A State Graph Manipulator Tool for Real-Time System Specification and Verification

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

Abstract

The current technology of verification engineering requires well-trained personnels in logics and automaton theory, who carefully tune their verification packages, to tame the well-known state-space explosion *problem. 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, etc. To let more people benefit from the technology of computer-aided verification even with little training in the related theories, a new tool called* State-Graph Manipulator *(SGM) was developed to package various sophisticated verification techniques as manipulators on state-graphs as high-level data-objects. An example user session of SGM is discussed and the results presented. Experiments conducted using SGM show how the tool, when used by a system designer, can increase verification efficiency and scalability.*

1. Introduction

In the recent few years due to several breakthroughs, *model-checking* [3, 8], an algorithmic approach to the formal verification of concurrent systems, has become more and more popular. Model-checking has also been used to verify real-time systems. *Timed automata* [5] has been widely used as the system model for verifying real-time properties specified in *Timed Computation Tree Logic* (TCTL) [9, 16]. Concurrency is generally modeled as the interleaving of computation sequences [22]. This causes state-space explosions [23] 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 [9, 12, 11], partial-order reductions [25, 14, 22, 26, 23, 15], bisimulation equivalences [24], minimization techniques [4, 7], 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 Manipulator* (SGM) which packages various verification techniques as manipulators on state-graphs as high-level data-objects.

The rest of the paper 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 and its relationship to timed automata and TCTL. Section 4 shows how an SGM session can be conducted in batch and interactive modes. The experimental results of Fischer's mutual exclusion protocol are also given. Section 5 gives the final conclusion.

2. Verification Framework

We adopt the framework of model-checking in which a system is described by a set of concurrent timed automata[3] with dense-time semantics and a timing property is specified in *TCTL (Timed Computation Tree Logic)*[3, 16] extended from CTL. A system consists of a set of *similar timed automata*, which differ at most in their indices (a distinguishing label). To compute the global state-space representations (i.e. *state-graphs*), first, the set of

0-8186-9209-X/98 $10.00

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 the 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 [9, 11] and bisimulation equivalence-based minimization technique [4], and *on-the-fly reduction* such as on-the-fly symmetry reduction [17] and on-the-fly bisimulation equivalence [13]. 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 [17, 9, 11, 27],
- *time-abstracting bisimulation*: equivalence classes are induced by abstracting away from the exact amount of time elapsed [24, 19],
- minimization techniques for minimal state-graph generation [4, 7],
- *quotient construction* by moving parts of a parallel system into the formula to be verified [20],
- *equivalence reduction of identifiers*: that is, two identifiers that are semantically equivalent are collapsed [20],
- *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 [23],
- *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 [6], and
- *reducing the number of clock variables*: active clocks are first detected and then clocks that are always equal are reduced into one [10].

Besides the above implementable manipulators, we have developed our own manipulators, namely *read-write reduction*, *clock-shielding*, and *timed symmetry-reduction* [27]. 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. In the following, we briefly describe how each currently implemented manipulator works in our tool. For further detailed information, the reader is advised to refer to the related work in [27].

- **Read-Write Reduction:** This manipulator is basically an analysis of the interactive behavior between the current partially-composed subsystem and that of the yet-to-be-composed partial system. The analysis mainly takes into account the values read and written by each transition from and to each discrete variable of the system. The read-write analysis deducts what literals, appearing in the system description and the specification, are invariably true in each mode. After associating each mode with an invariably true literal set, the manipulator eliminates transitions that are bound not to trigger in the global state-graph. Such transition eliminations result in some modes becoming irreachable, thus all irreachable modes are then detected and deleted.

- **Shield-Clock Reduction:** This manipulator tries to shield clocks from observation by external processes (that is the processes that have not yet been merged into the system under composition). The analysis works on clocks only and tries to detect all the clocks in a mode that are simply not read any more in the future or before a reset action on some future transition. Such clocks can in fact be ignored in that particular mode. A mode here means a collection of states that satisfy the same invariant condition. This shielding of clocks often results in some modes becoming identical or symmetrical. Further reduction is thus possible by merging identical modes and by normalizing symmetrical modes.

