Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

Symmetry and partial order reduction techniques in model checking Rebeca

M.M. Jaghoori, M. Sirjani, M.R. Mousavi, A. Movaghar

REPORT SEN-R0704 JULY 2007

ISSN 1386-369X

Symmetry and partial order reduction techniques in model checking Rebeca

ABSTRACT
Rebeca is an actor-based language with formal semantics that can be used in modeling concurrent and distributed software and protocols. In this paper, we study the application of partial order and symmetry reduction techniques to model checking dynamic Rebeca models. Finding symmetry based equivalence classes of states is in general a difficult problem known to be as hard as graph isomorphism. We show how, for Rebeca models, we can tackle this problem with a polynomial-time solution. Moreover, the coarse-grained interleaving semantics of Rebeca causes considerable reductions when partial order reduction is applied. We have also developed a tool that can make use of both techniques in combination or separately. The evaluation results show significant improvements in model size and model-checking time.

2000 Mathematics Subject Classification: 68N30 Mathematical aspects of software engineering ; 68Q60 Specification and verification
1998 ACM Computing Classification System: D.2.4 Software/Program Verification,F4.3.Formal languages, F.3.1 Specifying and Verifying and Reasoning about Programs
Keywords and Phrases: Rebeca; Actor; Partial order reduction; Symmetry reduction; Model checking
Note: This work was supported by: European IST-33826 STREP project CREDO, SYANCO project (DN 62-613), and “The Equational Logic of Parallel Processes” (nr. 060013021) of The Icelandic Research Fund

Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca

Mohammad Mahdi Jaghoori · Marjan Sirjani · MohammadReza Mousavi · Ali Movaghar

Abstract Rebeca is an actor-based language with formal semantics that can be used in modeling concurrent and distributed software and protocols. In this paper, we study the application of partial order and symmetry reduction techniques to model checking dynamic Rebeca models. Finding symmetry-based equivalence classes of states is in general a difficult problem known to be as hard as graph isomorphism. We show how, for Rebeca models, we can tackle this problem with a polynomial-time solution. Moreover, the coarse-grained interleaving semantics of Rebeca causes considerable reductions when partial order reduction is applied. We have also developed a tool that can make use of both techniques in combination or separately. The evaluation results show significant improvements in model size and model-checking time.

Keywords Rebeca · Actor · Partial order reduction · Symmetry reduction · Model checking

The work of the first author is funded by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services.

The work of the second author is partially funded by the "Synthesis and Analysis of Connector Components (SYANCO)" project (DN 62-613).

The work of the third author has been partially supported by the project "The Equational Logic of Parallel Processes" (nr. 060013021) of The Icelandic Research Fund.

M. M. Jaghouri (Corresponding author)
CWI, Amsterdam, The Netherlands
Tel: +31 (0)20 592 4299
Fax: +31 (0)20 592 4199
E-mail: jaghouri@cwi.nl

M. Sirjani
University of Tehran and IPM, Tehran, Iran
E-mail: msirjani@ut.ac.ir

M. Mousavi
Eindhoven University of Technology, Eindhoven, The Netherlands
Reykjavík University, Reykjavík, Iceland
E-mail: m.r.mousavi@tue.nl

A. Movaghar
Sharif University of Technology, Tehran, Iran
E-mail: movaghar@sharif.edu

1 Introduction

Rebeca [38,35,36] (reactive objects language) is an actor-based language [19,2], which can be used at a high level of abstraction for modeling concurrent and distributed reactive systems. The asynchronous message-passing paradigm in Rebeca allows for efficient modeling of loosely-coupled distributed systems. The simple Java-like syntax of Rebeca, unlike most of the traditional notations of formal languages, is an easy-to-learn notation for software practitioners. This encourages software engineers to use Rebeca for verifying software systems and protocols, e.g., [37,20]. Furthermore, Rebeca has been successfully applied in model checking security protocols [34] and system-level hardware design [29].

Rebeca is designed to suit model checking, which is the algorithmic approach to verification. In model checking, all the reachable states of a system are explored and tested against a particular property. State-space explosion, which is the exponential growth of the number of states, is the major obstacle in performing model checking in practice. To overcome this problem, numerous methods have been proposed that avoid the construction of the complete state graph [8]. Among these methods are symbolic verification [31], partial order reduction [40,16,33], and symmetry reduction [12,26]. These techniques are sometimes combined to achieve even more compression in the representation of the system under analysis [15,13,1].

When modeling concurrent systems with Rebeca, *rebecs* (reactive objects) are responsible for the behavior of the system. For modeling concurrency in these systems, the interleaving of the enabled actions from different rebecs is considered. It is, however, not always necessary to consider all the possible interleaved sequences of these actions. The partial order reduction method 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, by avoiding the full interleaving of the enabled actions, some states are excluded from the exhaustive state exploration in model checking [40,16].

Static partial order reduction is the most practical variant of the partial order techniques. It is based on determining the so called *safe* actions by static analysis of the model. At each state, once a safe subset of enabled actions is determined, it suffices to expand only the actions from this subset. Since actions correspond to atomically executed message servers, by excluding even one action from the expansion process, a considerable amount of reduction is gained (see the empirical results). In [27], we described how to determine if an actions is safe in Rebeca. In this paper, we formalize the correctness proof of the approach.

The symmetry reduction technique [12,26], on the other hand, takes advantage of structural similarities in the state-space. The main problem facing this technique is finding the appropriate equivalence classes of states, and is shown to be as hard as graph isomorphism [12,7], which is in turn in NP (but not known to be NP-complete). The problem is usually alleviated by deriving equivalence classes of states by considering the symmetry among higher-level constructs (such as processes or objects). Although the symmetry at a higher level might not induce all the possible symmetries (at state level), but it can be more efficiently handled. For example, the notion of scalar sets were proposed by Ip and Dill in [26], and later used by others (e.g. [6], [17]) to allow the modeler to (explicitly) exhibit the symmetry in the system. In other words, they propose a method for *specifying* the symmetry in the model. However, we try to detect the symmetry with no (or reduced) responsibility on modeler for explicitly specifying it, which is a tedious and error-prone manual task.

Based on the actor-based nature of Rebeca, in [28], we gave a polynomial time solution to this problem for detecting inter-rebec symmetry. The soundness of the algorithm is proved in this paper. The approach is then extended in this paper to cover even more topologies (by detecting intra-rebec symmetry). We also prove the soundness of the new more general algorithm. The algorithm can be adopted in other similar models of computation.

The formal semantics of Rebeca given in [35] is not expressed in enough detail for proving the correctness of the reduction methods. Therefore, we first provide a refined version of the formal semantics (as in [28]). In this semantics, dynamic rebec creation and dynamic change of topology, are neatly handled by introducing rebec variables (variables holding rebec identifiers). This also simplifies proving the inter- and intra-rebec symmetry soundness theorems in the presence of dynamic changes in the topology.

The *model checking engine of Rebeca* (Modere) was introduced in [27] to apply partial order reduction on Rebeca. We explain in this paper how symmetry reduction is added to it. These techniques can be employed separately or in combination. Finally, some case studies are modeled with Rebeca, and model checked with Modere. Symmetry and partial order reduction techniques (separately and together) are applied to these examples. The results show that not only these techniques can each be useful in model checking Rebeca, but, whenever possible, their combination yields reductions of up to 70% in these models.

In summary, this paper describes how partial order and symmetry reduction techniques can be applied in model checking Rebeca. This paper summarizes, integrates and extends the results of [27, 28]. In the following, we illustrate the complete work, while mentioning the contributions of this paper:

- To establish a formal proof for partial order and symmetry reduction algorithms we need a more refined formal semantics for Rebeca than what is available in previous work. A refined formal semantics is first presented in [28] and is used in this paper.
- An algorithm for applying static partial order reduction to model checking Rebeca is proposed in [27]. In this paper, we formalize the correctness proof of the algorithm.
- A polynomial time algorithm for inter-rebec symmetry detection is proposed in [28]. To the best of our knowledge, there is no similar algorithm in the literature, in the sense that it needs no symmetry-related input from the modeler. In this paper, the method is extended to detect intra-rebec symmetry which enables the automatic detection of symmetry in more topologies.
- The formal proofs for the correctness of both (inter- and intra-rebec) symmetry reduction techniques are presented in this paper.
- The model checking engine of Rebeca, Modere, is introduced in [27], which could apply partial order reduction. In this paper, we explain how to implement the algorithms for the inter- and intra-rebec symmetry detection techniques in Modere. Modere is now extended to use symmetry as well as partial order reduction. It can employ the techniques separately or in combination.

