Automatic Verification on the Large[*]

Farn Wang and Pao-Ann Hsiung
Institute of Information Science, Academia Sinica, Taipei, Taiwan 115, Republic of China
+886-2-27883799 ext. 1717; FAX +886-2-27824814; farn@iis.sinica.edu.tw

Abstract

An automatic verification method from a high-level resource-management standpoint is presented. Various manipulators can be incorporated in the method to construct, refine, reduce, and model-check state space representation. Proper combinations of manipulators can then be picked strategically by users or computers for less resource (time and space) consumption. An algorithm based on group theory to pick a manipulator combination is presented. Verification sessions are conducted to illustrate our idea.

1. Introduction

The general trend of engineering is to package complex technology with simple and friendly interfaces so that more users can be benefited. Since the famous Pentium-bug, people have been anticipating wide acceptance of the technology of computer-aided verification. Indeed, with today's powerful hardware and recently reported verification theory breakthroughs[2, 5], it seems that industrial application of verification theory is becoming more and more real. But most verification packages today are developed based on profound, complex theories that takes years of graduate study to master. Thus inevitably, only projects with big budgets can afford the advantage of computer-aided verification. This work aims at devising a packaging scheme for verification technologies so that users illiterate of verification technology can still benefit from it.

Here we give a brief description of our method which works on concurrent systems. For a system with m concurrent processes, we assume that we are given their m behavior descriptions, called *state-graphs* stored in an array, $G[1], \ldots, G[m]$ respectively. In traditional approach, a verification procedure will start by constructing the Cartesian product of $G[1], \ldots, G[m]$, as in table 1(a) with eight processes, to verify the given concurrent system. The state-

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

(a)

```
verify(G, 8, φ) /* Assume we have reducers
C1, ..., C6. */
state_graph *G;
int 8; /* number of processes */
CTL φ;
{
    G := G[1] × G[2];
    G := C1(C4(G));
    G := G × G[3];
    G := C2(C5(C6(G)));
    G := G × G[4];
    ......
    G := G × G[8]
    G := C1(C2(G));
    check G against φ;
}
```

(b)

Table 1. Verification sessions in comparison

graphs of local processes are stored in array G. The standard technology now is symbolic manipulation[5, 15, 4] which can be used in both Cartesian product calculation and model-checking. To cope with resource consumption requirement from different verification tasks, very often ingenious strategies have to be devised to keep space and CPU-time under control. To this end, users have to be knowledgeable of the theory of computer-aided verification and traverse through the final product state-graph G.

On the contrary, our method treats state-graphs as high-

[*]The work is partially supported by NSC, Taiwan, ROC under grant NSC 87-2213-E-001-007.

level data-objects and defines and implements many theoretically proven manipulators to merge, reduce, check them. From users' standpoint, the goal is to construct a verification procedure which composes a representation for global state space from all the state-graphs with manageable space and CPU-time consumption. There are three types of manipulators in our method: $\times$ for binary merge, **check**() for model-checking, and *reducers* for reducing the sizes of state-graphs. With the many manipulators in our method, users can easily test different combinations of manipulators. In table 1(b), we have another example verification session using our method. Users calculate the binary products of state-graphs and intermittently reduce them with different reducer combinations as the users see fit. It is up to users to pick their combinations of manipulators to reduce resource consumptions (memory and CPU time) to fulfill given verification tasks. Just like a mechanic can buy various components from shelves to build her/his dream car, users of our method can also enjoy the technology of CAV without deep understanding of the component technologies and construct the verification procedure better suits them with the manipulations supported in our method.

At this moment, we have successfully developed the several theoretically sound reducers. For each different verification task, there can be a different reducer combination for it which may cost the least memory space and CPU time. One research issue in our method is how to choose a good reducer combination for a given verification task when the number of reducers is large. In section 5, we also present an algorithm based on group theory[13, 14] to pick an efficient reducer combination for a given verification task.

