State Graph Manipulators (SGM)

A High-Level Specification and Verification Tool for Real-Time Systems*

Pao-Ann Hsiung and Farn Wang
Institute of Information Science, Academia Sinica, Taipei, TAIWAN.
E-mail: {eric,farn}@iis.sinica.edu.tw

SGM URL: **http://www.iis.sinica.edu.tw/~eric/sgm/**

Abstract

The current technology of verification engineering requires personnels to be well-trained in logics and automata theory, such that in order to attack the well-known *state-space explosion* problem they must carefully tune existing verification packages. Several researches have resulted in a large number of techniques for reducing the system state-space, such as symmetry-based reductions, partial-order semantics, bisimulation equivalences, and others. To let more people benefit from the technology of computer-aided verification, even people with little training in the related theories, a new verification tool called *State Graph Manipulators* (SGM) has been developed to package various sophisticated verification techniques as *manipulators* acting on high-level data-object representations of state-spaces called *state-graphs*. Experiments conducted using SGM show how the tool can increase verification efficiency and scalability. This article describes the features of SGM and how can a user may use SGM to experiment with state-space reductions and system verification.

1 Introduction

In the recent few years due to several breakthroughs, *model-checking* [4, 9], a formal and practical approach to the verification of concurrent systems, has become more and more popular. Model-checking has also been used to verify real-time systems. *Timed Automata* (TA) [6] has been widely used as the system model for verifying real-time properties specified in *Timed Computation Tree Logic* (TCTL) [10, 18]. Concurrency is generally modeled as the interleaving of computation sequences [25]. This causes state-space explosions [26] and the large sizes of the state-spaces become unmanageable, thus hindering verification. As a result, both the degree of concurrency and the complexity of systems verifiable are limited. Several techniques have been proposed in the literature for reducing the state-space size. Some of the techniques include symmetry-based reductions [10, 13, 14], partial-order reductions [28, 16, 25, 29, 26, 17], bisimulation equivalences [27], minimization techniques [5, 8], etc, which will be discussed in more details in Section 2.

Such reduction techniques usually require years of studying to master and are typically implemented with sophisticated data-structures for state-graphs. Moreover, since different verification tasks may need different ways of attacking the state-explosion problem, for each specific verification task the optimal combination of reduction techniques that reduces the state-spaces most and in the least time might be different. Even for a verification engineer properly knowledgeable of verification theory, to experiment with different combinations of the various reduction techniques may painfully take too much time and resources a project can sustain. This need for experimentation necessitates an environment where a designer can easily change the combination of reduction techniques applied to a given verification task. For this purpose, we have developed a new tool called the *State Graph Manipulators* (SGM) which packages various verification techniques as *manipulators* acting on high-level data-object state-space representations called *state-graphs*.

The rest of the article is organized as follows. Section 2 describes the verification framework of SGM and some potential state-space reduction manipulators. Section 3 describes the verification procedure language of SGM, its relationship to timed automata and TCTL. Section 4 describes in detail each of the six manipulators implemented in SGM. Section 5 shows how an SGM session can be conducted in both batch and interactive modes. The experimental results of Fischer's timed mutual exclusion protocol are also given. Section 6 describes the output format and shows how a user might trace the intermediate state-graphs for further investigation. Section 7 concludes this article.

*The work is partially supported by the National Science Council, Taiwan, ROC under grant NSC 87-2213-E-001-007.

2 SGM Verification Framework

We adopt the framework of model-checking in which a system is described by a set of concurrent timed automata [4] with dense-time semantics and a timing property is specified in *TCTL (Timed Computation Tree Logic)*[4, 18] extended from CTL. A system consists of a set of *timed automata*, labeled by unique indices. To compute global state-space representations (i.e. *state-graphs*), first, the set of timed automata are merged, two at a time. Second, during each merge iteration different sequences of reduction techniques may be applied to the intermediate state-spaces. Finally, the system is verified by model-checking the global state-graph against a given TCTL specification.

SGM allows a user to describe his/her system of timed automata, the property to be verified, and the actions to be performed on *state-graphs*, which are compact representations of state-spaces. SGM helps the designer in experimenting, online or offline, with the application of different sequences of reduction techniques to the intermediate state-graphs during the composition of a global state-graph. For a given verification task, this helps in finding a good sequence of *manipulators* (proven and implemented reduction techniques), which reduces the state-graphs and increases verification scalability.

The state-graphs manipulation approach adopted in SGM differs from conventional approaches including *global state-graph reduction*, such as global symmetry-based reduction [10, 14] and bisimulation equivalence-based minimization technique [5], and *on-the-fly reduction* such as on-the-fly symmetry reduction [20] and on-the-fly bisimulation equivalence [15]. Global state-graph reduction explicitly constructs the complete state-graph of the system and then tries to use different techniques to reduce the global state-graph size. This approach does not scale well because the global state-graph is often so large that its construction itself would blow up the available memory space. On-the-fly reduction approach applies reduction techniques to partially-composed global state-graphs. Not all reduction techniques are *simultaneously compatible*, that is they cannot be applied concurrently at the same time, they may be applied one after the other. Further, even the partial global state-graphs have large sizes, so on-the-fly techniques also fail at times. Hence, in order to avoid constructing a *large* global state-graph, we adopt the *compositional approach*, where the intermediate state-graphs obtained during each iteration of merging two component state-graphs are reduced as soon as they are constructed. The reduction obtained is thus accumulated over the iterations and thus allows the verification of systems with higher concurrency or more complex behavior.