In the next Section, we explain the behavior and refined semantics of Rebeca. Section 3 introduces symmetry reduction and in Section 4, partial order reduction is explained. The details of applying symmetry and partial order reduction techniques to Rebeca are also explained. In Section 5, the model checking engine of Rebeca (Modere), which incorporates these reduction techniques, is explained. Section 6 shows the empirical results of model checking some case studies and compares the reductions caused by symmetry, partial order and their combination. Section 7 concludes the paper.

2 Rebeca

Rebeca [38,35,36] is a modeling language with formal semantics based on an operational interpretation of the actor model [19,2]. 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. There are no shared variables in a Rebeca model. Furthermore, Rebeca inherits from actor the dynamically changing topology and dynamic creation of objects.

2.1 Basic Definitions

A Rebeca model is constructed by parallel composition of a set of rebecs, written as $R = \|_{i \in I} r_i$, where I is the set of the indices used for identifying each rebec. The number of rebecs, and hence I, may change dynamically due to the possibility of dynamic rebec creation. 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. The initial state is modeled by instantiating some rebecs.

Rebecs communicate by sending asynchronous messages. The messages that can be serviced by rebec r_i are collected in the set M_i. In the reactive class denoting the type of r_i, there exists a message server corresponding to each element of M_i. Message servers are executed atomically; therefore, each message server corresponds to an action. There is at least a message server '*initial*' in each reactive class, which is responsible for initialization tasks. Each rebec receives this message implicitly upon creation.

Rebecs may have variables that range over rebec indices (called rebec variables). These variables are used to designate the intended receiver when sending a message. By changing the values of rebec variables, one can dynamically change the topology of a model. For each rebec r_i, a subset of its rebec variables are identified as *known rebecs*. The actual values of known rebecs must be provided upon creation (this is enforced by the syntax). We use K_i to denote the (ordered) list of the initial values of known rebecs for rebec r_i. The ordering of this list is implied by the order in which they are defined in the corresponding reactive class. The initial topology (initial communication graph) of a system can be represented by a directed graph, where nodes are rebecs (those created at the initial state), and there is an edge from r_i to r_j iff $j \in K_i$.

Each rebec has an unbounded queue for storing its incoming messages. 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. The behavior of a Rebeca model is defined by the fair and interleaved execution of enabled actions. At the initial state, a number of rebecs are created statically, and an '*initial*' message is implicitly put in their queues. The execution of the model continues as rebecs send messages to each other and the corresponding enabled actions are executed.

Example 1 (Dining Philosophers) In this problem, there are a number of philosophers sitting around a table. There is one fork between each two philosophers. Each philosopher needs the forks on his/her both sides for eating. To model this problem in Rebeca, two reactive classes are introduced: `Phil` and `Fork`. Each `Phil` knows (as its known rebecs) his/her left and right `Forks`, and each `Fork` knows its left and right `Phils`. Figure 1 shows the Rebeca code of this example.

The parameter '3' passed to the reactive classes denotes the upper bound on queue length, provided by the modeler. This upper bound is used to avoid infinitely large states, due to unbounded queues (cf. x_j defined in next sub-section). The model checker will produce a proper message if queue overflow occurs. Having model checked this example, we know that 3 is the minimum upper bound with no queue overflow.

The `main` section of the code, specifies the initial configuration of the model. The first list of parameters passed to each rebec represent the values to be assigned to known rebecs. The parameters to the initial message server can be provided separately after the colon (empty list in this example). To avoid starvation, for every other philosopher we bind the left and right forks in the reverse order.

2.2 The Formal Semantics of Rebeca

In [35], the formal semantics of Rebeca is expressed as a *labeled transition system (LTS)*. In this section, we introduce and use a more detailed semantics for Rebeca, again expressed as an LTS. The details introduced here are useful for the proofs given in the remainder of this paper.

Definition 1 (LTS) $R = \langle S, A, T, s_0 \rangle$ is called a labeled transition system, where:

- The set of *global states* is shown as S.
- s_0 is the *initial state*.
- A denotes the set of the *actions* (message servers) of different reactive classes.

```
reactiveclass Fork(3) {
   knownobjects {
     Phil philL, philR;
   }
   statevars {
     boolean busy, requester;
   }
   msgsrv initial() {
     busy = false;
   }
   msgsrv request() {
     if (sender != self) {
        if (sender == philL) {
           if (busy) {
              requester = true;
              self.request();
           } else {
              busy = true;
              philL.permit();
           }
        } else {
           if (busy) {
               requester = false;
               self.request();
           } else {
               busy = true;
               philR.permit();
           }
        }
     } else {
        if (busy) {
           self.request();
        } else {
           busy = true;
           if (requester) {
              philL.permit();
           } else {
              philR.permit();
           }
        }
     }
   }
   msgsrv release() {
     busy = false;
  }
}
```

```
reactiveclass Phil(3) {
   knownobjects {
     Fork forkL, forkR;
   }
   statevars {
     boolean eating, fL, fR;
   }
   msgsrv initial() {
     fL = false;
     fR = false;
     eating = false;
     self.arrive();
   }
   msgsrv eat() {
     eating = true;
     self.leave();
   }
   msgsrv permit() {
     if (sender == forkL) {
        fL = true;
        forkR.request();
     } else {
        fR = true;
        self.eat();
   } }
   msgsrv arrive() {
     forkL.request();
   }
   msgsrv leave() {
     fL = false;
     fR = false;
     eating = false;
     forkL.release();
     forkR.release();
     self.arrive();
}  }
main {
  Phil phil0(fork0, fork3):();
  Phil phil1(fork0, fork1):();
  Phil phil2(fork2, fork1):();
  Phil phil3(fork2, fork3):();
  Fork fork0(phil0, phil1):();
  Fork fork1(phil1, phil2):();
  Fork fork2(phil2, phil3):();
  Fork fork3(phil3, phil0):();
}
```

Fig. 1 Rebeca code for the dining philosophers problem

– The *transition relation* is defined as $T \subseteq S \times A \times S$.

Since instances of similar reactive classes have similar actions, actions of each rebec are indexed by the identifier of that rebec. In the case of a nondeterministic assignment in a message server, it is possible to have two or more transitions with the same action from a given state. We may write $s \xrightarrow{a_i} t$ for (s, a_i, t).

Definition 2 (Data Variables) The variables for holding and manipulating data are called data variables. We assume that all data variables take values from the domain set D, which includes the *undefined* value (represented by $\perp$).
We may use a subscript 'd' to distinguish data variables.

Definition 3 (Rebec Variables) Rebec variables are those holding rebec indices. All rebec variables take values from $I \cup \perp$, where I is the index set, and $\perp$ again represents the undefined value.
We may use a subscript 'r' to distinguish rebec variables.

Rebec variables can participate in different expressions, only when:

- assigned to other rebec variables;
- compared for (in)equality;
- used to specify the receiver of a send statement; or,
- assigned (the index of) a dynamically created rebec.

Nevertheless, they can be passed around as arguments to messages, too.

For each rebec r_j, we assume one message queue ($r_j.m[\,]$) and one sender queue ($r_j.s[\,]$). Consider the number of parameters that each message server accepts. If h_j is the maximum of these numbers, we also need h_j parameter queues ($r_j.p_1[\,], \ldots, r_j.p_{h_j}[\,]$). The contents of these queues are to be considered together. For example, $r_j.m[1]$ and $r_j.s[1]$ show the oldest (unprocessed) message and its sender, respectively. The parameters to this message are kept in $r_j.p_1[1], \ldots, r_j.p_{h_j}[1]$.

We bound these arrays with at most x_j number of elements, to make it possible to perform model checking. However, remember that this upper-bound is supplied by the modeler and must be increased in case of a queue overflow. The domain of the message queue variables is $M_j \cup \perp$, where $\perp$ is re-used to represent an empty queue element. The sender queue variables are treated as rebec variables, while parameter queue variables can be either data or rebec variables.

Definition 4 (Global State) A global state is defined as the combination of the local states of all rebecs: $s = \prod_{j \in I_s} s_j$, where I_s is the set of rebec indices in the current state. Note that I_s may change in the course of transitions due to dynamic rebec creation.[1]
Each local state of a rebec r_j can be written formally as: $s_j = (r_j.v_1, \ldots, r_j.v_{w_j}, r_j.m[1], \ldots, r_j.m[x_j], r_j.s[1], \ldots, r_j.s[x_j], r_j.p_1[1], \ldots, r_j.p_{h_j}[x_j])$, where $h_j \geq 0$, $x_j \geq 1$; and $w_j \geq 0$.
Local (data or rebec) variables in r_j are represented by $r_j.v_i$; and w_j shows the number of local variables in r_j.