Section 2 briefly describes our problems. Section 3 presents the general framework of our method. Section 4 briefly describes the manipulators we have implemented so far. Section 5 shows an algorithm which uses group theory to find a local optimal combination of manipulators to counter with state-explosion problem. Section 6 illustrates our idea with experiments. Section 7 is the conclusion.

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

2. Problem presentation

We adopt the framework of model-checking for concurrent systems. That is, each verification task is composed of a pair: a system description in dense-time automata and a specification formulus in CTL (Computation Tree Logic)[2, 9, 10]. Due to page-limit, we shall only give brief descriptions of both. Interested readers should be directed to [2, 9, 10].

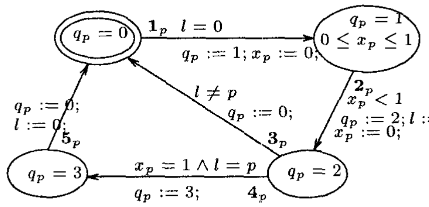

Figure 1. Fischer's mutual exclusion protocol

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

In our formalism, all processes run the same program. Thus the system behavior is totally described by a generic process' program. We shall describe such a program as a *process timed mode-transtion system (PTMTS)*. Here we have an example.

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

A CTL formulus has the following syntax.

$$\phi ::= x \sim c \mid y = c \mid \exists\Box\phi' \mid \exists\phi'\mathcal{U}\phi'' \mid \neg\phi' \mid \phi' \vee \phi''$$

Here x is a timer name, $\sim \in \{\leq, <, =, >, \geq\}$, c is a natural constant, and y is a variable name. $\exists\Box\phi'$ means there exists a computation, from the current state, along which ϕ' is always true. $\exists\phi'\mathcal{U}\phi''$ means there exists a computation, from the current state, along which ϕ' is true until ϕ'' becomes true. Traditional shorthands like $\exists\Diamond$, $\forall\Box$, $\forall\Diamond$, $\forall\mathcal{U}$, $\wedge$, and $\rightarrow$, can all be defined.

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

```
verify($G, m, \phi$)  /* Assume we have reducers $C_1, \ldots, C_6$. */
state_graph $*G$;
int $m$;
CTL $\phi$;
{
  state_graph $H, H_1, H_2, H_3$;                                   (1)
  int $i$;                                                           (2)

  $H := G[1]$;                                                       (3)
  for $i := 2$ to $m$, do {                                          (4)
    $H := H \times G[i]$;                                            (5)
    $H_1 := C_1(C_2(H))$;                                            (6)
    $H_2 := C_3(C_4(C_5(C_6(H))))$;                                  (7)
    if Size($H_1$) < Size($H_2$) then $H := H_1$;                    (8)
    else if Size($H_1$) > Size($H_2$) then $H := H_2$;               (9)
    else if Time($H_1$) < Time($H_2$) $H := H_1$;                    (10)
    else $H := H_2$;                                                 (11)
  }
  return check($H, \phi$);                                           (12)
}
```

Table 2. An example verification procedure

3. General framework of verification

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

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

Since our method allows reasoning on state-graphs as whole objects instead of searching for paths in them, users can test different combination of reducers to achieve their verification tasks from a resource management point of view without deep knowledge of the technology and theory inside the manipulators.

4. Manipulators

In this section, we shall first briefly describe some reducers which we have proven correct and implemented. Potential reduction algorithms in the literature can be found in [19, 17, 3, 6, 16, 18, 1, 7, 11].

4.1. Implemented reducers

The three implemented reducers have all been proven in theory to be correct. We refer readers to [23] for details on their operation and correctness. In this paper, we shall only give a sketchy view on how they work.

4.1.1. ReadWrite() : Variable value stability under concurrent read/write

In concurrent systems, each process behaves in a distinct way by writing specific values to and reading specific values from global variables. For example, in Fischer's mutual exclusion protocol in figure 1, each process writes only two values, 0 and its identifier, to global variable l. Also when a process reads from l, it only cares if the value is zero or its identifier. Whether the present value of l is the identifier of a particular peer process is of no concern to the reading process. This kind of behavior is actually very common in protocol systems.

