Modere: The Model-checking Engine of Rebeca

Mohammad Mahdi Jaghoori
IPM School of Computer Science & Sharif Univ of Tech, Tehran, Iran
jaghoori@mehr.sharif.edu

Ali Movaghar
Sharif Univ of Tech and IPM School of Computer Science, Tehran, Iran
movaghar@sharif.edu

Marjan Sirjani
University of Tehran and IPM School of Computer Science, Tehran, Iran
msirjani@ut.ac.ir

ABSTRACT

Rebeca is an actor-based language with formal semantics that can be used in modeling concurrent and distributed software and protocols. Automatic verification of these systems in the design stage helps develop error free systems. In this paper, we describe the model checking tool developed for verification of Rebeca models. This tool uses partial order reduction technique for reducing the size of the state space generated for a given model. Using this tool for model checking Rebeca yields much better results than the previous attempts for model checking Rebeca.

Categories and Subject Descriptors

D2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Verification

Keywords

Automated Verification tool, Partial Order Reduction, Rebeca, Actor model.

1. INTRODUCTION

Model checking is the automatic and algorithmic way for verification of system correctness. State space explosion is a major obstacle in performing model checking in practice. The problem arises when we try to explore all the reachable states of a system to see whether a specific property is met or not. To overcome this problem, numerous methods have been proposed that avoid the construction of the complete state graph [3]. Among these methods are symbolic verification [13], partial order reduction [6], modular (parameterized) model checking [11], and symmetry reduction [4, 12]. These techniques are sometimes combined to achieve even more compression in the representation of the system under analysis [5].

SAC'06, April, 23-27, 2006, Dijon, France.

In an asynchronous concurrent system, *processes* are responsible for the behavior of the system. At each state, the interleaving of the enabled actions from different processes determines the possible future states. It is, however, not always necessary to consider all the possible interleaved sequences of these actions. Partial order reduction method [6] suggests that at each state, the execution of some of the enabled actions can be postponed to a future state, while not affecting the satisfiability of the correctness property. Therefore, with avoiding the full interleaving of the enabled actions, some states are excluded from the exhaustive state exploration in model checking.

Rebeca [15, 16] (reactive objects language) is an actor-based language, which can be used at a high level of abstraction for modeling object-based concurrent systems. The asynchronous message-passing paradigm in Rebeca allows for efficient modeling of loosely coupled distributed systems. Being familiar with the Java-like syntax of Rebeca, software engineers are encouraged to use Rebeca for verifying software systems and protocols. The reactive objects (rebecs) in Rebeca play the same role as asynchronous processes. This makes Rebeca fit for the partial order reduction technique.

In this paper, we describe the architecture of Modere, the 'Model-checking Engine of Rebeca'. The tool manages the parallel execution of the reactive objects in a given Rebeca model, and provides the infrastructure for handling the communication among them. It uses Nested Depth First Search (NDFS) for performing automata-theoretic model checking. By storing the local states for each reactive class separately, and a "never release" strategy in memory management, Modere uses as little memory as possible. In addition, partial order reduction technique has been implemented in Modere for combatting the state explosion problem. Previously, Rebeca could be translated to Promela with R2P [17] and model checked with SPIN [8] or translated to SMV and model checked with NuSMV [14]. Comparisons show that partial order reduction which is used in Modere yields significant reductions compared to the previous approaches.

In the next Section, we present the semantics of Rebeca models. In Section 3, we explain the partial order reduction method and how it can be used in model checking Rebeca. A brief comparison of applying partial order reduction on Rebeca and Promela is also given in this section. In Section 4, the details of the model checking engine of Rebeca are elaborated. Section 5 demonstrates the empirical results of model checking a well known case study using our tool. Section 6 concludes the paper and presents the future work.

2. REBECA

Rebeca [15, 16] is an actor-based model [7, 1] with operational semantics. A Rebeca model, consisting of a set of concurrent rebecs (reactive objects), is defined as a closed system in which rebecs can communicate *only* through asynchronous message passing with no explicit receive and have unbounded buffers to store the incoming messages. Furthermore, Rebeca inherits from actor the dynamically changing topology and dynamic creation of objects.