2.1 Potential and Implemented State-Graph Manipulators

In the recent few years since 1992, *how to reduce the state-graph size* has been a hot research topic for formal verification researchers. This is because if formal verification is to be applied to industry its scalability must be improved, otherwise large complex systems can never be formally verified. This has resulted in a large number of techniques, some timed and some non-timed, in the literature and verification tools. Some of them are cited as follows:

- *symmetry-based reduction*: process indices, invariants, etc. are all permuted using some normalization scheme so that a normal form would represent the class of symmetrical modes/transitions [20, 10, 14, 30],
- *time-abstracting bisimulation*: equivalence classes are induced by abstracting away from the exact amount of time elapsed [27, 24],
- minimization techniques for minimal state-graph generation [5, 8],
- *quotient construction* by moving parts of a parallel system into the formula to be verified [22],
- *equivalence reduction of identifiers*: that is, two identifiers that are semantically equivalent are collapsed [22],
- *partial order reductions*: equivalent interleavings of concurrent events are grouped into traces and it suffices to explore just one representative interleaving from each trace for verifying temporal logic properties [26],
- *partial order reductions in symbolic state-space exploration*: an integration of partial order reduction techniques and symbolic state-space exploration [2],
- *partial order reductions for timed systems*: the problem of clock synchronizations causing partial order reductions to be less efficient in timed systems is overcome [7], and
- *reducing the number of clock variables*: active clocks are first detected and then clocks that are always equal are reduced into one [11].

Besides the above implementable manipulators, we have developed our own manipulators, namely *read-write reduction*, *clock-shielding*, *timed symmetry-reduction* [30], and *internal-transition bypassing*. All of these manipulators were theoretically proved for its soundness and implemented within our SGM tool. Other reduction techniques can be easily hooked into our tool as a new manipulator. Thus, researchers can take advantage of the state-graph manipulation framework provided by SGM to experiment with how his/her reduction technique works in collaboration with other existing techniques.

3 Verification Procedure Language

For the ease of system description, specification, and state-graph manipulations, State Graph Manipulators provides users with an interface language called *Verification Procedure Language* (VPL). As illustrated in Figure 1, the language consists of three parts: *system description*, *specification*, and *manipulation*. The first part is required for sessions, while the second and third parts are only for batch sessions. Sessions will be discussed in Section 5.

```
-----------------------------------------
-- (1) System Description Part
   automata A : i = 1..10, B : j = 2..5;
   clock x[1..total(A)], ...;
   register lock, ...;
   automaton A[i] {
     mode M1 { invariant ...;
       when ... }
     ........ }
   automaton B[j] {
     mode N1 { invariant ...;
       when ... }
     ........ }
-----------------------------------------
-- (2) TCTL Specification Part
   verify ......;
-----------------------------------------
-- (3) Manipulation Part
   manipulation
      graph G, .....;
      merge_graph(A[1], A[2], G);
      for..........;
-----------------------------------------
```

Figure 1: Verification Procedure Language

3.1 System description

A very flexible way of input is provided by SGM in the form of a *parametric* description. Both symmetric processes as well as asymmetric processes can be described using VPL. Each process is modeled by a timed automaton. A set of similar timed automata (TA) differ only in their indices. For describing a system of n TA, using VPL we need only to specify ONE TA with index i, instead of n TA. This is especially useful when n is large, for example, $n \geq 10$. Here, the index i is a parameter and it represents the index of a generic automaton, so that all mode predicates, transition triggering conditions, and transition assignments can be expressed using i.

We use a well-known benchmark example as our running example in this article. *Fischer's Mutual Exclusion Protocol* (FMEP) is a benchmark example that has been used for state-graph reduction technique evaluation in several literatures [21, 1, 23]. A generic timed automaton for this example is shown graphically in Figure 2 and the corresponding SGM input for an 11-process system obeying FMEP is given at the right of the figure.

VPL also allows easy change of the degree of concurrency by simply changing the value of parameter i in the first line (automata declaration) of a system description.

Theoretically, this part of the input corresponds to the timed automata model of a real-time system. A timed automaton (TA) is composed of various *modes* interconnected by *transitions*. Variables are distinguished into *clock* and *discrete*, where the former increments at a uniform rate and can be reset on a transition, while the latter change values only when assigned a new value on a transition. A TA remains in a particular mode as long as the values of all its variables, including clock and discrete, satisfy a *mode predicate*, which is a conjunction of clock constraints and boolean propositions. In the following, $\mathcal{N}$ and $\mathcal{R}$ represent the sets of non-negative integers and non-negative real numbers, respectively.

Definition 1 *:* **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η of C and D is defined as follows: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. $\|$

Let $B(C, D)$ be the set of all mode predicates over C and D. A TA may go from a mode to another mode, that is perform a transition, when the triggering condition (also specified as a mode predicate) is satisfied by the current valuation of the clock and discrete variables. On a transition from one mode to another, some actions are taken such as some clocks may be reset to zero and some discrete variables may be assigned new integer values.

