User-Friendly Verification

PAO-ANN HSIUNG and FARN WANG
Institute of Information Science, Academia Sinica, Taipei, TAIWAN, R.O.C.
E-mail: {eric,farn}@iis.sinica.edu.tw
URL: http://www.iis.sinica.edu.tw/~eric/sgm/

Key words: timed automata, TCTL, model-checking, state-space explosion, verification tool, concurrent real-time systems

Abstract: Model checking often faces the problem of reducing the large exponential sizes of state-space representations. Several reduction techniques such as bisimulation equivalence, partial-order semantics, and symmetry-based reduction have been proposed, but existing tools do not completely allow a user the flexibility in manipulating state spaces. We propose a new *user-friendly* verification environment where a user has full control on what techniques to apply and in what sequences to apply them. We have implemented the environment in a tool called *State-Graph Manipulators* (SGM). SGM packages verification techniques into efficient, reusable, modular *manipulators*, that act on high-level state-space representations called *state-graphs*. Further, we are also proposing four new state-space reduction techniques, namely *variable shielding*, *read-write analysis*, *internal transitions bypassing*, and *sibling transition multiplicity reduction*. They are implemented into SGM and experiments have been conducted to show their usefulness.

1. INTRODUCTION

In the recent few years due to several breakthroughs, *model-checking* [2], a formal approach to the verification of concurrent systems, has become more and more popular. Model-checking has been used to verify real-time systems. *Timed automata* [4] has been widely used as the system model for verifying real-time properties specified in *Timed Computation Tree Logic* (TCTL) [11]. Concurrency is generally modeled as the interleaving of

computation sequences [15]. This causes state-space explosions and the large sizes of state-space representations become unmanageable, thus hindering verification. 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 representation size, including symmetry-based reductions [10], partial-order reductions [15], bisimulation equivalences [16], and minimization techniques [3].

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-space representations most and in the least time might be *different*. Even for an 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).

SGM adopts and presents a high-level view of all verification intricacies through a graphical user-interface (GUI). Various different sophisticated verification technologies are packed into efficient *manipulators* that act on high-level data-objects representing state-spaces. These data-objects are called *state-graphs* as defined later in Section 4. With a friendly GUI, users can choose manipulators, from the menu bar, during state-graph constructions and compare the effects of applying different *sequences of manipulators* to state-graphs.

Section 2 describes verification-related tools and their techniques. Section 3 describes the verification framework, the verification procedure language, and the three interfaces of SGM. Section 4 describes the seven manipulators implemented in SGM. Section 5 gives the experimental results for several application examples. Section 6 gives the final conclusion.

2. PREVIOUS 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, yet very few tools have been, are, or will be providing high-level perceptions. NuSMV [6], a new version of the well-known *Symbolic Model Verifier* (SMV), is one tool that will be

adopting an approach similar to ours, but they do not yet have any published results on the benefits of using high-level techniques, nor is there any description of concrete implementations in [6].

Murϕ Verification System [9] is a language-based verifier tool for finite-state verification of concurrent systems with symmetry-based reduction. **SPIN** [12] is an automated protocol validation system using PROMELA with refining dependencies for efficient partial-order verification. **KRONOS** [7] is a well-known verification tool with minimization algorithms, inactive clock reduction [8], and clock equality [8] reduction techniques implemented. **UPPAAL** [5] 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. **SMV** [14] is a well-known tool for model-checking finite-state systems against CTL (*Computation Tree Logic*) specifications. A new version called NuSMV has implemented automatic variable ordering and cache configuration [6].

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 (implemented in different tools) to a verification task, then the user will have to spend a 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. Pain staking 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 problems. SGM is proposed with this motivation in mind.

3. SGM: A USER-FRIENDLY VERIFICATION FRAMEWORK

SGM provides a user-friendly verification environment through a *flexible* input language and three user interfaces: *graphical*, *interactive*, and *batch*.