- **Normalize-Region (Symmetry) Reduction:** This manipulator takes advantage of the symmetry inherent in the system. Several symmetrical sets of states (called modes) can be represented by a single mode after associating a permutation label with each of the incoming transitions of the representative mode. This manipulator extends the recent work of Emerson and Sistla [11] to the timed systems. We use a normalization procedure on all components of a mode representation based on a sorting scheme that assigns a sequential order to each component. Here, by mode components we mean all the data structures required for representing a mode, for example, zone (a difference matrix representing timing constraints), literal set (all literals appearing in the system description and specification that are true in a mode), and mode-vector (the label of a mode in the form of a vector where each component is the mode of a system process automaton). After grouping symmetrical modes, the group is normalized into a single mode. This results in a multigraph, and thus some further reduction is possible whereby redundant transitions between two modes are deleted.

The above three manipulators have been successfully implemented in SGM and experimental results show significant reductions that help increase verification scalability.

3. Verification Procedure Language

For the ease of system description, specification, and state-graph manipulation, our State-Graph Manipulator tool provides the user with an interface language called *Verification Procedure Language*. As illustrated below, the language consists of three parts: *system description*, *specification*, and *manipulation*. All three parts are mandatory for a complete input to SGM.

```
------------------------------
-- (1) System Description Part
   total_automata 10;
   clock ...;
   register ...;
   automaton ........
------------------------------
-- (2) TCTL Specification Part
   verify ......;
------------------------------
-- (3) Manipulation Part
   manipulation
        graph .....;
        ..........
------------------------------
```

3.1. System description

A very flexible way of input is provided by SGM in the form of a *parametric* description. Since our system model consists of a set of similar timed automata that differ only in their indices, we need to specify only ONE timed automata that has index i, a parameter. The parameter i will represent the index of a generic automata, so that all mode predicates, transition triggering conditions, and transition assignments can be expressed using i.

Fischer's Mutual Exclusion Protocol (FMEP) is a benchmark example that has been used for state-graph reduction technique evaluation in several literatures [18, 1, 21]. A generic timed automaton for this example is shown graphically in Figure 1 and the corresponding SGM input for a 10 processes system obeying FMEP is given below.

```
total_automata 10;
clock x[1..total_automata];
register lock;
----------------------------------------
automaton A[i] {
initially M1 and lock = 0 and x[i] = 0;
  mode M1 {
```

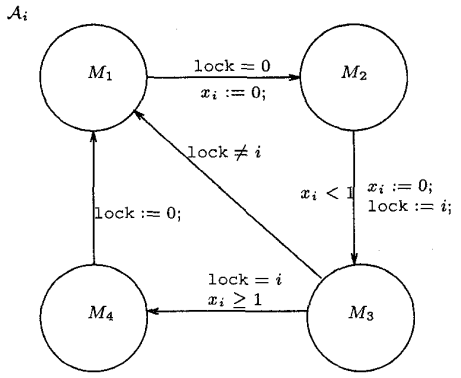

Figure 1. Fischer's Mutual Exclusion Protocol (*i*th automaton)

```
    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;}
}
```

The above description also allows easy change of the degree of concurrency by simply changing the `total_automata` value in the first line.

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 $(M, m_0, C, D, \chi, E, \tau, \rho)$ such that:

- M is a finite set of modes,
- $m_0 \in M$ is the initial mode,
- C is a set of clock variables,
- D is a set of discrete variables,
- $\chi : M \mapsto B(C, D)$ is an *invariance* function that labels each mode with a condition true in that mode,
- $E \subseteq M \times M$ is a set of transitions,
- $\tau : E \mapsto B(C, D)$ defines the transition triggering conditions, and
- $\rho : E \mapsto 2^{C \cup (D \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such 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.

$$\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi' \mathcal{U}_{\sim c} \phi'' \mid \neg\phi' \mid \phi' \vee \phi'' \qquad (1)$$

Here, η is a state predicate in $B(C, D)$, ϕ', ϕ'' are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$, and and $c \in \mathcal{N}$. Due to page-limit, we do not elaborate on the semantics of a TCTL formula, details can be found in [16].