2.1 Basic Definitions

A Rebeca model is constructed by the parallel composition of a set of rebecs, written as $R = \|_{i \in I} r_i$, where I is the index set that is used to identify each rebec. The number of rebecs, and hence I, may change dynamically while a model is being executed. Each rebec is instantiated from a reactive-class (denoting its type) and has a single thread of execution. A reactive-class defines a set of local variables that constitute the local state of its instances. In addition, the behavior of each rebec is characterized by the set of message servers in the reactive class that determines its type.

The rebecs communicate by sending asynchronous messages. The messages that can be serviced by rebec r_i are denoted by the set M_i. There is a message server corresponding to each element of M_i. There is at least a message server '*initial*' in each rebec, which is responsible for performing the initialization tasks, and is assumed to be in the queues of all rebecs in the initial state. Each rebec has an unbounded queue for storing its incoming messages. In addition, each rebec r_i has a static ordered list of known rebecs, whose indices are collected in K_i. The rebec r_i can send messages to its known rebecs $j \in K_j$. As a result, we can build the *static communications graph* of a given Rebeca model with a directed graph, where nodes are the rebecs, and there is an edge from r_i to r_j when $j \in K_i$. Nevertheless, rebecs can use variables that hold rebec indices for sending messages to a rebec, which is not known statically.

The behavior of a Rebeca model is defined as the interleaving of the enabled actions of the enabled rebecs at each state. A rebec is said to be enabled if its queue is not empty. In that case, the message at the head of the queue determines the enabled action of that rebec. At the initial state, there is an '*initial*' message in the queues of all rebecs. The execution of the model continues as rebecs send messages to each other and the corresponding enabled actions are executed. In Rebeca, each message server is executed atomically. Therefore, each message server corresponds to an action. To get more familiar with Rebeca, visit some of the case studies on Rebeca homepage [15].

2.2 The Formal Semantics of Rebeca

The semantics of a Rebeca model is expressed with a labeled transition system $\langle S, A, T, s_0 \rangle$, where S is the set of global states, A is the set of actions, T is the set of labeled transitions, and s_0 is the initial state of the system.

A global state of the system is defined as the combination of the local states of all rebecs: $s = \prod_{j \in I} s_j$. Each local state of a rebec r_j is identified by the values assigned to its local variables, together with the messages (and their parameters and sender) in its queue. The local variables of a rebec may contain either *data* (called data variables) or a *rebec index* (called rebec variables). Without loss of generality, we assume that all data variables take values from the domain set D. This domain set includes the *undefined* value represented by $\perp$. Furthermore, all rebec variables take values from $I \cup \perp$, where I is the index set, and $\perp$ again represents the undefined value.

It is also necessary to distinguish between the message, sender and parameter queues. Suppose that the message servers of r_j accept at most h_j number of parameters. Therefore, r_j has one message queue, one sender queue, and h_j parameter queues. To make queues easier to represent, we consider each queue as an array of variables. We assume an upper bound x_j on the number of the queue elements of r_j (all queues of r_j have the same upper bound). The domain of the message queue variables is $M_j \cup \perp$, where $\perp$ is re-used to represent an empty queue element. The domain of the sender queue variables is $I \cup \perp$, namely the same as rebec variables. Finally, the domain of parameter queue variables is $D \cup I$, which means that both data and rebec variables may be used as parameters to messages. We denote the i'th local, message queue and sender queue variable of rebec r_j as $r_j.v_i$, $r_j.m_i$ and $r_j.s_i$, respectively. The i'th element of the k'th parameter queue is written as $r_j.p_{ki}$.

Assuming that there are w_j local variables in r_j, a local state of r_j can be represented formally as $s_j = (r_j.v_1, \ldots, r_j.v_{w_j}, r_j.m_1, \ldots, r_j.m_{x_j}, r_j.s_1, \ldots, r_j.s_{x_j}, r_j.p_{11}, \ldots, r_j.p_{h_j x_j})$, where $h_j \geq 0$, $x_j \geq 1$ and $w_j \geq 0$. In the initial state s_0, $r_i.m_1 =$ '*initial*' for all rebecs r_i. If the *initial* message server of r_i accepts i_j parameters, the variables $r_j.p_{11}$, $r_j.p_{21}$, $\ldots$, $r_j.p_{h_j 1}$ are also initialized as specified in the model. All other (local and queue) variables are assigned the value $\perp$.