Definition 5 (Initial state) In the initial state s_0, a number of rebecs are created as indicated in the model. For every $j \in I_{s_0}$, $r_j.m[1]$ is set to '*initial*'; the variables corresponding to known rebecs are initialized accordingly; furthermore, if the *initial* message server of r_j accepts n_j parameters other than known rebecs ($n_j \leq h_j$), the variables $r_j.p_1[1]$, $r_j.p_2[1]$, $\ldots$, $r_j.p_{n_j}[1]$ are also initialized as specified in the model. All other (local and queue) variables (including $r_j.s_1$) are assigned the value $\perp$.

The transition relation is defined as follows. There is a transition $s \xrightarrow{a_j} t$ in the system, iff the action a_j (from rebec r_j) is enabled (i.e., $r_j.m[1] = a$) at state s, and its execution results in state t. Each action is defined as a (finite) sequence of some sub-actions. Henceforth, we define the different possible kinds of sub-actions. In the formulas below, the symbol $\leftarrow$ 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, plus shifting other elements of the queues. This sub-action 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,
   $\forall_{0<i<x_j, 0<k\leq h_j} r_j.p_k[i] \leftarrow r_j.p_k[i+1]$, and $r_j.p_k[x_j] \leftarrow \perp$.
2. *Assignment*: An assignment is a statement of the form '$w \leftarrow d$', where w is a local variable in r_j. If w is a data variable, d must take values from $D \setminus \perp$. It represents an expression evaluated using the values of the (local or parameter) data variables in state s. If w is a rebec variable, d can be either a (local or parameter) rebec variable, or the index assigned to a dynamically created rebec (see next item). As a result of this assignment, the value of d is assigned to w in state t.

[1] The subscript s of I_s may be omitted for simplicity in presentation, when no ambiguity arises.

3. *Rebec creation*: This statement has the form '$new\ rc(kr_1, \ldots, kr_m) : (p_1, \ldots, p_z)$' , where rc is the name of a reactive-class, each kr_i is a rebec variable, and p_k shows the k'th parameter to the *initial* message. This sub-action chooses a new index $v \notin I_s$ to be assigned to the newly created rebec, and $I_t \leftarrow I_s \cup \{v\}$. Hence, the global state t will also include the local state of r_v. In state t, the known rebec variables of r_v are initialized by the values of $kr_1 \ldots kr_m$; the message *initial* is placed in $r_v.m[1]$; the values of parameters p_1, …, p_z are placed in $r_v.p_1[1]$, …, $r_v.p_z[1]$, 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*: Rebec r_j may send a message m with parameters $n_1, \ldots, n_z$ to r_k, provided that $m \in M_k$, r_j has a rebec variable that holds the value k, and $z \leq h_k$ is the number of parameters that message m accepts. Each parameter n_i may be a data parameter ($n_i \in D \setminus \perp$) or a rebec parameter ($n_i \in I_s$), where the same rules as the right-hand side of an assignment apply. This send statement, results in the message m being placed in the first empty slot (tail) of the queue of the receiving rebec:

```
If $\exists_{0<y\leq x_k}(r_k.m[y] = \perp \wedge \forall_{0<u<y} r_k.m[u] \neq \perp)$ then
        $r_k.m[y] \leftarrow m$,
        $r_k.s[y] \leftarrow j$,
        $\forall_{1\leq i\leq z} r_k.p_i[y] \leftarrow n_i$ (other elements keep their $\perp$ value).
Otherwise, $x_k$ must be increased.
```

3 The Symmetry Reduction Technique

Roughly speaking, the aim of the symmetry reduction technique [26,12] is to find those parts of a system that yield similar behaviors. Viewing an LTS as a graph, the intuitive idea is to find those sub-graphs with the same structure, and constructing only one of these sub-graphs during state exploration. In this section, we present the formal definitions and the outline of the method for using symmetry reduction in model checking.

3.1 Preliminaries

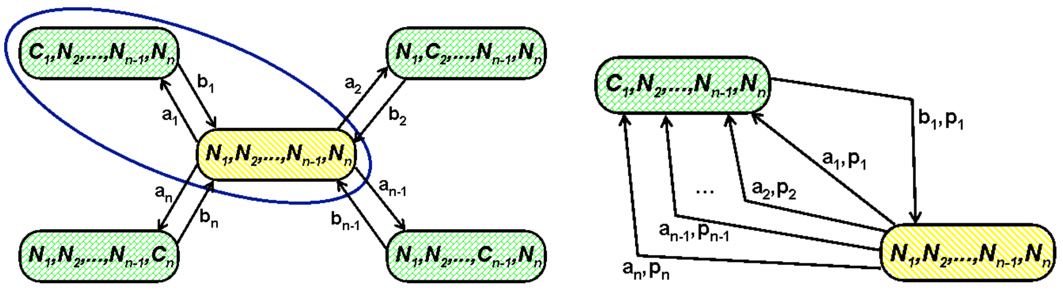
Consider a system M, consisting of n concurrently executing processes, represented by a labeled transition system $M = \langle S, A, T, s_0 \rangle$.

Definition 6 (Permutation) A permutation $\pi : S \rightarrow S$ is defined as a bijection (1-1 and onto function) on the set S of global states.
The set of all permutations on S is shown by $SymS$.

Definition 7 (Automorphism in LTS) A permutation $\pi : S \rightarrow S$, is said to preserve the transition relation when, if $s \xrightarrow{a} t \in T$ then $\pi(s) \xrightarrow{\pi(a)} \pi(t) \in T$. Such a permutation is called an *automorphism* of M, if $\pi(s_0) = s_0$.
The set of automorphisms of M is denoted by $AutM$.

The set $SymS$ (and similarly $AutM$) forms a group [18] with respect to functional composition (denoted by $\circ$). As a result, if $p_1, p_2 \in SymS$, then the permutation obtained by applying p_2 and then p_1, written as $p_1 \circ p_2$, is also in $SymS$. From now on, we use $SymS$ and $AutM$ to refer to the groups. Any subset G_1 of a given group G, which retains the characteristics of a group, is called a subgroup of G, written as $G_1 \leq G$. It is shown in [12] that $AutM \leq SymS$.

Definition 8 (Orbit) Consider a subgroup $G \leq AutM$. Two states s and s' are equivalent with respect to G iff $\exists_{\pi\in G}\pi(s) = s'$. The resulting equivalence classes are called *orbits*.



(a) Selecting representatives (b) Annotated quotient structure

Fig. 2 An example of a symmetric state-space

For example, in the state-space shown in Figure 2-a, the state (N_1, N_2, ..., N_n) forms one orbit, and other states form another orbit. Intuitively, for model checking M, it is sufficient to construct the state-space with one representative from each orbit (Figure 2-b). This example is explained shortly.

Definition 9 (AQS) The *annotated quotient structure (AQS)* for M with respect to $G \leq AutM$ is $\overline{M_G} = \langle \overline{S}, A, \overline{T}, s_0 \rangle$, where $\overline{S}$ is the set of the representative states (which contains exactly one state from each orbit) and $\overline{T} \doteq \{\overline{s} \xrightarrow{a,\pi} \overline{t} \mid \pi \in G, \overline{s} \in \overline{S} \wedge \overline{t} \in \overline{S} \wedge \overline{s} \xrightarrow{a} \pi(\overline{t}) \in T\}$.

In the AQS for M, all the states in any given orbit are replaced by the representative state of that orbit. However, the outgoing transitions of the (representative) states are preserved. These transitions are annotated with a permutation that helps find the original target state (in the original state-space M), by applying the permutation on the representative target state. Consider a path in $\overline{M}$ starting from the initial state. The corresponding path in M can be obtained by consecutively applying the permutations (on the transitions) to states.

Example 2 (taken from [12] with minor changes) Consider a simple system composed of n identical processes that start in a non-critical state and try to enter the critical section (by executing action a_i), and then leave the critical section (by executing action b_i). Associated to each process i is a variable V_i, which is represented by either C_i (when in critical section) or N_i (non-critical section).

Figure 2-a shows the state-space of this system. Consider the index set $I = [1..n]$, and the permutations defined in I as $p_i = (i, i+1, \ldots, n, 1, 2, \ldots, i-1)$.[2] By applying p_i to the indices, we can permute the states, as well. For example, applying any p_i to the state $\{N_1, N_2, \ldots, N_n\}$ results in the same state, while state $\{C_1, N_2, \ldots, N_n\}$ is mapped to $\{N_1, C_2, \ldots, N_n\}$ by p_2.