Suppose we are given a set H of process identifiers. 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. In [23], there is a lemma which says that *if we know that for an interval, y starts with value not in $D_{H:y}$ and no process with identifier in H is going to write values in $D_{H:y}$ to y, then we can conclude that at any instant in the interval y does not contain any value in $D_{H:y}$*. Such a lemma helps us to get rid of a lot of states which according to the lemma, will never exist. We have implemented a reducer **ReadWrite()** based on this lemma in [23]. In fact, it is observed from experiments that this reducer has wide applicability to many verification tasks. Thus in our implementation, it is implemented inside the merge operator $\times$.

4.1.2. Shield() : shielded timer elimination

In [23], a timer x at a state is defined *shielded* if

- x is not used in the specification and
- from that state on, no inequalities of x ever change Boolean values until x is reset again.

It is then proven as a lemma that two states are equivalent with respect to a model-checking problem instance iff the two states are identical except for the readings of those shielded timers. We have implemented a reducer **Shield**() based on this lemma.

4.1.3. Symmetry() : Region and transition reduction by symmetry

In a real-time concurrent system with identical processes, utilizing symmetry is a must in efficient verification. The idea in our method is very similar to the one presented in [12] except we extend the symmetry to cover timing inequalities of timers in zones. Basically, we use process identifiers in state information. For example, we may have two states $\{l = 0; q_1 = 2; x_1 = 0.5; q_2 = 1; x_2 = 0.3\}$ and $\{l = 0; q_1 = 1; x_1 = 0.3; q_2 = 2; x_2 = 0.5\}$ for Fischer's protocol (fig 1) with concurrency = 2. By permuting the identifiers of 1 and 2, we find that the two states become the same. Thus in our verification data-structure, we shall label permutations on transitions. According to [12] and [23], the technique is valuable in verifying systems with identical processes. However, this reduction leaves the state-graph as a multi-graph.

5. Automatic reducer combination for efficient verification

At this moment, we have three reducers in addition to the merge operator $\times$. As mentioned in section 4 that from the literature, there are many reduction algorithms with potential to become new reducers. Different reducers may have different impacts on different verification tasks. Moreover, when the number of reducers is large, sometimes it may become difficult for users to pick a good reducer combination to accomplish their verification tasks. In this section, we shall develop an algorithm based on group theory[13, 14] to automatically pick a "locally optimal" combination for a given task.

Suppose we have n reducers: $\mathbf{C}_1$, $\mathbf{C}_2$, ..., and $\mathbf{C}_n$. We shall simplify the verification procedure construction problem to the verification procedure template in table 3. Thus

```
verify(G, m, φ) /* Assume we have reduce operators
C_1, ..., C_n. */
state_graph *G;
int m;
CTL φ;
{
  state_graph H;                                          (1)
  int i;                                                  (2)

  H := G[1];                                              (3)
  for i := 2 to m, do                                     (4)
    H := C_{i_1}(C_{i_2}(... (C_{i_k}(H × G[i]))...));    (5)
  return check(H, φ);                                     (6)
}
```

Table 3. A verification procedure template

the goal for the construction of efficient verification procedure is restricted to finding a good sequence $i_1, \ldots, i_k$ from the integer interval $[1, n]$ such that the verification procedure costs less space and time.

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

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

5.1. Structure of reducer sequence groups

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

By group theory[13, 14], it is known that every permutation can be constructed as a sequential composition of binary permutations. This further implies that for any two reducer sequences γ, γ' composed of the same set of reducers, there is a finite sequence $\theta_1 \ldots \theta_h$ of binary permutations such that $\gamma\theta_1\theta_2 \ldots \theta_h = \gamma'$. For convenience, we adopt left-associativity to interpret the ordering of permutation operations. Thus all reducer sequences composed of the same set of reducers form a connected graph.