The transition relation $T \subseteq S \times A \times S$ is defined as follows. There is a transition $s \xrightarrow{a_j} t$ in the system, if the action a from rebec r_j is enabled at state s, and its execution results in the state t. The action a_j is enabled at state s, if $r_j.m_1$ is equal to the message corresponding to a.

In the following, we define the different possible kinds of sub-actions that a transition $s \xrightarrow{a_i} t$ may contain. In the formulas below, a $\leftarrow$ represents an assignment, where the value of the expression on the right hand side is computed with regard to variables in s, and is assigned to the variable on the left hand side, in state t.

1. Message removal: This sub-action includes the removal of the first element of message, sender and parameter queues and implicitly exists in all actions:
   $\forall_{0<i<x_j} r_j.m_i \leftarrow r_j.m_{i+1}$, and $r_j.m_{x_j} \leftarrow \perp$, and
   $\forall_{0<i<x_j} r_j.s_i \leftarrow r_j.s_{i+1}$, and $r_j.s_{x_j} \leftarrow \perp$, and
   $\forall_{0<i<x_j, 0<k\leq h_j} r_j.p_{ki} \leftarrow r_j.p_{k(i+1)}$, and $r_j.p_{k(x_j)} \leftarrow \perp$.

2. Assignment: An assignment can be a statement like '$w \leftarrow d$', where w is a local variable in r_j. If w is a data variable, d must be a value from $D \setminus \perp$, representing an expression evaluated using the values of the data variables in state s. If w is a rebec variable, d is either another rebec variable or an argument of the containing message server (which must in turn correspond to a rebec index). As a result of this assignment, the value of d in state s is assigned to w in state t.

3. Rebec creation: This statement has the form '*new* $rc(kr_1, \ldots, kr_m) : (p_1, \ldots, p_d)$' , where rc is the name of a reactive-class, each kr_i represents an index from the current set I (at state s), and p_k shows the

k'th parameter to the *initial* message. This sub-action results in a new index v being added to I (in state t), which is assigned to the newly created rebec. Hence, the global state t will also include the local state of r_v. In state t, each kr_i shows the index of the i'th known rebec of r_v, the message *initial* is placed in $r_v.m_1$, and the parameters p_1, ..., p_d are placed in $r_v.p_{11}$, ..., $r_v.p_{d1}$, respectively, and $r_v.s_1$ is assigned the value j (the creator rebec). All other (local and queue) variables of r_v are undefined ($\perp$).

4. Send: In dynamic Rebeca models, messages can be sent both to known rebecs, and to rebec variables. The rebec r_j may send a message m with parameters $n_1, \ldots, n_{h_k}$ to r_k, where $m \in M_k$, and either k belongs to K_j[1] or r_j has a rebec variable that holds the value k. In addition, n_i may be a data parameter ($n_i \in D$) or a rebec parameter ($n_i \in I$), where the same rules as the right hand side of an assignment applies. This send operation, results in the message m being placed in the first empty slot of the queue of the receiving rebec:

   If $\exists_{0<y\leq x_k}(r_k.m_y = \perp \wedge \forall_{0<z<y} r_k.m_z \neq \perp)$ then

   $r_k.m_y \leftarrow m, r_k.s_y \leftarrow j, \forall_{1\leq i\leq h_k} r_k.p_{iy} \leftarrow n_i$

   Otherwise, x_k must be increased.

3. PARTIAL ORDER REDUCTION

Partial order reduction is an efficient technique used in reducing the state space size when model checking systems consisting of a number of independent concurrent processes [6]. The concurrency in such systems is modeled by interleaving all the enabled transitions at each state. However, with regard to the specification being checked, there need not be a total ordering among the transitions of the system. Therefore, by finding the partial order relation amongst the transitions, it is possible to avoid all the interleaved sequences of those transitions.

In this paper, we explain and use *static* partial order reduction [9]. Static partial order reduction is considered the most practical variant of the partial order techniques, and is implemented in SPIN (a well known tool that uses partial order reduction, and is used for model checking Promela models) [8]. Static partial order reduction is based on detecting the safe transitions in a given system. A transition is called safe if the action associated to it is invisible and globally independent (see the definition of a transition in a labeled transition system in Section 2). An invisible action is one that does not affect the satisfiability of the propositions used in the specification. For example, in Figure 1(a), if q is not included in the specification, the action i is invisible.