We adopt the framework of model-checking in which a system is described by a set of concurrent timed automata [2] with dense-time semantics and a timing property is specified in TCTL (*Timed Computation Tree Logic*) [2,11] extended from CTL. To compute the compact global state-space representations (i.e. *state-graphs*), first, the set of timed automata is merged, two at a time. Second, during each merge iteration different sequences of reduction techniques may be applied to the intermediate state-graphs. 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* (defined in Section 4). 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* (proved and implemented verification techniques), which reduces the state-graph sizes and increases verification scalability.

We have developed seven manipulators in SGM, namely *state-graph merging*, *timed symmetry-reduction* [17], *clock-shielding*, *variable-shielding*, *read-write analysis*, *internal transition bypassing*, and *sibling transition multiplicity reduction*. Of the above, the first three were proposed by other authors and ourselves, the other four are new reduction techniques. These seven techniques will be discussed in Section 4. Other reduction techniques can be easily hooked into our tool as a new manipulator. Thus, researchers can take advantage of the framework to experiment with how one's reduction technique works in collaboration with other existing techniques.

3.1 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. Specification corresponds to a TCTL formula. Manipulation corresponds to a list of actions on state-graphs.

We will use *Fischer's Mutual Exclusion Protocol* (FMEP) as a running example in this paper. FMEP is a benchmark example that has been used for state-graph reduction technique evaluation in several literatures [13]. A generic timed automaton for the ith process and the corresponding SGM input for a 11-process system obeying FMEP are shown in Fig. 1. Fischer's protocol says that given a system of n concurrent processes, the time taken by each process for checking (or reading) whether a *lock* variable is zero must be lesser than that for assigning (or writing) its index value to the *lock* variable. We must prove that such a system will never allow more than one process into the critical section (mode M_4). VPL allows easy change of a system's degree of concurrency by only changing 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 variables of the former type increment at a uniform rate and can be reset on a transition, while variables of the latter type change values only when assigned a new value on a transition. A TA may remain 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, N and R represent the sets of non-negative integers and non-negative real numbers, respectively.

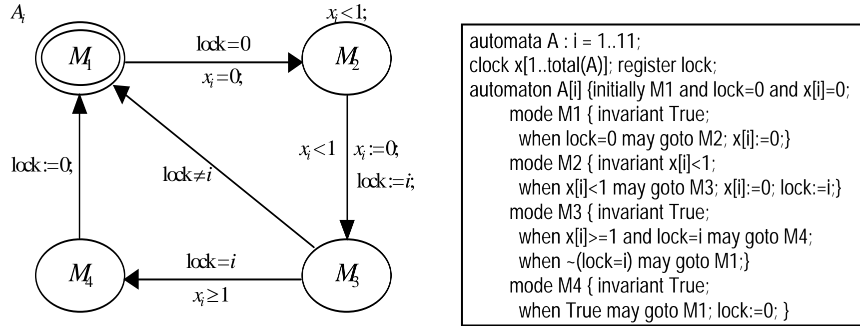

Figure 1. Fischer's Mutual Exclusion Protocol

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 over C and D is defined as follows: $\eta := \text{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where x, $y \in C$, $\sim\, \in \{\leq, <, =, \geq, >\}$, $c \in \mathrm{N}$, $d \in D$, 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, that is perform a transition, when the triggering condition (specified as a mode predicate) is satisfied by the current valuation of the clock and discrete variables. On a transition, 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 $\mathrm{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, χ_i: $M_i \rightarrow B(C_i, D_i)$ 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, τ_i: $E_i \rightarrow B(C_i, D_i)$ defines the transition triggering conditions, and ρ_i: $E_i \rightarrow 2^{C_i \cup (D_i \times \mathrm{N})}$ 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.