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

The following lemma depicts the structures of reducer sequence groups. Remember that at the end of section 1, we defined that for any sequence F, $[F]$ is the set of elements in F. Due to page-limit, the proof is omitted.

LEMMA 1 : *Given a set $\{\mathbf{C}_1, \ldots, \mathbf{C}_n\}$ of reducers, for any two reducer sequences γ, γ' constructed from the set, there is a sequence $\lambda = \gamma_1\gamma_2 \ldots \gamma_k$ of reducer sequences such that*

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

With the operations of binary permutations and appending, we know that we can draw a undirected *reducer sequence graph (CSG)* for a given set of reducers. The nodes in a CSG are reducer sequences while the arcs are determined by if the two nodes can be related by a binary permutation or an appending operation. In fig 2, we have a CSG for three reducers. Lemma 1 says that such a graph is connected.

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

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

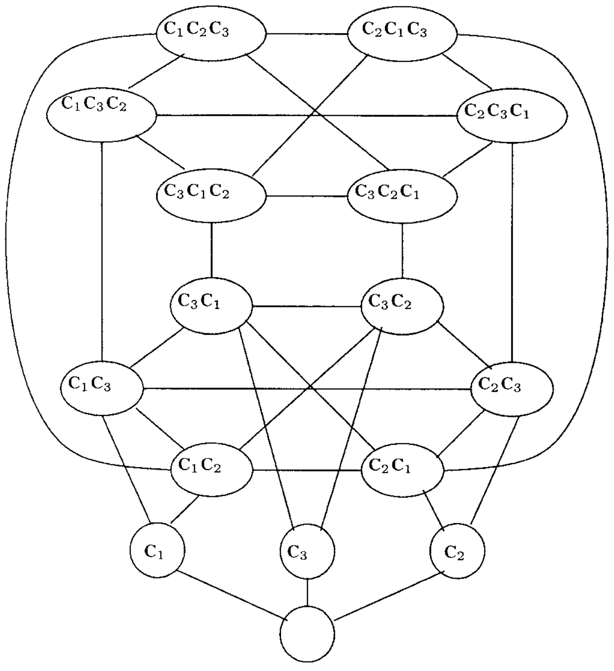

Figure 2. CSG for three reducers

5.2. Algorithm to pick a locally most efficient sequence

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

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

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

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

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

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

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

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

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

Table 4 is our algorithm in picking a reducer sequence for efficient verification. The termination of its execution is guaranteed because the CSG is finite and the search stops when γ is no greater than its neighbors with respect to $\mathcal{V}_s()$ and $\mathcal{V}_t()$. In section 6, we shall have experiment data to justify our algorithm.

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

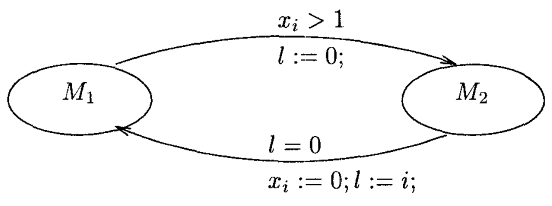

Figure 3. A small mutual exclusion protocol

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

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

6. Experiment

We have conducted experiments on our methods of verification on the large. At this moment, we have implemented the merge operator $\times$, three reducers: **readwrite()**, **shield()**, and **symmetry()**. We test our method on Fischer's timed mutual exclusion protocol in figure 1 and the simple mutual exclusion protocol in figure 3. In the following, we have collected data on the effects of different reducer sequences.

In table 5 and 6, we have shown the graph sizes in mode counts, transition counts, and their sums for small mutual exclusion protocol (figure 3) and the Fischer's protocol (figure 1) respectively.