It is easy to see that all p_i's are automorphisms of M, and $AutM = \{p_i | i \in I\}$. With respect to $AutM$, the state-space is partitioned into two orbits: the state $(N_1, N_2, \ldots, N_n)$ forms one orbit, and the rest form the other orbit. In Figure 2-a the (arbitrarily) chosen representatives of the two orbits are distinguished with a line around them. Figure 2-b shows the annotated quotient structure (AQS) of this system, which contains the selected representative states, and the (properly annotated) outgoing transitions of each (representative) state. □

Suppose that G denotes a group of automorphisms for a system; and the formula representing the desired property is also symmetric with respect to G.[3] Emerson, et.al, in [12], show that using $\overline{M_G}$ instead of M is enough in the automata theoretic approach to model checking. This approach is extended in [14] for model checking under fairness conditions. Bosnacki in [5] shows how symmetry reduction can be employed in *Nested Depth-First Search* (NDFS) [23]. To apply these methods, it is

[2] Henceforth, we write a permutation π as an ordered list, the i'th element of which denotes $\pi(i)$.

[3] We do not address the problem of detecting symmetry in formulas. In [12], it is proposed to use *indexed temporal logics*, and we adopt the same approach.

necessary that the orbit relation (the automorphism group) is first determined. Finding this group, known as the orbit problem, is in its general form in NP, but not known to be NP-complete (it is as hard as graph isomorphism or *GI-complete*) [12, 7]. In the following sections, we show how we can work around this problem for Rebeca efficiently, namely in polynomial time. A comparison with the related work is given in Section 3.4.

3.2 Inter-rebec Symmetry in Rebeca

3.2.1 Motivating Example - Dining Philosophers

Recall the dining philosophers problem in Example 1. This is an example of ring topology. Intuitively, taking into consideration the starvation-free binding of the rebecs, every other philosopher/fork should have symmetric behavior; no matter how the `Phil` and `Fork` classes are implemented.

Assume that the philosophers are assigned the indices 0 to 3 and forks are assigned 4 to 7. Following the same intuition, we would like to infer that, for example, r_0 (phil0) and r_2 (phil2) are equivalent. In other words, one automorphism should be a permutation π, where $\pi(0) = 2$ and $\pi(2) = 0$.

In the following, we define how a permutation on rebec indices can be applied on states. Furthermore, we exploit the conditions that should be checked for such a permutation to be an automorphism.

3.2.2 Formal Definitions

To exploit symmetry in the labeled transition system associated to a Rebeca model, we use the permutations acting on the index set $[1..n]$ of the rebecs. Permutations acting on states can then be derived from these permutations. In this section, we provide the theoretical foundations that help us find the proper automorphism groups efficiently, prior to the real construction of the state-space, i.e., by static analysis of the model.

Since rebecs of the same type (i.e., instances of the same reactive-class) consist of the same message servers, they exhibit similar behaviors. Intuitively, it is reasonable to limit permutations to those that preserve rebec types. It also gives us the intuition that if the communication pattern among rebecs is symmetric, the whole system is symmetric, as well. From now on, we consider a system $R = \langle S, A, T, s_0 \rangle = ||_{i \in I}\, r_i$ of a Rebeca model.

Definition 10 (Permutation on states) Assume that π is a permutation defined on the index set I. The application of π on a global state s, denoted by $\pi(s)$, is defined as follows:

1. The values of data variables, say $r_j.v_{di}$, $r_j.m[i]$ or $r_j.p_{dk}[i]$, in state s, are assigned to the local or queue variables $r_{\pi(j)}.v_{di}$, $r_{\pi(j)}.m[i]$ or $r_{\pi(j)}.p_{dk}[i]$ in state $\pi(s)$, respectively.
2. Suppose the value of a rebec variable, say $r_j.v_{ri}$, $r_j.s[i]$ or $r_j.p_{rk}[i]$, in state s is x. In state $\pi(s)$, the value $\pi(x)$ is assigned to the variable $r_{\pi(j)}.v_{ri}$, $r_{\pi(j)}.s[i]$ or $r_{\pi(j)}.p_{rk}[i]$, respectively.

Definition 11 (Automorphism in Rebeca - adapting Def. 7) A permutation π, defined in I, is said to preserve the transition relation when, if $s \xrightarrow{a_i} t \in T$ then $\pi(s) \xrightarrow{a_{\pi(i)}} \pi(t) \in T$. Such a permutation is called an *automorphism* of R, if $\pi(s_0) = s_0$.

Definition 12 (Preserving rebec types) A permutation π is said to preserve rebec types, if for all i,j such that $\pi(i) = j$, the rebecs r_i and r_j are instances of the same reactive-class.

Definition 13 (Preserving KR relation) If $K_i = (t_1, t_2, \ldots, t_{P_i})$ denotes the ordered list of the indices of the known-rebecs of r_i, where $i \in I$, a permutation π is said to preserve the known-rebec relation iff: $\forall_{i \in I} K_{\pi(i)} = \pi(K_i)$. The application of π on a list is defined as the list obtained by applying π on every element, e.g., $\pi(K_i) = (\pi(t_1), \pi(t_2), \ldots, \pi(t_{P_i}))$.

Theorem 1 (Soundness theorem) *If a permutation π preserves rebec types and $\pi(s_0) = s_0$, then π is an automorphism of R (cf. Definition 11).* (See Appendix A for proof)

Theorem 2 (Initial state theorem) *$\pi(s_0) = s_0$ iff π preserves rebec types and known rebec relation and the parameters to the initial message servers of symmetric rebecs are symmetric, i.e., if $\pi(i) = j$, and p is a parameter to the initial message server of r_j, the corresponding parameter for r_i should have*

- *the same value as p, if p is a data variables; or,*
- *$\pi(p)$, if p is a rebec variables.*

Proof Straightforward from the definition of initial state (Def. 5) and application of permutation on states (Def. 10).

Example 3 (Dining Philosophers) In the dining philosophers example, suppose that philosophers are assigned indices from 0 to 3 and forks are assigned indices from 4 to 7. Now we check if the permutation $\pi = (2, 3, 0, 1, 6, 7, 4, 5)$ is an automorphism of the system. Notice that π satisfies the intuitive condition we expect, i.e., $\pi(0) = 2$ and $\pi(2) = 0$ (see Section 3.2.1).

We should check the conditions in Theorem 1. First of all, notice that this permutation preserves rebec types; because, it does not map `Fork`s to `Phil`s or vice versa.

To check if $\pi(s_0) = s_0$, we first need to make sure that π preserves the known rebec relation (cf. Theorem 2). Consider r_0 (`phil0`), whose known rebecs are `(fork0, fork3)`, i.e., $K_0 = (4, 7)$. We have: $\pi(K_0) = (\pi(4), \pi(7)) = (6, 5)$. Notice that r_6 and r_5 represent `fork2` and `fork1`, respectively, which are the known rebecs of r_2 (`phil2`). So we showed that $\pi(K_0) = K_2 = K_{\pi(0)}$. Similarly, one can ensure that the known rebec relation is preserved for other rebecs, too.

Finally, observe that the initial message servers have no parameters, so it is also easy to see that $\pi(s_0) = s_0$. From Theorem 1, we can deduce that this permutation is an automorphism of the system. □

As shown in the example above, with the help of Theorem 1, we can easily verify if a given permutation is an automorphism of the system. But finding the proper permutations remains a problem. This is usually left to the modeler to use the symmetry-related constructs (e.g., scalar-sets) to exhibit the automorphisms (cf. Section 3.4). In Section 5.3, this problem is solved for Rebeca, and it is shown how we can automatically find the automorphisms. This approach does not need any changes to be made to the syntax of Rebeca, and is not based on any special syntactic notations used by modeler.

3.3 Intra-rebec Symmetry in Rebeca

In systems with loosely coupled objects, symmetry is usually caused by the symmetric composition of objects, and can be detected without scrutinizing the internal behavior of the objects (like in *ring* networks). Such systems are addressed by inter-rebec symmetry explained in the previous sub-section. However, sometimes, e.g., in the *star* topology, the internal structure of some objects also needs to be considered in order to reveal the symmetric composition in the system. Intra-rebec symmetry extends inter-rebec symmetry to address this problem. To this end, a new data type, namely scalar set, is added to the syntax of Rebeca.

When the modeler realizes that the behavior of a reactive class is identical (i.e., symmetric) towards some of its known rebecs, s/he can use scalar sets when introducing and using those known rebecs. The model checker can then consider this internal symmetry and detect the symmetric composition of the whole system. This also helps the modeler avoid repeating the symmetric part of the code in that reactive class.