Two actions i and j are called independent if the following conditions hold. First, when they are both enabled at a given state s, the execution of one of them cannot disable the other. In addition, the consecutive execution of both i and j, no matter which one is executed first, must result in the same state. These conditions are shown for the actions i and j in Figure 1(a). The actions of one process are considered dependent. Different transitions from one process may be enabled at the same state due to an action containing

[1]As stated earlier, by K_j, we mean the ordered list of (the indices of) the known rebecs of r_j.

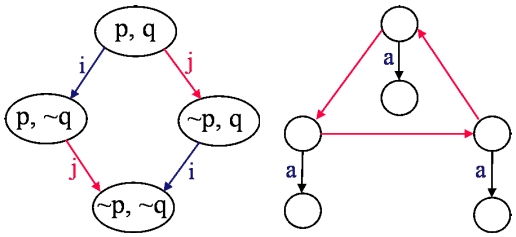

Figure 1: (a) Independent actions, (b) The ignoring problem.

nondeterministic assignments. In this case, the execution of one of these transitions leads to a state where the other transitions caused by that action are disabled. Therefore, the *independence* relation for actions is not a symmetric relation. An action is called globally independent, if it is independent from all the actions of other processes (rebecs). For example, an action that affects only the local variables of a rebec is globally independent.

Intuitively, the execution of a safe transition at a given state leads to a state where all other enabled transitions (from other processes) remain enabled. So the execution of other enabled transitions can be postponed to the future states. Therefore, we define a subset of the enabled transitions, called the *ample* set, which contains the minimal transitions that need to be explored at each state. Since the transitions of one process are interdependent, we need to find a process whose enabled transitions are safe. Static partial order reduction specifies that the enabled transitions of such a process can be chosen as the ample set. However, care must be taken not to postpone the other enabled transitions forever. This is called the *ignoring* problem. The ignoring problem may occur in the case of a loop, an example of which is shown in Figure 1(b). In this figure, the transition 'a' may be ignored completely, if the other transitions in the loop are safe. One of the solutions to the ignoring problem, called the *stack proviso*, requires that the execution of none of the transitions in the ample set should cut the search stack (which definitely closes a loop). If no proper ample set can be found, all the enabled transitions from all processes must be explored. Such a state is said to be fully explored.

3.1 Partial Order Reduction for Rebeca

For applying the static partial order reduction technique to model checking a given Rebeca model, we need to determine which of its actions are safe by inspecting the model statically. An action is safe iff all its sub-actions are safe. The safety of a sub-action is defined in the same way as the safety of actions. In this section, we intuitively explain the conditions that need to be checked statically for different sorts of sub-actions in a Rebeca model to make sure they are safe. As explained in Section 2, there are four types of sub-actions in Rebeca.

The *implicit* 'message removal' sub-action is always safe. This sub-action is invisible, because a specification is not allowed to use queue variables. Furthermore, it is globally independent, because it has no effect on the (local or queue) variables of other rebecs. The safety of this sub-action implies that we only need to check the safety of the explicit sub-actions.

An 'assignment' changes the value of a local variable and

has no effect on the variables of other rebecs, so all assignments are globally independent. As a result, an assignment is safe if it is invisible. An assignment in the 'initial' message server is always invisible, because all variables are just being initialized. In other message servers, the variable on the left hand side of the assignment should be checked to see if it is used in the specification.

Since a specification is not allowed to use queue variables, all 'send' operations are also invisible. Therefore, the safety of a 'send' depends on its being globally independent. At the first glance, it seems that two send operations do not disable each other. However, considering a 'send' as placing a message at the queue tail of the receiving rebec, it can be seen that each 'send' changes the queue tail. Therefore, two send operations to the same queue are always interdependent. Nonetheless, if we can find a queue, which is accessed (for write) only by one rebec, the send operations to this queue are globally independent. In addition, remember that queues in Rebeca are virtually unbounded[2], so a send is never blocked. Therefore, if a rebec is included in the known rebecs list of only one rebec, the 'send' operations to that rebec are safe.

If there are dynamic send operations in a Rebeca model, namely sending a message via a rebec variable, it is impossible to determine statically if a queue is accessed (for write) only by one rebec. This is also the case if the model contains dynamic rebec creation statements. In addition, the safety of dynamic rebec creations statements cannot be determined statically, either. Therefore, in dynamic Rebeca models, i.e. models containing dynamic statements, only assignments and (implicit) message removals can be safe.