Now we compare the data with our predicting valuation $\mathcal{V}_s()$. In table 5, for reducer sequence A, B, C, D, and E, $\mathcal{V}_s(A) = \frac{42 \times 949}{222^2} \approx 0.809$, $\mathcal{V}_s(B) = \frac{33 \times 1032}{200^2} \approx 0.851$, $\mathcal{V}_s(C) \approx 0.810$, $\mathcal{V}_s(D) \approx 0.789$, and $\mathcal{V}_s(E) \approx 0.789$. As we can see that the valuation indeed gives a good prediction of how fast the multiplication rate grows. It is obseved that by permutating the **shield**() and the **symmetry**() reducer, there is no difference made with respect to the space complexity prediction. Thus we can further calculate $\mathcal{V}_t(D) = \frac{0.02 \times 0.32}{0.10^2} = 0.64$ and $\mathcal{V}_t(E) = 0.6$. And we see from table 5, reducer sequence E indeed looks like to have the smallest space and time complexity.

In table 6, for reducer sequence A, B, C, D, and E, $\mathcal{V}_s(A) = \frac{230 \times 149443}{5252^2} \approx 1.246$, $\mathcal{V}_s(B) = \frac{101 \times 5173}{760^2} \approx 0.905$, $\mathcal{V}_s(C) \approx 0.946$, $\mathcal{V}_s(D) \approx 0.752$, and $\mathcal{V}_s(E) \approx 0.752$. Again the valuation indeed gives a good prediction of how fast the multiplication rate grows. Since reducer sequence D and E tie in space complexity predica-

```
good_sequence($G, m$) /* Assume we have reduce operators $\mathbf{C}_1, \ldots, \mathbf{C}_n$. */
state_graph $*G$;
int $m$;
{
    state_graph $H1, H2, H3$;                                                                      (1)

    Randomly pick a reducer sequence $\gamma$ from $\mathbf{C}_1, \ldots, \mathbf{C}_n$.                                  (2)
    Repeat forever {                                                                                (3)
        $B_\gamma := \{\gamma(j, j') \mid 1 \leq j < j' \leq |\gamma|\} \cup \{\gamma' \mid \exists \mathbf{C} \in \{\mathbf{C}_1, \ldots, \mathbf{C}_n\} - [\gamma'](\gamma = \gamma'\mathbf{C})\} \cup \{\gamma\mathbf{C} \mid \mathbf{C} \in \{\mathbf{C}_1, \ldots, \mathbf{C}_n\} - [\gamma]\}$;   (4)
        Pick $\gamma' \in B_\gamma$ such that for all $\gamma'' \in B_\gamma$, $\mathcal{V}_s(\gamma') \leq \mathcal{V}_s(\gamma'')$;                              (5)
        If $\mathcal{V}_s(\gamma) < \mathcal{V}_s(\gamma')$, return $\gamma$; else if $\mathcal{V}_s(\gamma) > \mathcal{V}_s(\gamma')$, $\gamma := \gamma'$; else if $\mathcal{V}_t(\gamma) \leq \mathcal{V}_t(\gamma')$, return $\gamma$; else $\gamma := \gamma'$;   (6)
    }
}
```

Table 4. Our reducer sequence picking algorithm

	#Modes/#Transitions/#Modes+#Transitions Construction Time (seconds)						
n	2	3	4	5	6	7	8
A	14/28/42 0.01	57/165/222 0.11	203/746/949 0.66	725/3040/3765 3.75	2646/12003/14649 20.81	10469/49411/59880 126.95	O/M
B	11/22/33 0.02	51/149/200 0.11	215/817/1032 1.00	850/3759/4609 8.07	3311/15864/19175 74.91	13430/66567/79997 1025.10	O/M
C	8/14/22 0.02	15/35/50 0.09	24/68/92 0.30	35/116/151 0.73	46/171/217 1.36	60/251/311 2.58	76/357/433 4.64
D	6/11/17 0.02	12/29/41 0.10	20/58/78 0.32	30/100/130 0.84	42/157/199 1.88	56/231/287 3.82	72/324/396 7.24
E	6/11/17 0.02	12/29/41 0.10	20/58/78 0.30	30/100/130 0.78	42/157/199 1.74	56/231/287 3.43	72/324/396 6.61

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

Table 5. Small Mutual Exclusion Protocol Example

tion, we further calculate $\mathcal{V}_t(D) = \frac{0.07 \times 2.92}{0.61^2} \approx 0.549$ and $\mathcal{V}_t(E) = 0.47$. It seems that we have a contradiction in our predicting valuation $\mathcal{V}_t()$ as we can check from table 6. At this moment, we do not have experiment data to tell if our predicting valuation will be correct on very large concurrency.