Definition 2 *:* **Timed Automaton**
A *Timed Automaton* (TA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$ such that:

- M_i is a finite set of modes,
- $m_i^0 \in M_i$ is the initial mode,
- C_i is a set of clock variables,
- D_i is a set of discrete variables,
- $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode,
- $E_i \subseteq M_i \times M_i$ is a set of transitions,
- $\tau_i : E_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and
- $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such

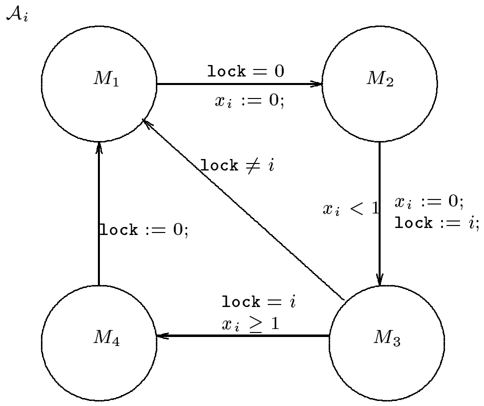

Figure 2: Fischer's Mutual Exclusion Protocol ($\mathcal{A}_i$)

```
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;}}
```

as resetting some clock variables and setting some discrete variables to specific integer values. ||

3.2 TCTL Specification

A TCTL formula has the following syntax.

$$\phi ::= \eta | \exists\Box\phi' | \exists\phi'\mathcal{U}_{\sim c}\phi'' | \neg\phi' | \phi' \vee \phi'' \tag{1}$$

Here, η is a state predicate in $B(C, D)$, $C = \cup_{i=1}^{n} C_i$, $D = \cup_{i=1}^{n} D_i$, n is the number of TA in a system, ϕ', ϕ'' are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$, and $c \in \mathcal{N}$. For further details on the semantics of a TCTL formula, users are advised to refer to [18].

In VPL, we support the full syntax of TCTL formulae 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 FMEP, 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)});
```

3.3 State-Graph Manipulation

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 Section 4.

Here, state-graphs are variables which have to be declared first. Then, a list of actions follows the declaration. A *simple* action 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 manipulator 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. The for-loops iterate on the indices of the declared state-graphs and the if-then-else statements compare state-graph sizes in terms of the number of modes and the number of transitions. Such comparisons allow users to dynamically change the choice of manipulator sequences. Some default merging and application of manipulators are also provided for users who prefer not to handle the details of each iteration. These include `sequential_merge()` (the next intermediate state-graph is always the result of merging the current intermediate state-graph and the next automaton in sequence), and all the manipulators without an input parameter such as `shield_clock()`, this means that the manipulators are to be applied to each merged intermediate state-graph in the sequence they are declared.

Continuing with the Fischer's mutual exclusion protocol example, state-graph manipulation can be specified as follows.