Note that the notion of scalar sets in Rebeca is different from the traditional notion of scalar sets (cf. [26,6]) in two ways. Firstly, each scalar set is used inside one reactive class (and not globally for the whole system). The smaller scope of scalar sets makes it easier for the modeler to realize the symmetry and use scalar sets to induce it. Secondly, traditional scalar sets are fully symmetric, i.e., any permutation on the (elements of the scalar) set must preserve the system structure, which is too restrictive. In Rebeca, we relax this condition by requiring rotary permutations (cf. Definition 18) instead of all permutations. As a result, the symmetry in more systems can be modeled with the scalar sets. Section 3.4 provides a more comprehensive comparison with related work.

3.3.1 Motivating Example - Bridge Controller

In this example, there is a two-way bridge with the capacity of only one train at a time. For modeling this system in Rebeca, two reactive-classes are introduced: `Train` and `Controller`. Two `Train` instances, `t1` and `t2`, are used for modeling the trains arriving at either side of the bridge. The controller remembers which train has recently passed or is currently on the bridge. Thus, it can provide mutual exclusion; and, schedule the next train in such a way that starvation is avoided.

Suppose we do not consider the internal behavior of the rebecs, and only rely on the communication structure (as required by Theorem 2). Assume the indices 1 and 2 for the trains, and the index 3 for the controller. Then the known rebecs of the controller are $K_3 = (1, 2)$. We expect $\pi = (2, 1, 3)$, which maps `t1` to `t2` and vice versa, to be an automorphism. Surprisingly, π does not preserve the known rebec relation, because: $\pi(K_3) = (\pi(1), \pi(2)) = (2, 1) \neq K_{\pi(3)}$.

In this example, in addition to the communication structure, we need to take into account the internal symmetry of the controller. Intuitively, a permutation that changes the order of the known rebecs of the (internally symmetric) controller should also be an automorphism. More generally, the rebec in the center of a star topology is a good example of internally symmetric rebecs. To exploit the symmetry in such systems, we need to designate the symmetric internal structures of rebecs. To this end, we introduce scalar sets to the syntax of Rebeca. The constructs for employing scalar sets ensure such symmetry.

3.3.2 Scalar Sets in Rebeca

A *scalar set* is a set of consecutive *scalar values*. Scalar values are natural numbers, on which only a special add operation (written as +%, but simply called 'add') is allowed. In fact, +% represents adding modulo the size of the set. Formally, if i denotes a scalar value from the scalar set $[d, d+1, ..., d+n-1]$, then i +% c means $((i + c - d)\ \%\ n) + d$, where % denotes remainder of integer division and c can be any (non-scalar) integer.

To reveal internal symmetry inside rebecs, one can use the newly introduced scalar sets in the declaration of (symmetric) known rebecs. For example, in the bridge controller, the known rebecs of the `Controller` can be defined using a scalar set, as follows:

```
    knownobjects {Train trn[i:1,2];}
```

In the declaration above, `i` is defined as a scalar set with the range [1,2]. This also defines two known rebecs of type `Train` (which is similar to the definition of arrays). The constraints on using scalarsets ensure that the containing reactive class (namely `Controller`) has symmetric behavior towards these known rebecs. A reactive class, in general, may have different groups of such known rebecs, which are defined with different scalar sets.

The scalar sets defined in this way, can be used as the data types for defining other variables; or, again in array-like definition of variables (or even other known rebecs). Variables and known rebecs defined with this array-like syntax are called *grouped* variables and known rebecs, respectively. `signal` and `waiting` in Example 4, defined as follows, are examples of such variables.

```
     boolean signal[i],waiting[i];
```

Definition 14 (Scalar Variables) Scalar variables are variables for holding scalar values. The type of these variables must be a (previously defined) 'scalar set'. These variables are distinguished by a subscript 's' when necessary.

To simplify further discussions about the grouped known rebecs and variables using the same scalar set, we define the concept of cluster below.

Definition 15 (Cluster) Grouped known rebecs, grouped state variables, and scalar variables that are defined using the same scalar set are called a *cluster.* The scalar set identifying each cluster is called the type of that cluster.

For example, in Figure 3, `trn`, `signal` and `waiting` form a cluster of type (the scalar set) `i`. In other words, in each reactive class, there is one cluster associated to each scalar set defined in it.

Definition 16 (Scalar Expression) Given a scalar set 'sclr', a scalar expression of type sclr is defined to be:

– a scalar variable of type sclr; or,
– a scalar variable of type sclr added to an integer; or,
– a nondeterministic choice from all values of sclr.

Scalar expressions are the only means for (indexed) access to grouped known rebecs and variables. It is required that a scalar expression used as an index is of the same type as the cluster containing the accessed known rebecs/variables. For instance, in Example 4, '`i +% 1`' is a scalar expression (of type `i`) that is used for indexing `signal`. Notice that `signal` is in a cluster of type `i`.

For simpler manipulation of grouped known rebecs and variables, we define `forEachValueOf` blocks, that associate a number of statements with a scalar set (for example see message server `Arrive` from `Controller` in Figure 3). Inside the block, the name of the scalar set can be used as a scalar variable, which can, in turn, participate in forming scalar expressions, and hence in indexing grouped known rebecs and variables (but not on the left hand side of assignments).

The behavior of these blocks is similar to the *for* loop construct in programming languages such as C and Java. The statements in such blocks are repeated for each value of the given scalar set. However, the statements forming the body of each block must be written in such a way that the result of the execution is independent of the order of the iterations. Two sufficient (but not necessary) restrictions (taken from [26] and adapted to Rebeca) to obtain this property are that:

– the set of variables written by any iteration should be disjoint from the set of variables referenced (read or written) by other iterations; and,
– each known rebec can be chosen as destination for sending messages, only in one iteration. For instance, you cannot use both `trn[i]` and `trn[i +% 1]` in send statements inside one block.

Furthermore, scalar expressions can be assigned to scalar *state* variables of the same type (i.e., not to the scalar variables associated to `forEachValueOf` blocks). Scalar expressions can also be used to form boolean expressions. The only boolean operation allowed is to compare scalar expressions of the same type for (in)equality.

3.3.3 Using Scalar Sets for Detecting Symmetry

In the following, grouped variables are shown as $r_i.v_g[e]$, where $r_i.v_g$ represents a group of variables local to r_i that share a name v_g, and e is the scalar value used for indexing these variables. For the sake of simplicity in the proofs, we assume that known rebecs and state variables that are not part of any cluster (i.e., those other than grouped known rebecs and variables), form a cluster with the type of a singleton scalar set (say [1]). Therefore, all state variables can be shown as $r_i.v_g[e]$.

Definition 17 (Grouped KR lists) The grouped known rebecs list of r_i is defined as $L_i = (L_{i1}, L_{i2}, \ldots, L_{ib_i})$. Each L_{ij} denotes the ordered list of the known-rebecs sharing the same name.

In this definition, b_i is the number of such lists for r_i, and each L_{ij} is assumed to have d_{ij} elements. Obviously, the elements in each L_{ij} have the same type. For example, in Figure 3, `Controller` has two known rebecs that share the name `trn`. So its grouped known rebecs list contains one sub-list with two elements (see Example 4). It can be seen that the known rebecs list of rebec r_i (as defined in Section 2) can be obtained by $K_i = \coprod_{j=[1..b_i]} L_{ij}$, where $\coprod$ represents concatenation of lists.

Definition 18 (Rotary permutation) For a given i and j, the rotary permutations acting on the members of L_{ij} are defined as $\psi_c(x) \doteq x +\% \ c$, where $1 \leq c \leq d_{ij}$.

Definition 19 (Preserving grouped KR relation) A permutation π, defined on the index set I, is said to preserve the *grouped known rebec relation* iff for every $i \in I$ and $1 \leq j \leq b_i$, we can find a rotary permutation ψ_c, such that $L_{\pi(i)j} = \pi(\psi_c(L_{ij}))$. For each L_{ij} the proper ψ_c is denoted as ψ_{ij}.

Note that preserving the known rebec relation is a necessary condition for preserving the *grouped* known rebec relation, and it becomes sufficient whenever for all i and j, L_{ij} is a singleton ($d_{ij} = 1$).

Definition 20 (Permutation on states - extension of Def. 10) For any permutation π defined on the index set I, together with a rotary permutation ψ_{ij} for each L_{ij}, the action of π on a global state s, denoted $\pi(s)$, is defined as follows. Suppose 'e' is a scalar variable that takes values from the scalar set associated to L_{ij}.

1. The value of a data state variable, say $r_i.v_{dg}[e]$, in state s, is assigned to the variable $r_{\pi(i)}.v_{dg}[\psi_{ij}(e)]$ in state $\pi(s)$.
2. Suppose the value of a rebec state variable, say $r_i.v_{rg}[e]$, in state s is x. In state $\pi(s)$, the value $\pi(x)$ is assigned to the variable $r_{\pi(i)}.v_{rg}[\psi_{ij}(e)]$.
3. If the value of a scalar state variable, say $r_i.v_{sg}$, in state s is y; in state $\pi(s)$, the value $\psi_{ij}(y)$ is assigned to the variable $r_{\pi(i)}.v_{sg}$.
4. Queue variables are treated in the same way as in Definition 10.