As far as temporal property specification is concerned, SGM uses TCTL. A TCTL formula has the following syntax. $\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi' U_{\sim c}\phi'' \mid \neg\phi' \mid \phi' \vee \phi''$. Here, η is a mode predicate in $B(C, D)$, ϕ', ϕ'' are TCTL formulae, $\sim\, \in \{<, \leq, =, \geq, >\}$, and $c \in \mathrm{N}$. Due to page-limit, we do not elaborate on the semantics of a TCTL formula. Details can be found in [11].

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)});
```

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-limit.

State-graph manipulation for FMEP example 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]);
    normalize_region(g[i]); }
  model_check(g[total(A)-1]);       -- model-check global graph
```

3.2 User Interfaces

Besides a human-readable input language, user-friendly verification is achieved by SGM through three user interfaces including graphical, interactive, and batch. A user has to first create a text input file using VPL.

In the graphical mode, as shown in Fig. 2 after a user loads 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. A 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.

Figure 2. SGM Graphical User Interface

In the interactive mode, SGM uses a command shell to accept commands from the user one at a time. This mode is useful when windows cannot be displayed, for example, over a terminal connection. The commands can be classified as graph commands, shell commands, and miscellaneous commands. Using the commands, a user can change his/her sequence of manipulators based on initial experimental results.

In the batch mode, SGM needs an input file that contains all the three parts of VPL, including the manipulation part, so that SGM can run all the listed actions in a batch.

4. MANIPULATORS

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) $\{1,2,\ldots,n\}$, $A_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$, 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.

Definition: 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) [1], 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 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. Three types of manipulators are supported: merging of state-graphs, reduction techniques, and model-checking algorithms.

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.

4.2 Timed Symmetry Reduction

This is an extension of the *non-timed* symmetry-based reduction technique proposed recently by Emerson and Sistla [10], 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*.

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 4.7). Finally, after all regions and transitions are reduced, we have a unique representation for each region.

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* [8], 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 future evolution of a system are called inactive in [8]). 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 [8] was a fix-point 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.

4.4 Variable Shielding

Applying the same concept of clock shielding to the discrete variables, we have constructed a new manipulator called **shield_variable()**. A discrete variable is said to be *shieldable in a mode* if it does not occur in a given TCTL specification and along all possible out-going paths starting from that mode either its value will never be read again or it will be assigned a new value before being read. This indicates the value of a shieldable discrete variable in a mode has no effect on the global behavior of a system.

By shielding some variables in the modes of a state-graph, some modes become identical in all respects and can thus be identified into a single mode.

Resulting multiple transitions can be reduced using *Sibling Transition Multiplicity Reduction* technique as described later in this section.

4.5 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 1 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.

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

4.6 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. The BIT manipulator will be illustrated in the Ring Network example of Section 5.

4.7 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.

5. APPLICATION EXAMPLES

Several academic as well as industrial examples were verified using SGM. The three application examples presented here include: (1) Fischer's

timed mutual exclusion protocol (FMEP) [13], (2) Rules of graphical user interface for a simple calculator, and (3) ring network token passing.

5.1 Fischer's Timed Mutual Exclusion Protocol

This example was used for illustration throughout the article. The timed automaton for the *i*th process was shown in Fig. 1 and system description using VPL were described in subsection 3.1. As shown in Table 1, after experimenting with five different sequences of the manipulators in SGM we notice that for Fischer's mutual exclusion protocol [13], although both the sequences (D) and (E) 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. Further, comparing the time taken by the two sequences for state-graph reductions, we see 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.