3.1.1 A Comparison with SPIN

In SPIN, assignments to *local* variables are considered safe, while allowing the specification to use only the *global* variables. So the choice of using local or global variables by the modeler determines the safety of the assignments to these variables.

On the other hand, channel operations are only safe if the executing process has the proper exclusive access (read or write) to that channel. In addition, since channels may become full, the complete safety of a read or write depends on the used channel not being empty or full, respectively. This is called conditional safety. However, in Rebeca, read from channels (queues) is performed implicitly in every message server for removing the message at the head of the queue, which is always safe (see the previous subsection). In addition, the unbounded queues in Rebeca eliminate the need for conditional safety.

4. MODERE

The 'Model checking Engine of Rebeca (Modere)' is the part of Rebeca model checker that performs the real model checking. Modere uses the automata-theoretic method for model checking [18]. In this method, the system and the negation of the specification are specified each with a Buchi automaton. The system satisfies the specification iff the language of the automata generated by the synchronous product of these two automata is empty. Otherwise, the product automata has an accepting cycle (cycle with an accepting state) that shows the undesired behavior of the system.

[2]We need to assume an upper bound on the length of each queue, so that the model can be model checked. However, this length can be increased when queue overflow is reported.

```
a  int cnt = 0;
b  State seed = NULL;
c  bool cycle = false;
d  systemDfs (initState); // start-up

1  proc systemDfs (State now)
2    int flag = -1;
3    for int r=1 to n do // n = number of rebecs
4      bool disabled = true;
5      if (cnt == r)
6        cnt++;
7        if (flag == -1) flag = r;
8      fi
9      State[] nxt = getNextSysStates(now.sys, r);
10     for each State s in nxt do
11       disabled = false;
12       specDfs (s);
13       if (cycle) return;
14     od
15     if (flag!=-1 && !disabled && cnt==r+1) cnt--;
16   od
17   if (flag != -1) cnt = flag;
18 end

19 proc specDfs (State now)
20   State[] nxt = getNextSpecStates(now.spec,now.sys);
21   for each State s in nxt do
22     if (cnt==n+1 && s==seed)
23       cycle = true;
24       return;
25     fi
26     if (s,cnt) not in StateSpace
27       add (s,cnt) to StateSpace
28       systemDfs (s); // continue current DFS
29       if (cnt==0 && isAccepting(s))
30         seed = s;
31         cnt = 1;
32         systemDfs (s); // start 2nd DFS
33         cnt = 0;  // back to 1st DFS
34       fi
35     fi
36     if (cycle) return;
36   od
38 end
```

Figure 2: Recursive pseudo-code for implementing fairness in NDFS for Modere.

Given a Rebeca model, the Rebeca model checker generates the automata for the system and the negation of the specification, as C++ classes (in separate files). These files are placed automatically beside the file that contains the engine of Modere. The whole package must be compiled to produce an executable for model checking the given Rebeca model.

Modere uses the Nested Depth First Search (NDFS) [10] algorithm for computing the product automata on-the-fly. In NDFS, one DFS is used to generate the product automaton, and at the same time, find the accepting states. As shown in Figure 2, the synchronous product is computed by the interleaving of `systemDFS()` and `specDFS()`. The second DFS is called once per each accepting state (of the product automaton) as seed in a post-order fashion (lines 29-34). The second DFS checks the reachability of the seed state from itself, which means an accepting cycle (lines 22-25). In our algorithm, non-zero values of `cnt` show that we are in the second DFS. The different values are used for fairness, which is explained shortly.

Modere uses a non-recursive implementation of the code

in Figure 2, and handles the search stack manually. Furthermore, it does not dynamically allocate a memory block on heap, if it needs to release it afterwards. This helps avoid fragmentation of memory.

Modere has an object oriented architecture. In Modere, reactive classes are defined as C++ classes and rebecs are their instances. Each reactive class (and similarly the class representing the specification automaton) has a local hash table for storing its local states, where each local state is assigned a unique id number. Each global state is the composition of the local states of all rebecs and the specification automaton, which are in turn represented by their id numbers. This method is similar to the method used in SPIN, and referring to [8] it causes up to 60% reduction in storing the state space.