It is interesting to see that the soundness theorem (Theorem 1) presented in the previous subsection still holds (by taking the proper rotary permutations into account and considering the new definition of applying permutations on states). However, we need to rephrase the initial state theorem (Theorem 2) as follows.

Theorem 3 (Initial state theorem) $\pi(s_0) = s_0$ *if and only if* π *preserves rebec types and grouped known rebec relation and the parameters to the initial message servers of symmetric rebecs are symmetric.*

Proof Straightforward from the definition of initial state (Def. 5) and application of permutation on states (Def. 20).

Note that the same definition of symmetric parameters is applicable, because scalar variables cannot be passed as parameter to the initial message server. That is due to the fact that the scope of scalar variables is inside reactive classes.

Example 4 (Bridge Controller) In Figure 3, an abridged Rebeca model of the *Bridge Controller* system is shown. The controller needs a queue length of at least 6 to avoid queue overflow (cf. x_j defined in Section 2.2). This number is 3 for each train (declared in the header of the reactive classes).

The controller has a symmetric behavior towards its known rebecs (the trains). To exhibit the symmetric behavior of the `Controller`, its known rebecs (`trn`) are defined using a scalar set (`i`). The same scalar set is also used in declaring the state variables in this reactive-class. Intuitively, each `trn` is associated with one `signal` and `waiting`.

In the message server `Arrive` in `Controller`, the grouped known rebecs and variables are accessed inside a `forEachValueOf` block. In execution, the code in this block is repeated for the different values

```
reactiveclass Controller (6) {
  knownobjects {Train trn[i:1,2];}
  statevars
   {boolean signal[i],waiting[i];}
  ...
  msgsrv Arrive() {
    forEachValueOf (i) {
      if (sender == trn[i]) {
        if (! signal[i +% 1]) {
           signal[i] = true;
           trn[i].Pass(true);
        } else {
           waiting[i] = true;
} } } } }
```

```
reactiveclass Train (3) {
  knownobjects {
    Controller cntlr;
  }
  ...
  msgsrv Pass() {
    ...
} }

main {
  Controller ctrl(t1,t2):();
  Train t1(ctrl):();
  Train t2(ctrl):();
}
```

Fig. 3 Using scalar sets in Rebeca

of `i`, which can be either 1 or 2. This is like a *for* loop in programming languages. In addition to the loop behavior, it ensures the internal symmetry of the controller.

Recall that the permutation $\pi = (2, 1, 3)$ does not preserve the known rebec relation (and so Theorem 2 is not useful here). However, it does preserve the *grouped* known rebec relation. To check that, consider the grouped known rebec relation for the controller: $L_3 = (L_{31})$, where $L_{31} = (1, 2)$. We must find a proper rotary permutation ψ that satisfies: $L_{\pi(3)1} = \pi(\psi(L_{31}))$. It is easy to verify that $\psi = (2, 1)$ satisfies this condition. After checking the same property for other rebecs, Theorem 3 ensures that $\pi(s_0) = s_0$ and hence π is an automorphism of the model. □

3.4 Related Work on Symmetry

Scalar sets were introduced in [26] for use in a simplified version of Murphi. The idea was later adopted in other tools including Symmetric SPIN [6] and UPPAAL [17]. The idea of modules in SMC [39] is very similar to scalar sets. In these methods, the modeler should use scalar sets to exhibit the symmetry in the system. S/he must first identify the symmetric parts (processes) in the system. Then s/he uses scalar sets to highlight the symmetry explicitly in the model. The restrictions on the use of scalar sets disallow the modeler to code wrong symmetries. This, on the other hand, makes the modeling task difficult. The model checking tools use such information in the model to extract the symmetry. Traditionally, scalar sets can be used in models where all permutations are automorphisms (full symmetry); and therefore cannot be used for example for ring topology.

In inter-rebec symmetry, the symmetric parts (rebecs) in the system are detected automatically. Therefore, there is no need to illustrate them explicitly with scalar-sets. In addition, it is not restricted to full symmetry (in contrast to the traditional approaches based on scalar sets). In Section 5, we show how this can be done in polynomial time (with no input from the modeler). We have found no similar algorithm in the literature, as it has usually been the modeler's responsibility to code the symmetry (e.g., in [6, Section 6], it is stated: "A more ambitious attempt would be the automatic detection of scalar sets directly from the Promela sources.").

In intra-rebec symmetry, we propose using scalar sets, but only for specifying the symmetry in *one reactive class* (which is much simpler than the whole system), and hence all its instances. This extends our approach to systems, in which, part of the symmetry is due to the internal symmetry of some rebecs. This typically applies to the star topology, like the 'bridge controller' example. Section 6.2 is an example of other applicable topologies. Compared with the traditional uses of scalar sets, the error-prone task of specifying the symmetry by the modeler is highly reduced. Furthermore, the `+%` operation gives more flexibility in using scalar sets.

In [11], symmetry can be detected by examining *static communication channels*. They have applied their technique in [10] to find symmetry in simplified Promela models (e.g., excluding dynamic process creation). Similar to inter-rebec symmetry, their approach does not add symmetry-related constructs to the language syntax. However, to specify static channels properly, the modeler must follow certain guidelines (not specified in Promela grammar), whereas known rebecs are a built-in feature of Rebeca. This means that, unlike Rebeca, the modeler must be concerned about modeling the system appropriately for the symmetry detection technique (while not being specifically checked for the guidelines by the Promela parser).

Similarly, in models with internally symmetric processes (where intra-rebec symmetry is applicable), though no scalar sets are introduced in [11,10], the internally symmetric process should be carefully written in such a way that its symmetry can be detected. In fact, the modeler should implicitly adhere to similar restrictions (as those induced by scalar sets), so that the inherent symmetry can be detected.

Furthermore, we do not require the initial communication graph (induced by known rebecs) to be static, and we allow dynamic rebec creation. Finally, similar to our approach, they also use automorphisms on processes, from which the automorphisms on states are extracted. However, they suggest that, due to the small number of processes, standard graph isomorphism algorithms can be employed. Clearly, when the number of processes increases, such algorithms become impractical.

In Section 5, we propose a polynomial-time solution to the problem of finding automorphisms of rebecs in a given Rebeca model, based on the theory presented in this section (for both inter- and intra-rebec symmetry). The algorithm can be generalized for similar models of computation (see Section 5 for more explanation).

4 Partial Order Reduction

Partial order reduction is an efficient technique for reducing the state-space size when model checking concurrent systems for next-time-free linear-time temporal logic (LTL-X) [40,16,33]. In these systems, concurrency is modeled by interleaving. It is, however, not always necessary to consider all the possible interleaved sequences of these actions. The partial order reduction method suggests that at each state, the execution of some of the enabled actions can be postponed to a future state. It is therefore possible to avoid examining all the interleaved sequences of those transitions.

In this section, we explain an algorithm (from [22]) for exploiting *static* partial order reduction, which is considered to be the most practical variant of the partial order reduction techniques. It is shown in [22] that the reduced state-space generated by applying this algorithm satisfies the same LTL-X properties as the original state-space. This algorithm is also implemented in SPIN (a well known tool that exploits partial order reduction, and is used for model checking Promela models) [21]. The following definitions are taken from [22] and adapted to Rebeca; but these changes do not affect the proof given in [22]. In the definitions below, the set of all enabled actions at a given state s are shown by $\mathit{enabled}(s)$. Recall from Section 2.2 that action a_j is enabled in s, if the corresponding message is at the queue head of rebec r_j.

Definition 21 (Invisibility) An action that does not affect the satisfiability of the (propositions used in the) specification is called invisible. A transition labelled with an invisible action is also called invisible.

Definition 22 (Independence) The symmetric irreflexive relation $I \in A \times A$ on actions is said to be an independence relation iff for all $(a_i, b_j) \in I$ and for each s such that $\{a_i, b_j\} \subseteq \mathit{enabled}(s)$, the following conditions hold:

- if exists t_1 such that $s \xrightarrow{a_i} t_1$ then $b_j \in \mathit{enabled}(t_1)$.
- if for some $t_1, t_2, t'_1, t'_2 \in S$, $s \xrightarrow{a_i} t_1 \xrightarrow{b_j} t'_1$ and $s \xrightarrow{b_j} t_2 \xrightarrow{a_i} t'_2$ then $t'_1 = t'_2$.