Table 1. Fischer's Mutual Exclusion Protocol (11 processes)

	#Modes #Transitions Construction Time (sec)									
n	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
A	70	1239	28593	O/M	O/M	O/M	O/M	O/M	O/M	O/M
	160	4013	120850							
	0.06	3	-							
B	31	182	1024	5804	O/M	O/M	O/M	O/M	O/M	O/M
	70	578	4149	28451						
	0.07	1	13	-						
C	36	224	1270	O/M	O/M	O/M	O/M	O/M	O/M	O/M
	80	743	6357							
	0.08	3.3	186							
D	16	39	76	130	204	301	424	576	760	219
	35	109	247	472	810	1290	1944	2822	3937	1063
	0.07	0.6	3	10.7	35	108	276	-	-	-
E	16	39	76	130	204	301	424	576	760	219
	35	109	247	472	810	1290	1944	3400	4786	1174
	0.08	0.9	4.7	18	61	212	487	-	-	-

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

5.2 Rules of graphical user interface for a 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 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 action part was mapped to a set of transition assignment statements. The set of automata obtained from the rules is shown in Fig. 3.

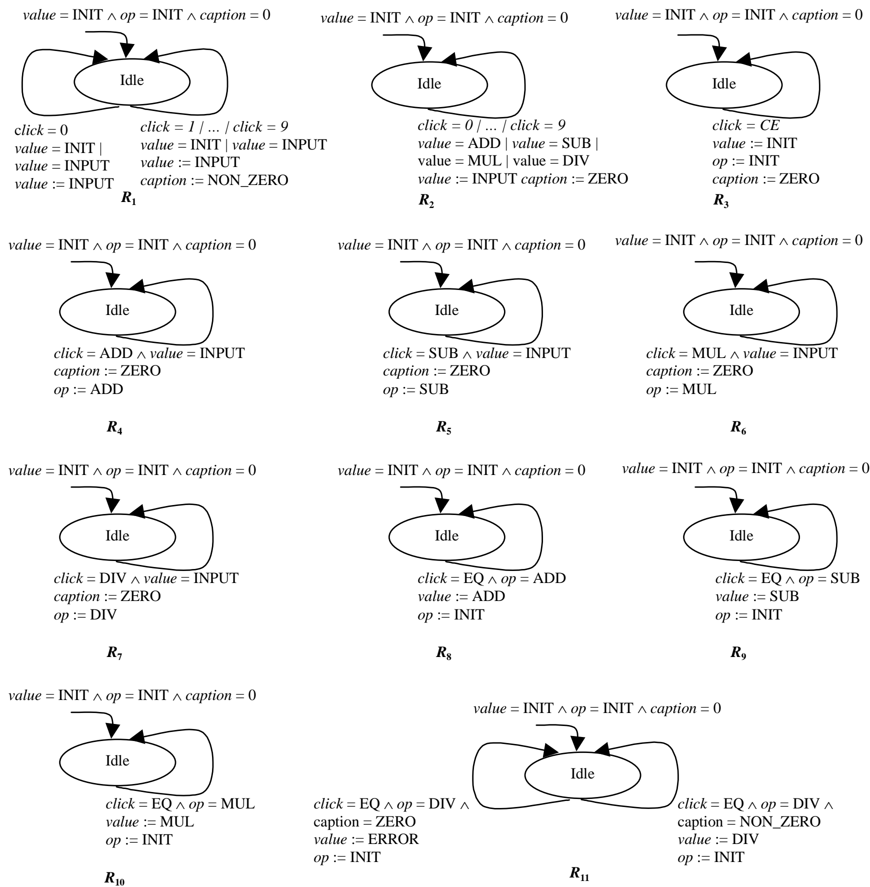

Figure 3. Calculator GUI Example

As shown in Table 2, on applying the manipulators **read_write()** and **shield_variables()** after each **merge()**, we found a significant reduction in state-graph sizes. The reduction was as much as 73.6% for transitions and 24.7% for modes.