All rebecs have an '`execute()`' method that, given (the id number of) a local state, returns (the id number of) the next local states[3]. State exploration in Modere takes place by calling the `execute()` method of all the enabled rebecs at each state (inside the `getNextSysStates()` at line 9).

In a Rebeca model, we need to consider only the fair sequences of execution. An infinite sequence is considered (weakly) fair when all the rebecs are infinitely often executed or disabled. For this purpose, a slightly different version of the algorithm proposed in [2] is used in Modere. Suppose there are n rebecs in the system. We assume that there are $n + 2$ copies of the state space. The first DFS (in NDFS) is performed in the state space zero (lines a and 33). When starting the second DFS, the algorithm switches to the state space 1 (line 31). Then, when in state space i, the execution of rebec i (or its being disabled) causes a transition to the state space $i + 1$ (line 6). If the seed state is visited in state space $n + 1$ (line 22), a fair accepting cycle has been detected. In practice, these $n + 2$ copies are implemented using only $n + 2$ extra bits for each stored state.

4.1 Implementing Partial Order Reduction in Modere

Naive combination of static partial order reduction with NDFS confronts a problem. Since the contents of the first and the second DFS stacks are different, the stack proviso may cause different states to be fully explored in the two DFS routines. To solve this problem, the first DFS can use one bit to show whether each state was fully explored. This bit is used in the second DFS instead of the stack proviso. In addition, it is necessary to change the second DFS to report a cycle upon revisiting a state which is on the first DFS stack [10].

For using static partial order reduction, the safe actions of each reactive class are identified statically before Modere is executed. There is an array called "safety" associated to each reactive class. This array includes one boolean element for each action and shows whether the action is safe.

The first DFS algorithm in Modere is altered so that it first checks if it can find a rebec whose all enabled transitions are safe. If such a rebec is found, its enabled transitions are executed. If none of its transitions lead immediately to a state on the stack, all other enabled rebecs are ignored (in fact their execution is postponed to the succeeding states). However, if any of the next states was on stack, the execution of that rebec is terminated and the algorithm continues with the next rebec. If all these attempts fail, the state is marked as 'fully expanded' and then all the enabled rebecs are executed.

Note that every state visited by the second DFS is already visited by the first DFS (due to the post-order nature of NDFS explained earlier in this section). The second DFS, at each state, simply checks if the state is marked as 'fully expanded'. If not, all the rebecs whose enabled actions are safe are executed. This ensures that the second DFS at least explores all the states visited by the first DFS. However, it may cause the second DFS to execute some rebecs that were not executed by the first DFS. This problem is solved by allowing only the first DFS to store new states. Therefore, the second DFS cannot explore any states that were not first visited by the first DFS.

4.2 Rebeca to Promela Convertor

In [17], a tool for translating Rebeca to Promela (R2P) is introduced. In that tool, each reactive class is translated to a `proctype` and Promela processes represent rebecs. Channels are used to as a substitute to rebec queues. The equivalent Promela code for each message server is enclosed in an `atomic` block. Since only global variables are allowed in the property specification, all rebec variables must be mapped to global variables in the equivalent Promela model. This way, the generated Promela model does not depend on the property that will be checked.

The generated Promela code can be fed to SPIN to be model checked. However, using global variables renders all assignments unsafe (see Section 3.1.1). Furthermore, since SPIN is basically designed for fine-grained interleaving, it works very slowly for the atomic message servers in Rebeca. In addition, channel operations are conditionally safe in SPIN, which slows down the execution of 'send' and 'message removal' sub-actions. All these factors make Modere much faster and more efficient for model checking Rebeca.

5. EMPIRICAL RESULTS

We used the dining philosophers problem to show the performance of Modere compared to R2P. In this problem, there are some philosophers that are sitting round a table. There is a fork between each two philosophers. Each philosopher repeatedly thinks and eats. S/he must take both forks on his two sides for eating, and after eating releases them.

To model this problem, two reactive classes are introduced: Philosopher (3 variables) and Fork (4 variables). The only safe action in this model is the initial message server of Fork, which contains only assignments. The complete Rebeca code is not included due to lack of space. The starvation-freedom property states that all philosophers can finally eat whenever they try. The safety property requires a fork not to be taken by two philosophers simultaneously.