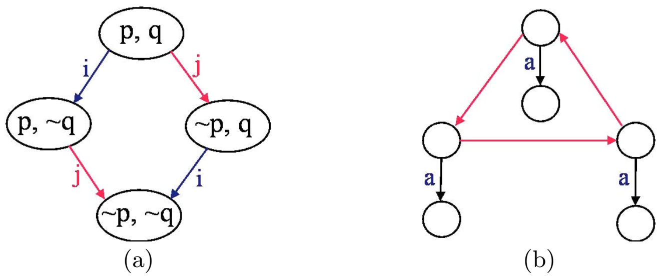


Fig. 4 (a) Independent actions; and, (b) The ignoring problem

Two transitions labelled with independent actions are called independent.

Intuitively, the independence of two actions a_i and b_j, means that whenever a_i and b_j are both enabled at a given state s, the execution of one of them cannot disable the other, and the consecutive execution of both, no matter which one is executed first, must result in the same state.

Definition 23 (Global independence) An action is called globally independent, if it is independent from all other actions of other rebecs. The transitions labelled with globally independent actions are called globally independent.

Definition 24 (Safety) An action is called safe if it is invisible and globally independent. All the transitions labeled with a safe action are also safe.

Assume that i and j are two actions, which occur in state space only as shown in Figure 4(a). If the atomic proposition q is not included in the specification, then i is invisible. Furthermore, i and j are independent, as they satisfy both conditions of Definition 22. As a more general case, an action that only changes the local variables of a rebec is globally independent.

To apply static partial order reduction, the safe actions must be known a priori, i.e., must be determined by static analysis of the model before staring the model checking. Intuitively, the execution of a safe action at a given state leads to a state where all other enabled actions (from other rebecs) remain enabled. So the execution of other enabled actions can be postponed to the future states. Therefore, at each state, we can define a subset of the enabled actions, called the *ample* set, which contains the minimum actions that need to be explored.

Note that, due to the presence of nondeterministic assignments, a single action execution may correspond to a nondeterministic choice among several transitions. In this case, the execution of one of these transitions leads to a state where the other transitions labeled by that action are disabled. Therefore, such transitions are interdependent and all of them must be expanded if the corresponding action is in ample set.

This strategy alone may cause some other enabled actions to be postponed forever. This so called *ignoring* problem may occur in the case of a loop, an example of which is shown in Figure 4(b). In this figure, the action 'a' may be ignored completely, if the other actions in the loop are safe. Generally, the solutions to the ignoring problem are called *provisos* (e.g., stack proviso [22], alternate proviso [32], etc.) whose need was first recognized by Valmari [40]. The stack proviso, implemented in SPIN, requires that the execution of none of the actions in the ample set should cut the DFS search stack (which definitely closes a loop). If no ample set satisfying the chosen proviso can be found, all the enabled actions from all rebecs must be explored. Such a state is said to be *fully expanded.*

4.1 Partial Order Reduction for Rebeca

In a Rebeca model, at each state, at most one action from each rebec is enabled (because there is at most one message at the queue head). This action may result in more than one transition (if it has nondeterministic assignments). Therefore, at a given state, the ample set can be the set of transitions due to the enabled action of a rebec, provided that: 1) the enabled action is safe; 2) some proviso, e.g., stack proviso, is also fulfilled. As explained in the previous subsection, the static partial order reduction technique requires that the safety of actions is determined statically. In what follows, we study the safety of different sub-actions (cf. Section 2) and actions in Rebeca.

Lemma 1 *Message removal is always safe.*

Proof The 'message removal' sub-action means shifting queue variables (of the current rebec) toward the queue head. This sub-action is invisible, because a specification is not allowed to use queue variables. It is also globally independent, because it doesn't change the (state or queue) variables of other rebecs. □

Lemma 2 *Assignments in the `initial` message server and assignments to variables that are not included in the specification are safe.*

Proof 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, if the variable on the left hand side of an assignment is not used in the requested specification, that assignment is also invisible, and hence safe. □

Lemma 3 *Sending a message from r_i to r_j is safe if r_i is the exclusive sender to r_j.*

Proof Sending a message can be considered as placing that message at the queue tail of the receiving rebec. Specifications are not allowed to use queue variables, so all 'send' operations are invisible. Therefore, safety of 'sends' depends on their being globally independent. But since each 'send' implicitly changes the queue tail, different 'sends' to the same queue are always interdependent. To obtain global independence (i.e., independence from actions from other rebecs), all 'sends' to a given queue must be performed by one rebec. This condition is achieved if r_i is the exclusive sender to r_j. In that case, 'sends' by r_i to r_j are globally independent and hence safe. □

Theorem 4 *If a message server is only composed of safe assignment and safe send statements, its corresponding action is safe.*

Proof The action corresponding to a message server includes an implicit message removal sub-action, which is shown to be safe. Therefore, the safety of an action depends on the safety of its explicit sub-actions. It is straightforward to see that an action composed of (the sequential execution of) only safe (explicit) sub-actions, is itself safe. □

4.1.1 Detecting Safe Actions Statically

According to Theorem 4, in order to determine the safety of message servers statically, we need to determine the safety of assignments and send statements. By lemma 2, the assignments in `initial` message server are always safe. This, for example, directly leads to the safety of the `initial` message server if it contains no 'send' statement. The safety of other assignments is also easy to determine by considering the desired property.

To guarantee the safety of send statements, we need to find those queues that are accessed (for write) only by one rebec. In order to find such queues statically, it is necessary that the model does not involve dynamic rebec creation nor dynamic change of topology. Both can be statically determined by making sure that rebec variables do not appear on the left hand side of assignments or as parameters of message servers. Otherwise, only the safety of assignments can be determined statically. As shown in Section 6, even in such cases, this can lead to considerable reductions.

In a model with no dynamic statements, the known-rebecs can be statically bound to real rebecs. Then by analyzing the send statements in the model, it is possible to find the rebecs that access each queue. Note, however, that the `initial` message is not considered in this analysis, because it is implicitly placed in the queue of all rebecs at the initial state and has no sender. As a result, it cannot disable or be disabled by other actions.

4.2 Related Work on Partial Order Reduction

Partial order reduction has been implemented in many model checking tools, including SPIN [21], PV [32], VIS [3] and COSPAN [30]. The approach in SPIN is the closest to ours, because it is based on explicit state enumeration and uses stack proviso. PV uses an alternate proviso, and VIS and COSPAN are based on implicit state exploration.

In SPIN, assignments to local variables are considered safe, while allowing the specification to use only global variables. Channel operations are safe if the executing process has the proper exclusive access (read or write) to the channel. The safety of a read/write depends on the used channel not being empty/full. This is called conditional safety. In Rebeca, read from channels (queues) is performed implicitly in every message server for removing the message at the queue head, which is always safe (cf. Lemma 1). In addition, unbounded queues in Rebeca eliminate the need for conditional safety (recall that we bound the queues, but in case of a queue overflow, the bound must be increased, resulting in a behavior identical to a system with unbounded queues).

There is also a tool for translating Rebeca to Promela [38]. In that tool, Promela processes represent rebecs, and channels are used to as a substitute to rebec queues. Since only global variables are allowed in the property specification, all rebec variables are 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 rather 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.

5 The Rebeca Model Checking Tool

The Rebeca model checking tool consists of two components: a translator and an engine. The ‘Model-checking Engine of Rebeca (Modere)’ is the component that performs the actual task of model checking. It is based on the automata-theoretic approach. In this approach, the system and the specification (of the negation of the desired property) are specified each with a Büchi automaton. The system satisfies the property when the language of the automaton generated by the synchronous product of these two automata is empty. Otherwise, the product automaton has an accepting cycle (a cycle containing at least one accepting state) that shows the undesired behavior of the system. In this case, an error trace witnessing a counter-example to the property is reported and the modeler can change the system model until the undesired behavior is eliminated.

Given a Rebeca model and some LTL specification (for the negation of the desired property), the translator component generates the automata for the system and the specification. These automata are represented by `C++` classes. The files containing these classes are placed automatically beside the engine (Modere). The whole package is then compiled to produce an executable for model checking the given Rebeca model.

5.1 Modere Structure

Modere has an object oriented architecture. In Modere, reactive classes are defined as `C++` classes and rebecs are their instances. Similarly, (the automaton of) the specification is represented by an instance of a class. Each rebec has a local hash table for storing its local states, where each local state is assigned a unique id number. All rebecs have an `execute()` method that, given (the id number of) a local state, returns (the id number of) the next local state. Each global state is the composition of the local states of all rebecs and the specification automaton, which are represented by their id numbers. This method, which is similar to the method used in SPIN, causes up to 60% reduction in storing the state-space [21], compared to storing all the information directly in global states. Interleaving in Modere is achieved by calling the `execute()` method of all enabled rebecs at each state.