Table 2. Rules of Calculator GUI

	#Modes #Transitions										Time (sec)
n	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	
A	11	11	22	33	44	55	69	83	97	90	12.9
	660	671	1364	2079	2816	3575	4554	5561	6596	1229	
B	11	11	21	31	41	51	61	71	81	90	27.5
	650	561	1091	1641	2211	2801	2711	2421	1931	1229	
C	10	10	10	19	28	37	55	64	73	83	32.6
	590	510	519	1005	1509	2031	2444	2182	1740	1133	

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

5.3 Ring Network

This example illustrates how BIT reduces state-graphs. In Fig. 11, we have a ring network consisting of three processes $\{p_0, p_1, p_2\}$ and three tokens $\{t_0, t_1, t_2\}$. Each process enters a critical section only when its token is true. Initially, only process p_0 has its token t_0 set as true, thus p_0 enters the critical section. After p_0 leaves the critical section, it sets t_0 to false and t_1 to true, thus allowing process p_1 to enter the critical section. Likewise, processes p_1 and p_2 behave in a similar manner. We observe that token t_0 is accessed only by p_0 and p_2 and not by p_1. Similarly, token t_1 is accessed only by p_1 and p_0, and t_2 by only p_2 and p_1. When we construct an intermediate state-space representation for processes p_0 and p_2, token t_0 becomes internal. The action of reading t_1 along transition (*Idle*, *Critical-Section*) of process p_0 becomes an internal action in the intermediate state-space representation of p_0 and p_2. This action is not observable by external process p_1.

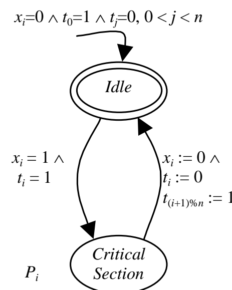

Figure 4. Ring Network

From Table 6, we can observe that the BIT manipulator in sequence (B) reduces the intermediate state-graph sizes and achieves a greater scalability such that 7 processes in a ring network can be verified as compared to that of only 6 processes without applying BIT. Sequence (C) can achieve a greater reduction but at the cost of time.

Table 3. Ring Network

	#Modes / #Transitions						Total time
n	**2**	**3**	**4**	**5**	**6**	**7**	**(seconds)**
A	32/64	221/663	1364/5450	7842/38460	O/M	-	N/A
B	14/28	63/202	302/1440	1508/9600	7567/61220	18/18	125
C	12/26	51/1840	244/1355	1254/9433	6574/63974	18/18	1270

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

6. CONCLUSION

We have successfully developed a user-friendly verification environment as a state-graph manipulation tool called *State-Graph Manipulators* (SGM) for the specification and verification of real-time systems which are modeled as timed automata and model-checked against TCTL specifications. We have also proposed four new state-graph reduction techniques: Variable-Hiding, Read-Write, BIT, and STMR. 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 tuned for efficiency and scalability).

Other existing reduction techniques will be gradually implemented into SGM. Further, new reduction techniques and their interaction with existing techniques will be investigated using SGM

Acknowledgments

We acknowledge the help provided by Mr. Ruey-Cheng Chen and Mr. Chao-Chi Chang of National Taiwan University for the implementation of the SGM Graphical User Interface. We also acknowledge Prof. Yue-Sun Kuo for providing us the Calculator GUI rules example.

References

[1] 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*, 1992.

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

[3] 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.

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

[5] J. Bengtsson, F. Larsen, K.Larsson, P. Petterson, Y. Wang, and C. Weise, “New generation of UPPAAL,” In *Procs of the Intl Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

[6] A. Cimatti, F. Clarke, E Giunchiglia, and M. Roveri, “NuSmv: a reimplementation of smv,” In *Procs of the Intl Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

[7] C. Daws, A. Olivers, S. Tripakis, and S. Yovine, “The tools KRONOS,” In *Hybrid System III*,” Lecture Notes in Computer Science, volume 1066, pages 208-219, 1996.

[8] 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.

[9] A. J. Dill, D. L.Drexler, A. J. Hu, and C. H. Yang, “Protocol verification as a hardware design aid,” In *Procs of the IEEE Intl Conf on Computer Design: VLSI in Computers and Processors*, 1992.

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

[11] 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.

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

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

[14] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publisher, 1993.

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

[16] 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.

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