Table 1(a) shows that checking the starvation-freedom property increases the number of states, but the safety property leaves it unchanged. This is because the automaton representing safety has only two states, where the second state shows the undesired behavior that never happens in this example. Table 1(b) shows that partial order reduction causes almost 45% reduction in the number of states.

Table 2 shows the results of using SPIN for model checking the Promela file automatically generated from the same Rebeca model by R2P. Because of the different schemes used by SPIN and Modere for storing states, it is not possible to

[3]More than one next local state is possible if there are nondeterministic assignments.

Table 1: Dining philosophers checked with Modere

4 phil/4 fork	Time	Memory(KB)	Global states
No Spec	0:00:01	27,520	46,882
No Starvation	0:00:38	45,048	251,871
Safety	0:00:02	27,520	46,882

(a) No reduction

4 phil/4 fork	Time	Memory(KB)	Global states
No Spec	0:00:01	25,628	27,240
No Starvation	0:00:24	37,544	150,255
Safety	0:00:01	25,628	27,240

(b) Partial order Reduction

Table 2: Dining philosophers translated to Promela and checked with SPIN (using no reduction)

4 phil/4 fork	Time	Memory(KB)	Global states
No Spec	0:04:58	713,292	1.42618e+006
No Starvation	$\gg$2:00:00	$\gg$1,000,000	$\gg$2.1292e+006
Safety	0:05:50	806,424	1.6095e+006

directly compare the number of the generated states. However, the memory usage and the verification times show a dramatic increase with respect to Table 1. Part of it is caused by the inevitable inefficiency imposed by the incompatibilities between Rebeca and Promela. Furthermore, SPIN cannot distinguish between the 'initial' and other message servers. Even if R2P is enhanced to map only visible state variables to global variables, it is still impossible for SPIN to detect the safety of the 'initial' message server of Fork in this example. Therefore, using partial order reduction by SPIN in this example produces no reductions in the results of Table 2.

6. CONCLUSIONS

Rebeca is an actor based language that can be used in object based modeling and verification of reactive systems, specially software protocols and distributed systems. In this paper, we introduced the Model checking Engine of Rebeca called Modere. Modere uses NDFS as an automata-theoretic approach to model checking. It performs model checking on-the-fly. Modere uses static partial order reduction for overcoming the state space explosion problem. For this purpose, safe actions are determined by statically inspecting the model.

We showed that directly model checking Rebeca has more potentials for partial order reduction than translating it to Promela and then using SPIN. The empirical results presented in this paper show that the partial order technique can be quite useful in combatting the state explosion problem. The comparisons show that model checking Rebeca using a model checker specifically written for Rebeca has much better results than the older approaches of first translating a Rebeca model into Promela or SMV and then using their model checkers. Modere requires an order of magnitude less memory space and time.

In future, symmetry is going to be added to the Rebeca model checking engine. The combination of partial order and symmetry can yield even better state space reduction in model checking Rebeca models. Furthermore, compositional verification can be added to the engine, too.

7. REFERENCES

[1] G. Agha. The structure and semantics of actor languages. In *Proc. REX Workshop*, pages 1–59, 1990.

[2] D. Bosnacki. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Technische Universiteite Eindhoven, 2001.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[4] E. Emerson and A. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1–2):105–131, 1996.

[5] E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In E. Brinksma, editor, *TACAS '97*, volume 1217 of *LNCS*, pages 19–34. Springer 1997.

[6] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, 1995.

[7] C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.

[8] G. J. Holzmann. The model checker `SPIN`. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[9] G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *7th IFIP WG6.1 International Conference on Formal Description Techniques*, pages 197–211, 1994.

[10] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32, 1996.

[11] T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming (JLAP)*, 52–53:183–220, 2002.

[12] C. Ip and D. Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.

[13] K. McMillan. *Symbolic Model Checking*. Kluwer Academic, Boston, MA, USA, 1993.

[14] `NuSMV` user manual. `http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html`.

[15] Rebeca homepage. `http://khorshid.ut.ac.ir/~rebeca/`.

[16] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamamenta Informaticae*, 63(4):385–410, 2004.

[17] M. Sirjani, A. Shali, M. M. Jaghoori, H. Iravanchi, and A. Movaghar. A front-end tool for automated abstraction and modular verification of actor-based models. In *IEEE ACSD 2004*, pages 145–150, 2004.

[18] M. Y. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In D. Kozen, editor, *LICS*, pages 322–331, 1986.