The full syntax of TCTL specifications is supported by SGM. The specification begins with the keyword `verify`. The meanings of the keywords are obvious as they are just English words for the specific meanings that they represent such as `all_paths` means *for all paths in the state-graph starting from the initial mode.*

An example TCTL specification for a 3-automata Fischer's mutual exclusion protocol is given below.

```
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

The last part of an input for the SGM tool is the most important part as it allows the user flexibility in choosing *what* to do, *when* to do, and *how* to do state-graph manipulation actions. 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, two different ways of state-graph manipulation are as follows. The left one is a user-defined one and the right one is a default one.

```
-------------------------------------
-- A user-defined sequential merge  --
-------------------------------------
manipulation
  graph g[1..total_automata-1];
  for(i:=1; i<total_automata; 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_automata-1]);
  print_graph(g[total_automata-1]);
-------------------------------------
```

```
-------------------------------------
--  System-defined sequential merge --
-------------------------------------
manipulation
  sequential_merge();
  normalize_region();
  shield_clock();
  model_check();
  print_graph();
-------------------------------------
```

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

4. An SGM Session

SGM supports two different modes of execution: 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 interactive mode only requires 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.

4.1. 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 A[1] & A[2] into g[1]
......70 modes, 160 transitions
 Shielding clocks in g[1].
  Merging identical modes in g[1].
  31 modes, 160 transitions
  Reducing multigraph g[1].
  31 modes, 70 transitions
```

```
 Normalizing regions in g[1].
 31 modes, 70 transitions
  Removing irreachable modes in g[1].
  16 modes, 70 transitions
  Reducing multigraph g[1].
  16 modes, 35 transitions
......
```

4.2. Interactive Mode

A user-interface is provided by SGM, which interprets commands input by the user. A user can change his/her actions (manipulation sequence) based on the preliminary experimental results obtained (by `print_graph()`, `print_size()`,`print_time()` commands). An example interactive session of SGM is as follows.

```
sgm> load fischer.s;
     A[1], A[2], A[3] declared.
sgm> graph g[1], g[2];
sgm> merge_graph(A[1], A[2], g[1]);
     Merging A[1] & A[2] into g[1] ...
     ..........70 modes, 160 transitions
sgm> copy_graph(g[1], g[2]);
sgm> shield_clock(g[1]);
     Shielding clocks in graph g[1].
      Merging identical modes in g[1].
      31 modes, 160 transitions
      Reducing multigraph g[1].
      31 modes, 70 transitions
sgm> normalize_region(g[1]);
     Normalizing regions in graph g[1].
     31 modes, 70 transitions
      Removing irreachable modes in g[1].
      16 modes, 70 transitions
      Reducing multigraph g[1].
      16 modes, 35 transitions
sgm> normalize_region(g[2]);
     Normalizing regions in graph g[2].
     70 modes, 160 transitions
      Removing irreachable modes in g[2].
      36 modes, 160 transitions
      Reducing multigraph g[2].
      36 modes, 80 transitions
sgm> shield_clock(g[2]);
     Shielding clocks in graph g[2].
      Merging identical modes in g[2].
      16 modes, 80 transitions
      Reducing multigraph g[2].
      16 modes, 35 transitions
sgm> print_size(g[1]);
     Graph A[1]: 16 modes, 35 transitions.
sgm> print_size(g[2]);
     Graph A[1]: 16 modes, 35 transitions.
sgm> print_time(g[1]);
     0.07 second
sgm> print_time(g[2]);
     0.08 second
     -- manipulator sequence for g[1]
     -- is better, g[2] is freed
sgm> free_graph(g[2]);
sgm> model_check(g[1]);
     -- an illegal action
     WARNING: g[1] is not a
              global state-graph
sgm> merge_graph(g[1], A[3], g[2]);
     Merging g[1] & A[3] into g[2] ...
     .......86 modes, 191 transitions
sgm> shield_clock(g[2]);
     Shielding clocks in graph g[2].
      Merging identical modes in g[2].
      57 modes, 191 transitions
      Reducing multigraph g[2].
      57 modes, 130 transitions
sgm> normalize_region(g[2]);
     Normalizing regions in graph g[2].
     57 modes, 130 transitions
      Removing irreachable modes in g[2].
      23 modes, 130 transitions
      Reducing multigraph g[2].
      23 modes, 47 transitions
sgm> model_check(g[2]);
     Specification satisfied!
sgm> quit
```

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.