```
----------------------------------------
--  A user-defined sequential merge   --
----------------------------------------
manipulation
  graph g[1..total(A)-1];
  for(i:=1; i<total(A); 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]);
    normalize_region(g[i]); }
    model_check(g[total(A)-1]);
    print_graph(g[total(A)-1]);
```

Semantically, the above two methods of manipulation differ only in the order in which the manipulators `shield_clock()` and `normalize_region()` are applied.

4 Manipulators and their Implementation

The main factor that makes SGM user-friendly is the complete flexibility provided to a user in manipulating state-graphs through different permutations of manipulators. Any technique that acts on state-graphs can be implemented as a *manipulator*. We first define state-graphs and regions.

Definition 3 *:* **State-Graphs**
Given a concurrent real-time system S of n timed automata ($n > 0$) with process identifiers (also called indices) $\{1, 2, \ldots, n\}$, $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$, a state-graph $G = (q, Q, T)$ is a graph representation of a state-space, such that:

- $q \in V$, is a starting node of the state-graph,
- Q includes all the nodes in the state-graph, where each node represents a region (Definition 4) of the state-space and is semantically a collection of states satisfying some conditions in $B(\cup_{i=1}^n C_i, \cup_{i=1}^n D_i)$, and
- T includes all the arcs in the state-graph, where each arc represents a possible transition of the system from one node to another, that is, a TA in the system changes modes along the transition. $\|$

A *region* is formally defined as follows.

Definition 4 *:* **Regions**
A *region* is a collection of states such that: (1) they have the same *mode-vector label* (μ), which gives the mode names that each process timed automata is in, (2) they satisfy the same timing constraint in the form of a *zone* (ζ), which is a *Difference-Bound Matrix* (DBM) [3, 12], and (3) they have the same *set of literals* (L), representing the discrete variables' valuation. Thus, a region can be uniquely represented by a triple (μ, ζ, L). $\|$

A manipulator mainly checks the characteristics of each region in a state-graph, performs some comparisons between region characteristics or regions themselves, merges identical regions/transitions, removes redundant regions/transitions, etc. Currently, three types of manipulators are supported in SGM, namely, merge operations of state-graphs, reduction techniques, and model-checking algorithms. Six manipulators will be presented in this section and the experimental results on the effects of their combinations can be referenced from the examples provided by SGM. The six manipulators show how two state-graphs can be merged (**merge_graph()**), how a state-graph can be reduced by looking at the regions themselves (**normalize_region()**), or at the data-structures within a region, such as at clock variables (**shield_clock()**) and at discrete variables (**read_write()**), how a state-graph can be reduced by bypassing some transitions (**bit()**) or eliminating some transitions (**smtr()**).

4.1 State-Graph Merging

This manipulator called **merge_graph()** is not a reduction technique, but a necessary construction technique for composing two state-graphs into a single state-graph that represents the concurrent behaviors of the two components. In SGM, concurrency is modeled using interleaving semantics, that is, the execution of two or more concurrent transitions of a system is modeled as a set of interleaved computations. All possible interleavings of the concurrent transitions must be considered for a complete analysis. Although, reductions on interleavings are possible such as using the partial-order reduction technique, yet **merge_graph()** is implemented as a pure composition operator so that reduction effects can later be compared.

As defined earlier in Definition 4, a region has three components: *mode-vector*, *zone*, and *literal set*. The starting region of the newly composed state-graph is generated by merging the starting regions of the two component state-graphs. The two starting regions are merged as follows: their mode-vectors are merged into a new mode-vector, their zones are conjuncted to give the new zone, and the new literal set is the union of the two literal sets of the starting regions. Once the starting region is generated, all other regions are derived by considering each out-going transition of the component regions. In the course of generating new regions, all triggers on transitions and invariant conditions in regions must be satisfied, clocks must be reset, and discrete variables must be assigned new values by updating the literal set. SGM utilizes *Binary Decision Diagrams* (BDDs) for boolean operations such as conjunctions and disjunctions of zones and invariants. All newly generated regions are stored as nodes of a 2-3 tree for efficient search and comparison.

4.2 Timed Symmetry Reduction

This is an extension of the *non-timed* symmetry-based reduction technique proposed recently by Emerson and Sistla [14], such that *clocks* and their *readings* are also considered for symmetry reductions. We have independently implemented our *timed-symmetry reduction technique* using a normalization scheme in the form of a manipulator called **normalize_region()**. This manipulator is an indispensable one when the system is a set of *concurrent symmetric processes*, which differ only in their index labels.

Our implementation of the timed-symmetry reduction technique adopts a *sorting procedure* on the process indices by considering all the data-structures of a region: mode-vectors, zones, and literal sets. After sorting, each region can be represented by a normalized form, that is mode-vectors, zones, and literal sets are all normalized. If an existing region is itself in the normal form, then it is used for representing all regions with the same normal form. If no such normal form region exists, then a new normal region is created. In this way, the manipulator reduces the number of regions. Transitions will be duplicated when regions with the same normal form are merged into a single region, thus further reduction is possible by considering the redundancy in transitions (see subsection 4.6). Finally, after all regions and transitions are reduced, we have a unique representation for each region.

For example, suppose we are given a two-process state-graph having nodes with names $M_1.M_2$ and $M_2.M_1$, where M_1 and M_2 are mode names in the two process TA. The clock invariance of $M_1.M_2$ is $x_1 < 1 \wedge x_2 \geq 1$ and that of $M_2.M_1$ is $x_2 < 1 \wedge x_1 \geq 1$. When we sort the process indices $\{1, 2\}$ in an ascending order of the clock readings, these two nodes have the same normal form, i.e., $M_1.M_2$ and its clock invariance represent the normal form. Thus, the number of nodes can be reduced.

4.3 Clock Shielding

This manipulator was individually devised and implemented in SGM by the authors, but there is already published literature on *clock activity* [11], which is semantically the same as that implemented in our manipulator. Though their semantics is the same as ours, yet we adopt a different strategy in detecting *inactive clocks* (clocks that do not affect strategy in detecting *inactive clocks* (clocks that do not affect future evolution of a system are called inactive in [11]). We call them *shielded clocks* instead, because the clocks themselves are not inactive, they progress like all other clocks, but their readings are not useful currently so we instead *shield* them from external observation. The method presented in [11] was a fixpoint iteration strategy where the set of inactive clocks is obtained after a few iterations. Our method is by tracing all the paths in a state-graph starting from the region under consideration. Briefly, the manipulator works as follows: if a clock is not in the TCTL specification and if it is never read before it is reset, or will never ever be read again, then the clock can be shielded. As demonstrated by Daws and Yovine, this is a useful reduction technique. Our implementation and experimental results using SGM further show how this technique works in collaboration with other reduction techniques implemented as manipulators.