7. Conclusion

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

References

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Partial-Order Reduction in Symbolic State-Space Exploration," Intl. Conf. CAV'97.

[2] R. Alur, C. Courcoubetis, D.L. Dill. Model Checking for Real-Time Systems, IEEE LICS, 1990.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi, "Minimization of Timed Transition Systems," Intl. Conf. CONCUR'92, August 1992, Lecture Notes in Computer Science, Vol. 630, pp. 340-354.

[4] R. Alur, T.A. Henzinger, P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. in Proceedings of 1993 IEEE Real-Time System Symposium.

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

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

Table 6. Fischer's Mutual Exclusion Protocol Example

[6] A. Bouajjani, J-C. Fernandez, N. Halbwachs, and P. Raymond, "Minimal State Graph Generation," Science of Computer Programming, Vol. 18, No. 3, 1992, pp. 247-269.

[7] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial Order Reductions for Timed Systems," to appear in Procs. CONCUR'98.

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

[9] E. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, Proceedings of Workshop on Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[10] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems 8(2), 1986, pp. 244-263.

[11] C. Daws and S. Yovine, "Reducing the Number of Clock Variables of Timed Automata," in Procs. Real-Time Systems Symposium, Dec. 1996, pp. 73-81.

[12] E.A. Emerson, A.P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM TOPLAS, Vol. **19**, Nr. 4, July 1997, pp. 617-638.

[13] I.N. Herstein. Topics in Algebra, 2nd Edition, 1975, Xerox College Publishing, Lexington, Massachusetts.

[14] K. Hoffman, R. Kunze. Linear Algebra, 2nd edition, 1971, Prentice Hall, Inc., Englewood Cliffs, New Jersey.

[15] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic Model Checking for Real-Time Systems, IEEE LICS 1992.

[16] K.G. Larsen, P. Pettersson, and W. Yi, "Compositional and Symbolic Model-Checking of Real-Time Systems," in Procs. of the 16th IEEE Real-Time Systems Symposium, Dec. 1995.

[17] K.G. Larsen and W. Yi, "Time Abstracted Bisimulation: Implicit Specifications and Decidability," Intl. Conf. Mathematical Foundations of Programming Semantics, April 1993, Lecture Notes in Computer Science 802.

[18] D. Peled, "All from One, One for All – on Model Checking Using Representatives," Intl. Conf. CAV'93, June 1993, Lecture Notes in Computer Science, Vol. 697, pp. 409-243.

[19] S. Tripakis and S. Yovine, "Analysis of Timed Systems Based on Time-Abstracting Bisimulations," Intl. Conf. CAV'96, July 1996, Lecture Notes in Computer Science, Vol. 1102.

[20] F. Wang, A.K. Mok, E.A. Emerson. Real-Time Distributed System Specification and Verification in APTL. ACM TOSEM, Vol. 2, No. 4, Octobor 1993, pp. 346-378.

[21] F. Wang. Timing Behavior Analysis for Real-Time Systems. IEEE LICS 1995.

[22] F. Wang. Reachability Analysis at Procedure Level through Timing Coincidence. in Proceedings of the 6th CONCUR, Philadelphia, USA, August 1995, LNCS 962, Springer-Verlag.

[23] F. Wang, P.-A. Hsiung. Iterative Refinement and Condensation for State-Graph Construction. Technical Report TR-IIS-98-009, Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC.

[24] F. Wang, C.T. Lo. Procedure-Level Verification of Real-Time Concurrent Systems. to appear in Proceedings of the 3rd FME, Oxford, Britain, March 1996; in LNCS, Springer-Verlag.