4.3. 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 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 2 and 3. Further, comparing the time taken by the two sequences for state-graph reductions, we see from Figure 4 it is also the first sequence that uses a shorter time.

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

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 one. For another mutual exclusion example, the sequence does not remain the same [27]. An automatic selection procedure for the best sequence of manipulators has been proposed by the authors in another related work [27]. Since this article is mainly an introduction to our tool, we have not described our manipulators in detail and the comparison of the reduction effects of our manipulators with other existing tools. Interested readers may refer to [28] for further information.

The first version of the SGM tool can currently be licensed for personal use. For further information on tool retrieval please refer to the following URL:

"http://www.iis.sinica.edu.tw/~eric/sgm/".

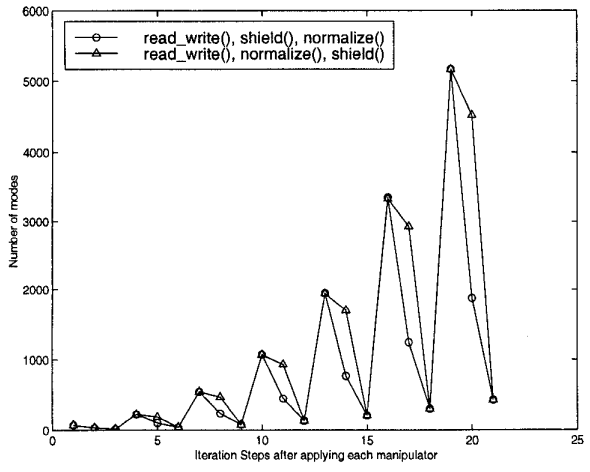

Figure 2. FMEP example (modes comparison)

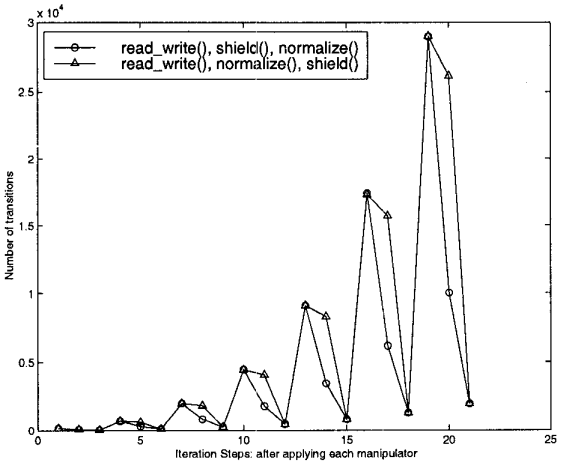

Figure 3. FMEP example (transitions comparison)

5. Conclusion

We have successfully developed a state-graph manipulation tool called State-Graph Manipulator (SGM) for the specification and verification of real-time systems which are modeled as 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).

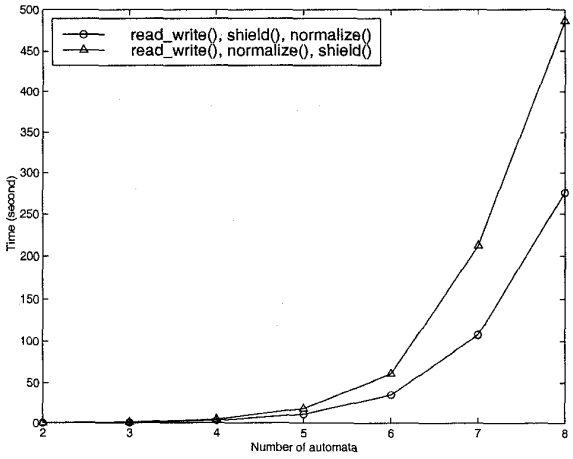

Figure 4. FMEP example (time comparison)

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. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Proc. Intl. Conf. CAV'97*, 1997.

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

[4] 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.

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

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

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

[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. In *Proc. 5th Annual Symposium on Logic in Computer Science*, June 1990.

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

[10] 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.

[11] E. Emerson and A. 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.

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

[13] 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.

[14] 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.

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

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

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

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

[19] K. 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.

[20] 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.

[21] 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.

[22] 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.

[23] D. 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.

[24] 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.

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

[26] 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.

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

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