For example, as shown in Figure 3, in the two different computation runs starting from the two different nodes with the same mode-vector $M_4.M_1.M_2$, clocks x_1 and x_2 are not read in transition triggering conditions before being reset. Hence, the clocks can be shielded from external observation in $M_4.M_1.M_2$ without affecting the global behavior of the system.

The following Lemma from [31] shows the validity of this manipulator.

LEMMA 1 *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 clock x which is shielded in both ν and ν', the truth value of ψ is the same in ν and ν'.*

4.4 Read-Write Analysis

This is a new reduction technique that reduces the state-graph through an analysis of the discrete variable values as they are changed during transitions. Recall that each region is associated with a *literal set* (Definition 4). A literal in the set is basically of the form $(v \neq k)$, where v is a global discrete variable and k is any integer value. By analyzing all the possible values that a process automaton writes to, or does not write to a global discrete variable, this manipulator, called **read_write()**, computes the literal set of each region. The literal set represents the values that each global discrete variable does not possibly have in a region. Through such an analysis and literal sets formulation, transitions that have triggering conditions conflicting with the literal sets will never be triggerable and can thus be eliminated from further consideration. Lemma 2 explains the reduction technique implemented in **read_write()**.

Suppose we have an intermediate state-graph composed from state-graphs for processes with indices 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, with proof sketch, which governs the stability of global variable values, derivable from intermediate state-graphs, in concurrent read/write systems.

LEMMA 2 *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 $i, h \leq i < k$, ν_i goes to ν_{i+1} without making assignment to y on a transition from a process with 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: Since the values in $D_{H:y}$ are not going to be written to y by other processes with identifiers not in H, and 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 holds. ||

For example, given three processes with indices $\{1, 2, 3\}$ suppose process with index i only writes values 0 and i to a discrete variable *lock*. For the computation shown in Figure 4, only processes 1 and 2 are used to

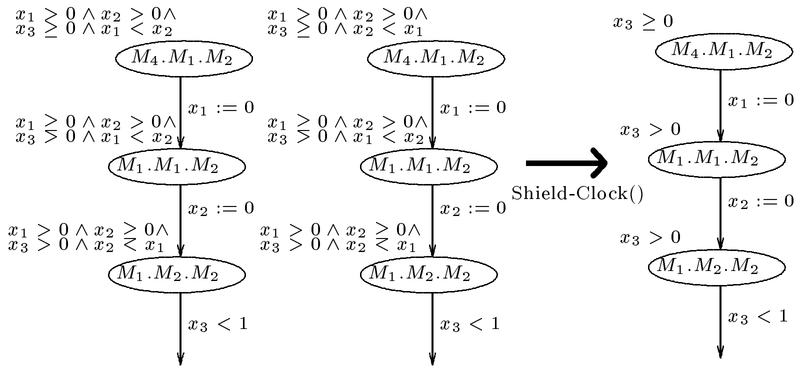

Figure 3: Shield Clock Manipulator Illustration Example

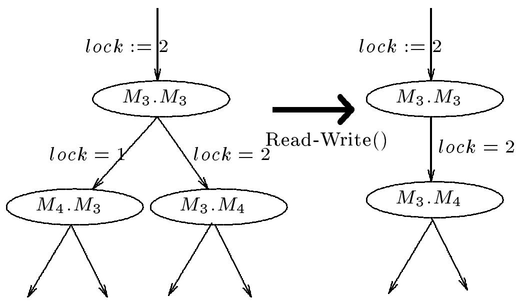

Figure 4: Read-Write Manipulator Illustration Example

construct the state-graph. We can be sure that in node $M_3.M_3$, the variable *lock* will definitely not have a value of 1, since 1 is only written by process 1 and in the incoming transition of $M_3.M_3$, *lock* is assigned a value of 2, hence *lock* can only take value 2 or that assigned by other processes. Thus, the transition with trigger $lock = 1$ will never happen.

4.5 Bypass Internal Transition

During compositional verification, an intermediate state-graph represents the concurrent behavior of k processes, $1 \leq k \leq n$. These k processes are called *internal*, while the rest are called *external*. Some behavior of internal processes in a state-graph such as the occurrence of a transition might not be observable by external processes. Such behaviors do not affect the global evolution of a system. *Bypass Internal Transition*(BIT) is a reduction manipulator that detects the occurrence of a transition in an intermediate state-graph, that is not observable by external processes. BIT detects internal behavior by checking (1) if the invariance condition of a node v implies that of a successor node v', where (v, v') is a transition under consideration, (2) if $\tau(v, v')$ is implied by the invariance condition of v, (3) if (v, v') only accesses variables not accessed by external processes, (4) if (v, v') does not reset any clocks, and (5) v' is not related to a given TCTL specification. Once such a transition is detected, BIT bypasses the transition by eliminating (v, v') and by adding new transitions between v and the successor nodes of v'.

BIT can achieve reduction of state-graph sizes because if a transition (v, v') is bypassed, it may happen that v' becomes unreachable from the initial node of a state-graph. Some redundancy in the newly added transitions may also occur. This is explained in the next manipulator.

4.6 Sibling Transition Multiplicity Reduction

Reduction techniques such as symmetry reduction, clock shielding, etc. often results in more than one transition between a pair of nodes in a state-graph. Out of the identical transitions between a pair of nodes, only one need be left in the state-graph. But, there is still some redundancy left in the transitions. *Sibling Transition Multiplicity Reduction* (STMR) is a reduction manipulator that detects such redundancy by checking if two transitions are corresponding transitions of two similar processes and their common source node is symmetric with respect to the two process indices. On detecting such transitions only one is left in the state-graph by STMR. This is because when a computation of a concurrent system reaches the source node of the corresponding transitions, the system cannot distinguish between the two processes.

Applying STMR after each of the other manipulators helps reduce state-graph sizes to a large extent because the reductions are accumulated through the sequence of manipulator applications.

For example, suppose we have two transitions t_1 and t_2 between nodes M_1 and M_2 in a state-graph such that they are corresponding transitions of two different similar processes with indices $\{1, 2\}$. Mode M_1 has a clock invariance $x_1 = x_2 < 1$ and literal set $\{lock \neq 1, lock \neq 2\}$. We note that the invariance condition and literal set of M_1 are all symmetric with respect to $\{1, 2\}$, thus when the system reaches M_1, taking transition t_1 or taking t_2 is the same for the system.

5 SGM Sessions

Currently, SGM (v1.2) supports three different modes of execution: a graphical user interface (GUI) mode, a *batch* mode, and an *interactive* mode. The batch mode needs an input file consisting of the three parts as described in Section 3, whereas the GUI and interactive modes only require the first two parts to be stored in an input file. Users can input commands one at a time to SGM in the interactive mode.

5.1 Graphical User Interface

SGM v1.2 now comes with a Graphical User Interface developed using the Qt X-windows programming library from the Troll Tech (http://www.troll.com). The Qt version used is 1.41, the library of which is required for running SGM. Since Qt allows free distribution of its UNIX version X-windows programming library, we have bundled the Qt library along with SGM v1.2 in the "sgm_1.2/lib/" directory.

The SGM GUI mainly illustrates a way in which sophisticated verification techniques (such as merging of state-space representations, reducing their sizes, ...) could be performed easily through the application of different modularized manipulators on state-graphs (high-level representation of state-spaces).

Each state-graph is represented by a small box in the window display area. The boxes can be double clicked with a mouse to check its more detailed information. Different manipulators can be applied on a state-graph box using the right button of a mouse. Two state-graphs can be merged by first selecting them and then using the Merge → Selected Graphs menu. Further, all the system graphs would be merged into a global one by the sequential merge command. During this merging, manipulators may be applied in each merge iteration but these manipulators have to be specified in the input file.

Graphs can be viewed, renamed, printed, saved, reset, freed, etc. Users can thus easily have an overall view of how different sequences of manipulators reduce state-graphs differently and to a different extent.

5.2 Batch Mode

The SGM input described for Fischer's mutual exclusion protocol example in the previous section must be stored in a file with a ".s" extension, e.g., `fischer.s`. The executable binary file of the SGM tool reads the input file and starts performing all the *actions* listed in the `manipulation` section of the input.

A part of a particular session of SGM is as follows. Sizes of intermediate state-graphs are all reported after the application of each manipulator.

