson and A.P. Sistla, "Utilizing Symmetry When Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach," *ACM Trans. Programming Languages and Systems,* vol. 19, no. 4, pp. 617-638, July 1997.

[16] J.C. Fernandez, "Aldebaran: A Tool for Verification of Communicating Processes," Technical Report Spectre C14, LGI-IMAG Grenoble, 1989.

[17] J.-C. Fernandez and L. Mounier, "On the Fly Verification of Behavioral Equivalences and Preorders," *Proc. Third Int'l. Workshop Computer-Aided Verification,* July 1991.

[18] P. Godefroid and D. Pirottin, "Refining Dependencies Improves Partial-Order Verification Methods," *Proc. Fifth Int'l Conf. Computer Aided Verification,* pp. 438-449, June 1993.

[19] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-Time Systems," *Proc. IEEE Logics in Computer Science,* 1992.

[20] I.N. Herstein, *Topics in Algebra,* second ed. Lexington, Mass.: Xerox College Publishing, 1975.

[21] K. Hoffman and R. Kunze, *Linear Algebra,* second ed. Englewood Cliffs, N.J.: Prentice Hall, 1971.

[22] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[23] G.J. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[24] P.-A. Hsiung and F. Wang, "A State-Graph Manipulator Tools for Real-Time System Specification and Verification," *Proc. Int'l Conf. Real-Time Computing Systems and Applications (RTCSA '98),* 1998.
[25] P.-A. Hsiung and F. Wang, "User-Friendly Verification," *Proc. 1999 Int'l Conf. Formal Description Techniques for Distributed Systems and Comm. Protocols/Protocol Specification, Testing, and Verification (FORTE/PSTV)* J. Wu, S.T. Chanson, Q. Gao, eds., Oct. 1999.
[26] IEEE, *ANSI/IEEE 802.3, ISO/DIS 8802/3,* IEEE CS Press, 1985.
[27] C.N. Ip and D.L. Dill, "Better Verification through Symmetry," *Formal Methods in System Design,* vol. 9, nos. 1/2, 1996.
[28] L. Lamport, "A Fast Mutual Exclusion Algorithm," *ACM Trans. Computer Systems,* vol. 5, no. 1, pp. 1-11, Feb. 1987.
[29] F. Laroussine and K.G. Larsen, "Compositional Model Checking of Real Time Systems," *Proc. Sixth Int'l Conf. Concurrency Theory (CONCUR '95),* pp. 27-41, Aug. 1995.
[30] F. Larsen, K. Larsson, P. Petterson, and Y. Wang, "Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction," *Proc. Int'l Real-Time Systems Symp. (RTSS '97),* 1997.
[31] K.G. Larsen, P. Petterson, and W. Yi, "Compositional and Symbolic Model-Checking of Real-Time Systems," *Proc. 16th IEEE Real-Time Systems Symp.,* pp. 76-87, Dec. 1995.
[32] K.G. Larsen, B. Steffen, and C. Weise, "Fischer's Protocol Revisited: A Simple Proof Using Modal Constraints," *Hybrid System III,* pp. 604-615, 1996.
[33] R. Mateescu and H. Garavel, "XTL: A Meta-Language and Tool for Temporal Logic Model-Checking," *Procs. Int'l Workshop Software Tools for Technology Transfer (STTT '98),* July 1998.
[34] K.L. McMillan, *Symbolic Model Checking.* Kluwer Academic, 1993.
[35] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.
[36] H. Miller and S. Katz, "Saving Space by Fully Exploiting Invisible Transitions," *Proc. Conf. Computer Aided Verification (CAV '96),* 1996.
[37] X. Nicollin, J. Sifakis, and S. Yovine, "Compiling Real-Time Specifications into Extended Automata," *IEEE Trans. Software Eng.,* vol. 18, no. 9, pp. 794-804, Sept. 1992.
[38] K. Petty, "The PATHO Operating System and User's Guide," technical report, Univ. of California, Berkeley, 1993.
[39] A.S. Tanenbaum, *Computer Networks,* second ed. Englewood Cliffs, N.J.: Prentice Hall, 1989.
[40] S. Tripakis and S. Yovine, "Analysis of Timed Systems Based on Time-Abstracting Bisimulations," *Proc. Conf. Computer Aided Verification (CAV '96),* 1996.
[41] F. Wang and P.-A. Hsiung, "Automatic Verification on the Large," *Proc. Third IEEE Int'l Symp. High Assurance Systems Eng. (HASE),* Nov. 1998.

Farn Wang received the BS degree in electrical engineering from National Taiwan University (NTU) in 1982 and the MS degree in computer engineering from National Chiao-Tung University (NCTU) in 1984. From 1986 to 1987, he was a research assistant in the Telecommunication Laboratories, Ministry of Communications. In 1987, he entered the PhD program of Dartmouth College. In 1988, he joined the PhD program in computer science at the University of Texas at Austin (UT-Austin). In 1993, he received the PhD degree and moved back to IIS, Academia Sinica, Taiwan, as an assistant research fellow. In 1997, he was promoted to associate research fellow. Dr. Wang has been researching automated verification since he joined the Real-Time Aystem Lab at UT-Austin in 1988. He has been involved in the development of a series of experimental verification tools: ARTL, VERIFAST, SGM, and RED. He has also published papers in parametric analysis and synthesis of real-time systems. His most recent work can be found at: http://www.iis.sinica.edu.tw/~farn/. He has also taught courses at NCTU and NTU.

Pao-Ann Hsiung received the BS degree in mathematics and the PhD degree in electrical engineering from the National Taiwan University (NTU) in 1991 and 1996, respectively. From 1993 to 1996, he was a teaching assistant and system administrator in the Department of Mathematics, NTU. From 1996 to 2000, he was a postdoctoral researcher at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, Republic of China (ROC). Since February 2001, he has been an assistant professor in the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, ROC. Dr. Hsiung is a member of the IEEE, the IEEE Computer Society, and the ACM. He has been included in several professional listings such as *Marquis' Who's Who in the World* (17th millenium edition, 2000), *Outstanding People of the 20th Century* (second edition, 2000, Cambridge, England), *Who's Who in Formal Methods*, and ACM SIGDA's design automation professionals. Dr. Hsiung was on the program committee and served as either session organizer or chair of several international conferences such as PDPTA '99, RTC '99, DSVV '2000, and PDPTA 2000. He has published approximately 50 papers in international journals and conferences. His main research interests include hardware-software codesign and coverification, real-time system specification and verification, system-level design automation of multiprocessor systems, parallel architecture design and simulation, and object-oriented design techniques in system syntheses.