```
Merging graphs A[1] & A[2] into _T[10] [2 automata]
.....70 modes, 160 transitions (Graph _T[10])
Shielding clocks in graph _T[10].
    Merging identical modes in graph _T[10].
    31 modes, 160 transitions (Graph _T[10])
    Reducing (multi)graph _T[10].
    31 modes, 70 transitions (Graph _T[10])
Symmetry-reducing regions in graph _T[10].
31 modes, 70 transitions (Graph _T[10])
    Removing irreachable modes in graph _T[10].
    16 modes, 70 transitions (Graph _T[10])
    Reducing (multi)graph _T[10].
    16 modes, 35 transitions (Graph _T[10]) ......
```

5.3 Interactive Mode

Currently, SGM possesses a text-based user-interface which interprets commands input by a user. The commands implemented in SGM can be classified as graph commands, shell commands, and miscellaneous commands. Graph commands help a user to merge, manipulate, check, and print state-graphs. Shell commands can allow a user to execute all external commands through a shell swap. Some external commands have been implemented into the interface such as `chdir`, so that a user may easily find the input file. Miscellaneous commands include the controlling of log file verbosity and help commands.

Using the commands, a user can change his/her sequence of actions (manipulator sequence) based on some preliminary experimental results obtained. An example interactive session of SGM is as given in Fig. 6.

SGM warns the user if he/she tries to perform some illegal actions such as model checking a non-global (intermediate) state-graph. Based on the higher state-graph size decrease rate and smaller construction time, the user opts for the {`shield_clock(), normalize_region()`} manipulator sequence for all future reductions.

5.4 Experimental Results

As shown in Table 1, after experimenting with different sequences of the manipulators currently implemented in SGM we notice that for the Fischer's mutual exclusion protocol (FMEP) example, although both the sequences {`read_write(), shield(), normalize()`} and {`read_write(), normalize(), shield()`} have the same final effect, that is, they reduce the intermediate state-graphs to the same size, yet 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 Figures 7 and 8. Further, comparing the time taken by the two sequences for state-graph reductions, we see from Figure 9 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. Several other examples were experimented using SGM, but due to page-limit we have only presented

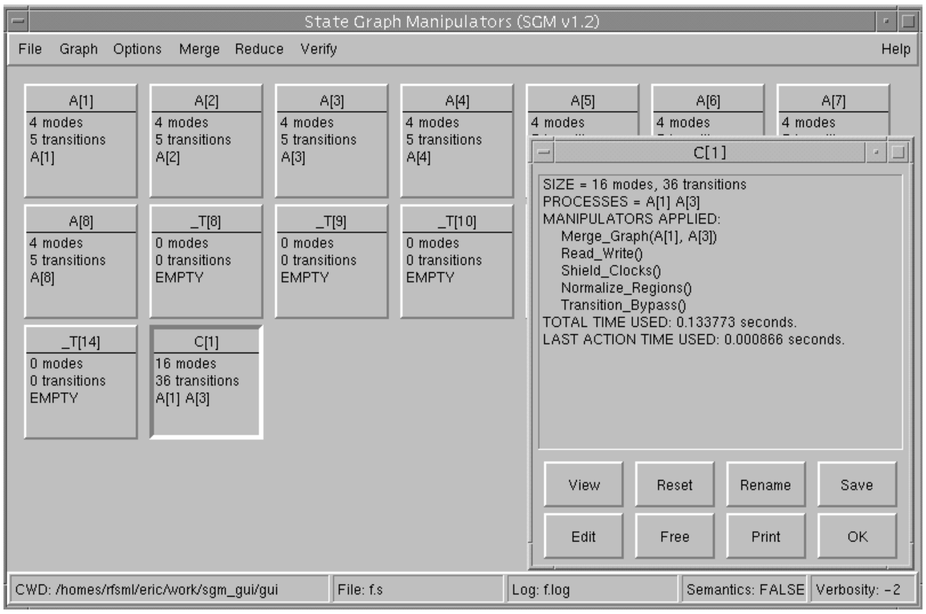

Figure 5: SGM Graphical User Interface

one. For another mutual exclusion example, the sequence does not remain the same [30]. See [30] for details.

6 State-Graph Tracing

In the current version of SGM, state-graph tracing can only be done through a log file specific to each session. By default, SGM does **not** log any operations. The default verbosity level of log files is -2, which implies no logging at all. There are several levels of verbosity for the output log file generated. For generating a log file of SGM operations, the verbosity level must be increased. When the SGM input file is named `application.s`, the log file will be named as `application.log`. Currently, any existing `application.log` file will be overwritten without any warning. This will be made more flexible in future versions.

For a minimum logging of only resulting state-graphs and not intermediate actions, users are advised to use a verbosity level of -1. Verbosity level of -1 outputs only the description of state-graphs constructed and their sizes. After each `merge_graph()`, `shield_clock()`, `normalize_region()`, `transition_bypass`, ... operation, the resulting state-graph is logged in the log file. Intermediate actions such as reducing a multigraph, merging identical modes, etc. are all not logged at the verbosity level of -1.

Depending on a user's needs, he or she may choose different verbosity levels for the output. All levels generate human-readable text files. Verbosity levels in SGM are as follows:

- **Level -2: (Default)** Except for the information on the standard console, no output of any sort is generated.
- **Level -1: (Recommended)** Only descriptions of state-graphs and their sizes are output. This level is useful for persons wanting to trace only the output graphs at the end of each action or manipulation. For further tracing the next level must be used. *Increase verbosity level by giving the command "`v ++`" in the interactive mode, or using the option "`-v`" in the batch mode.*
- **Level 0: (Tracing)** Each action (merge_graph, shield_clocks, ...) is logged with minimal description on the internal processing of state-graphs. This level is recommended for those who want to trace how the manipulators work on state-graphs.
- **Level 1: (Probing)**A slightly more detailed description of each action is logged. The level is good for an initial probe into SGM execution of actions.
- **Level 2: (Detailed)** A more complete description of each action is logged. The log file can be quite large. Use this level only if you cannot find what you want in Level 1.
- **Level 3: (Debugging)** A very complete description of all actions and internal processing is logged. This is useful for an expert desiring internal workings of SGM, but the log file can very large (sometimes easily exceeding your quota limitations, if any). Care must be taken when using this level of verbosity. Authors do not recommend using this level due to its large size.

```
sgm> load fischer.s
sgm> graph g[1], g[2]
sgm> merge_graph A[1] A[2] g[1]
    Merging graphs A[1] & A[2] into g[1] [2 automata] ...
    ..........122 modes, 298 transitions (Graph g[1])
    Elapse time : 0.261824 seconds.
sgm> merge_graph A[1] A[2] g[2]
    Merging graphs A[1] & A[2] into g[2] [2 automata] ...
    ..........122 modes, 298 transitions (Graph g[2])
    Elapse time : 0.261824 seconds.
sgm> shield_clock g[1]
    Shielding clocks in graph _T[11].
        Merging identical modes in graph _T[11].
        77 modes, 298 transitions (Graph _T[11])
        Reducing multigraph of graph _T[11].
        77 modes, 194 transitions (Graph _T[11])
    Elapse time : 0.084422 seconds.
sgm> normalize_region g[1]
    (Symmetry-Reducing) Normalizing regions in graph g[1].
    77 modes, 194 transitions (Graph g[1])
        Removing irreachable modes in graph g[1].
        47 modes, 194 transitions (Graph g[1])
        Reducing transitions in (multi)graph g[1].
        47 modes, 148 transitions (Graph g[1])
    Elapse time : 0.037288 seconds.
sgm> normalize_region(g[2]);
    (Symmetry-Reducing) Normalizing regions in graph g[2].
    122 modes, 298 transitions (Graph g[2])
        Removing irreachable modes in graph g[2].
        74 modes, 298 transitions (Graph g[2])
        Reducing transitions in (multi)graph g[2].
        74 modes, 222 transitions (Graph g[2])
    Elapse time : 0.064578 seconds.
sgm> shield_clock g[2]
    Shielding clocks in graph g[2].
        Merging identical modes in graph g[2].
        47 modes, 222 transitions (Graph g[2])
        Reducing multigraph of graph g[2].
        47 modes, 148 transitions (Graph g[2])
    Elapse time : 0.058603 seconds.
sgm> free_graph(g[2]); -- manipulator sequence used for constructing g[1]
                       -- is selected as the better one, g[2] is freed
sgm> merge_graph g[1] A[3] g[2]
     ...
sgm> shield_clock g[2]
     ...
sgm> normalize_region g[2]
     ...
sgm> quit
```

Figure 6: An Interactive Session of SGM

Table 1: **Fischer's Mutual Exclusion Protocol**

	#modes/#transitions						
n	2	3	4	5	6	7	8
read_write()	70/160	220/693	541/1990	1071/4469	1947/9104	3355/17393	5171/29001
shield()	31/70	100/303	231/824	444/1797	766/3462	1241/6164	1874/10012
normalize()	16/35	39/109	76/247	130/472	204/810	301/1290	424/1944
time (sec)	0.07	0.61	2.92	10.69	35.25	107.84	275.61
read_write()	70/160	220/693	541/1990	1071/4469	1947/9104	3355/17393	5171/29001
normalize()	36/80	182/617	465/1826	928/4074	1697/8291	2927/15707	4520/26166
shield()	16/35	39/109	76/247	130/472	204/810	301/1290	424/1944
time (sec)	0.08	0.89	4.66	17.87	61.19	212.40	486.67

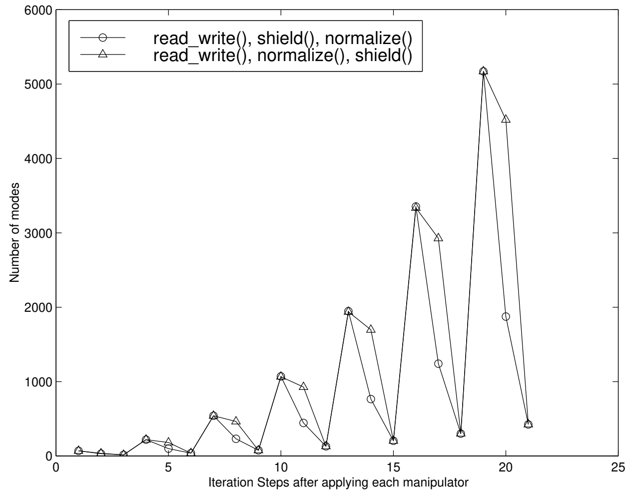

Figure 7: FMEP example (modes comparison)

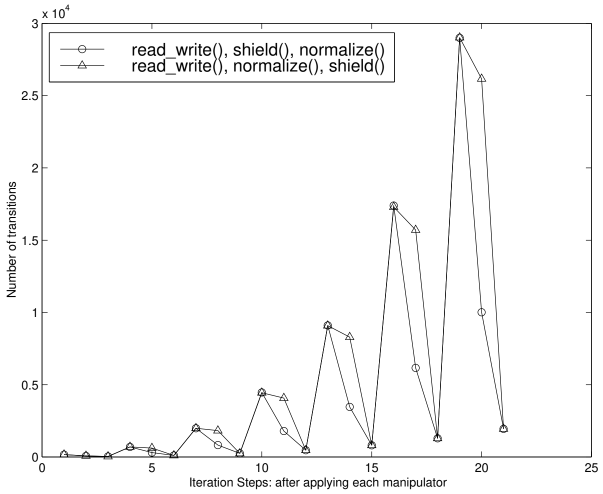

Figure 8: FMEP example (transitions comparison)

7 Conclusion

We have successfully developed a state-graph manipulation tool called *State Graph Manipulators* (SGM) for the specification and verification of real-time systems which are modeled as a set of timed automata and model-checked against TCTL specifications. SGM allows system designers to experiment with different sequences of manipulators that best fit a particular verification task at hand. At the same time, SGM allows verification researchers to experiment with how a new reduction technique developed by him/her would collaborate with other existing techniques. We expect that SGM would be a useful tool to both the verification expert as well as the verification layman (one who just wants to see how much his/her verification task could be best reduced).

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *REX Workshop, Real-Time Theory in Practice, Lecture Notes in Computer Science*, volume 600, pages 1–27, June 1991.

[2] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Proc. Intl. Conf. CAV'97*, 1997.

[3] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. IEEE Intl. Conf. Real-Time Systems Symposium (RTSS'92)*, 1992.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill. Modeling checking for real-time systems. In *Proc. IEEE Logics in Computer Science*, 1990.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proc. Intl. Conf. CONCUR'92, LNCS*, volume 630, pages 340–354, August 1992.

[6] R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

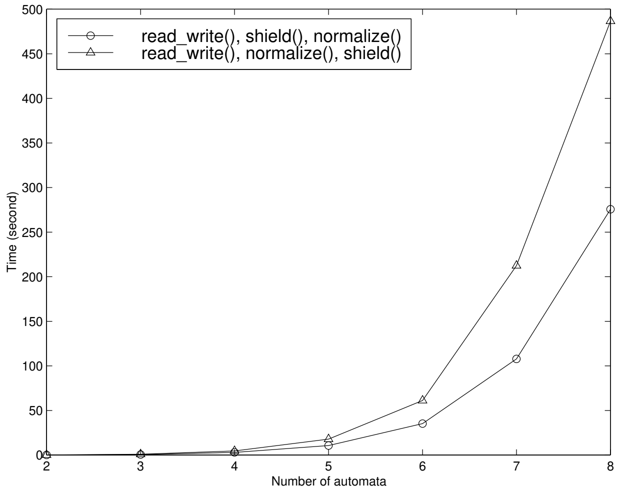

Figure 9: FMEP example (time comparison)

[7] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *Procs. CONCUR'98*, 1998.

[8] A. Bouajjani, J-C. Fernandez, N. Halbwachs, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, 1992.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual Symposium on Logic in Computer Science*, June 1990.

[10] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Lecture Notes in Computer Science*, volume 697, 1993.

[11] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. Real-Time Systems Symposium*, pages 73–81, December 1996.

[12] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. International Conference on Computer-Aided Verification, LNCS*, volume 407. Springer Verlag, 1989.

[13] E. A. Emerson and C. S. Jutla. Symmetry and model checking. In *Lecture Notes in Computer Science*, volume 697, 1993.

[14] E.A. Emerson and A.P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.

[15] J.-C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In *Proc. 3rd Intl. Workshop on Computer-Aided Verification, LNCS*, volume 575, July 1991.

[16] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proc. 6th Annual Symposium on Logic in Computer Science*, pages 406–415, July 1991.

[17] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110:305–326, 1994.

[18] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proc. IEEE Logics in Computer Science*, 1992.

[19] P.-A. Hsiung and F. Wang. A state-graph manipulator tool for real-time system specification and verification. In *Procs. of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, October 1998.

[20] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), 1996.

[21] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[22] K. G. Larsen, P. Petterson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proc. 16th IEEE Real-Time Systems Symposium*, pages 76–87, December 1995.

[23] K. G. Larsen, B. Steffen, and C. Weise. Fischer's protocol revisited: A simple proof using modal constraints. In *Hybrid System III, Lecture Notes in*

Computer Science, volume 1066, pages 604–615, 1996.

[24] K.G. Larsen and W. Yi. Time abstracted bisimulation: Implicit specifications and decidability. In *Proc. Intl. Conf. Mathematical Foundations of Programming Semantics, LNCS*, volume 802, April 1993.

[25] A. Mazurkiewicz. Basic notions of trace theory. In *Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency, Lecture Notes in Computer Science*, volume 354, pages 285–363, 1988.

[26] D.A. Peled. All from one, one for all: On model checking using representatives. In *Proc. of the 5th International Conference on Computer-Aided Verification, Lecture Notes in Computer Science*, volume 697, pages 409–423, 1993.

[27] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *CAV'96, Lecture Notes in Computer Science*, volume 1102, 1996.

[28] A. Valmari. A stubborn attack on state explosion. In *Proc. Workshop on Computer Aided Verification*, June 1990.

[29] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets, Lecture Notes in Computer Science*, volume 483, pages 491–515, 1991.

[30] F. Wang and P.-A. Hsiung. Automatic verification on the large. In *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, November 1998.

[31] F. Wang and P.-A. Hsiung. Iterative refinement and condensation for state-graph construction. In *Technical Report, TR-IIS-98-009, Insitute of Information Science, Academia Sinica, Taiwan*, July 1998.