Modere uses Nested Depth First Search (NDFS) [23] for computing the product automaton and performing model-checking on-the-fly. In this algorithm, the product automaton is generated using one DFS, which is at the same time used to find the accepting states. A second DFS is called in a post-order fashion, once for each accepting state as seed. It checks the reachability of the seed state from itself, which corresponds to an accepting cycle.

Modere uses a non-recursive implementation of NDFS, and handles the search stack manually. Furthermore, it does not dynamically allocate a memory block on heap, if it needs to release it afterwards; instead, temporary memory is allocated only on stack. As a result, fragmentation of memory is avoided.

5.2 Fairness in Modere

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, an algorithm based on Choueka's flag algorithm, using $n + 1$ copies of the state-space, is proposed in [4], where n is the number of the processes (rebecs) existing in the system.

In Modere, a slightly different version of this algorithm is used. The first DFS (in NDFS) is performed in the state-space zero. When starting the second DFS, the algorithm switches to the state-space 1. After that, whenever in state-space i, the execution of rebec i (or its being disabled) causes a transition to the state-space '$(i + 1)\%(n + 1)$'. If the seed state (at which the second DFS has started) is again visited in state-space zero, a fair accepting cycle has been detected. In practice, these $n + 1$ copies are implemented using only $n + 1$ extra bits for each stored state.

5.3 Exploiting Symmetry

In order to exploit symmetry, the automorphisms of the system need to be computed, *before* starting the model checking. In the following, we concentrate on the algorithm for finding the orbit relation and automorphisms in a given Rebeca model, based on the theory provided in Section 3. The algorithm is executed in the 'translator' component of Rebeca model checker.

To find the orbits, every pair of rebecs should be checked for equivalence. This results in $O(n^2)$ times performing the equivalence test. With the help of the ordering assumed on the known rebecs of each rebec, this test can be performed in polynomial time. Therefore, the whole algorithm for finding the orbits can be accomplished in polynomial time. We first explain the equivalence test algorithm for inter-rebec symmetry (Figure 5), and then extend it to support intra-rebec symmetry (Figure 6), as well. The complexity analysis is presented afterwards.

In the following, the names in `type-writer` font refer to variables in the algorithms. Consider the algorithm in Figure 5. In this algorithms, checking for equivalence of two rebecs, say r_i and r_j, is accomplished by finding a permutation π (=`pi`) that maps `i` to `j` (and hence r_i to r_j) while preserving

```
01  check (i, j) : boolean;
02    if (type[i] != type[j]) return false;// must preserve rebec types
03    pi[i] := j;                          // i is mapped to j
04    p1 := K(i); p2 := K(j);
05    while p1 not empty do
06       x := removeFirstElement(p1);
07       y := removeFirstElement(p2);
08       if (pi[x] is undefined)
09         pi[x] := y;
10         p1 += K(x); p2 += K(y);      // add to the end of the list
11       else if (pi[x] != y)  // known-rebec relation is not preserved
12         return false;
13    od
14    return true;
15  end
```

Fig. 5 Testing equivalence of two rebecs considering only inter-rebec symmetry

```
01  check (i, j) : boolean;
02    if (type[i] != type[j]) return false;
03    pi[i] := j;
04    p1 := L(i); p2 := L(j); // the grouped known rebec lists of i,j
05    while p1 not empty do
06       x := removeFirstElement(p1);
07       y := removeFirstElement(p2);
07.1     determine proper t   // see text for explanation
07.2     for m:=0 to lengthOf(x) do
08         if (pi[x[m]] is undefined)
09           pi[x[m]] := y[t+%m];  // add t to m modulo lengthOf(y)
10           p1 += L(x[m]); p2 += L(y[t+%m]);
11         else if (pi[x[m]] != y[t+%m])
12           return false;
12.1     od
13    od
14    return true;
15  end
```

Fig. 6 General equivalence test in Rebeca (inter- and intra-rebec symmetry)

the known rebec relation. In case of success, π is an automorphism of the system (soundness of this algorithm is ensured by Theorem 1).

First, π(`i`) is assumed to be `j`. In order to preserve known rebec relation, $\pi(K_i)$ must match K_j; i.e., applying π on the elements of `K(i)` (representing K_i) must result in the elements of `K(j)`. At each step, there are two list of rebec indices, represented by `p1` and `p2`, that must match via π, and are initialized at line 4 by `K(i)` and `K(j)`. Lines 6 to 12 ensure that the first element of `p1`, assigned to `x`, matches the first element of `p2`, assigned to `y`. When we have π(`x`) = `y`, their known rebecs must also match. This is ensured in line 10, by adding their known rebecs to `p1` and `p2`. The algorithm repeats the process and returns `true` if all the indices in `p1` and `p2` match. Otherwise, it returns `false`.

In order to add support for intra-rebec symmetry, we use grouped known rebec lists, i.e., $L(i)$ instead of $K(i)$ (lines 4 and 10 in Figure 6). As a result, each member of `p1` or `p2`, is itself a list of rebec indices (call them sublists). To match the indices within the sublists, we must find a proper rotary permutation ψ_t (c.f. Definition 19). The variable `t` is used to determine ψ_t at each step, and `t+%m` computes $\psi_t(m)$.

To ensure that a proper value for `t` is selected in Line 7.1, consider the sublists `x` and `y` in Figure 6 that must match via ψ_t. There are three possibilities. First, suppose π(`x`[u]) is already defined to be `y`[v] for some u and v (when this line is to be executed). In that case, `t` is set to $v - u$. Second, if

π is defined for no element of x, then t can be an arbitrary number (less than the size of x and y). In that case, the algorithm sets t to 0. Third, if for some u, π(x[u]) is defined, but is not in y, the algorithm fails and returns false. After determining t (in fact ψ_t), lines 7.2 to 12.1 make sure that other members of x are mapped to the proper indices from y (i.e., $\pi(x[m]) = y[\psi_t(m)]$. As before, the algorithm iterates over all elements of p1 and p2 to make sure that the grouped known rebec relation is preserved. Based on Theorems 1 and 3, once the algorithm returns true, the computed permutation π (together with computed proper rotary permutations) is an automorphism.

5.3.1 Complexity Analysis

In this part, we show that both algorithms presented in this section work in $O(n^2)$ (where n is the number of rebecs in the given model). In both algorithms, line 8 ensures that each element of π (=pi) is assigned at most once. Since π has n elements, lines 9 and 10 are executed $O(n)$ times. In the following paragraph, the text in parentheses applies only to the analysis of the second algorithm (for intra-rebec symmetry).

Consider the indices (in the sublists) of known-rebecs of x([m]) that are added to p1 in line 10. Once an index is added to p1, it is later checked exactly once in line 8 (although in line 6 a sublist of indices are removed together from p1). (Line 7.1 may also need to check each index in x at most $O(1)$ times for finding proper t) However, the number of indices added to p1 in line 10 is at most $n - 1$, which happens if x([m]) contains all other rebec indices. Therefore, lines 8 to 12 are at most executed $O(n * (n - 1)) = O(n^2)$ times. This means that the running time of the whole algorithm is $O(n^2)$.

As mentioned before, these algorithms may be called $O(n^2)$ times (in the translator component of the model checker, which means before the model checking itself) to compute the automorphisms. This means a running time of $O(n^4)$ for finding the automorphisms.

5.3.2 Discussion

Considering the raw idea of symmetry explained in Section 3.1, the methods presented here may miss some of the automorphisms on the states. In other words, these algorithms are sound (according to the theorems in Sections 3.2 and 3.3) but not complete. In fact, all other approaches to using symmetry reduction (cf. related work in Section 3.4) suffer from the same inaccuracy. That is what we lose, as we want to avoid the GI-complete (exponential time) analysis of the whole state space. However, intuitively, it is usually believed that the symmetry in the transition system is mainly due to symmetric processes/rebecs.

To the best of our knowledge, the algorithms given in this section, have no counterpart in the literature, as it has usually been the modeler's task to specify the symmetry explicitly, or general graph-isomorphism algorithms are employed (see related work in Section 3.4). In Rebeca, we can find the automorphisms on rebecs, because we have encapsulated objects that communicate only via asynchronous message passing. Having assumed an order on the lists of known rebecs, we can find automorphisms in polynomial time. Automorphisms on rebecs are then applied on states.We expect our techniques to be generalizable to other high-level models of concurrency of which encapsulation and asynchronous message passing form the basis.

For each pair of rebecs (r_i,r_j), if they are equivalent, the algorithm calculates a permutation (automorphism) that maps r_i to r_j. The automorphisms are stored and used during the model checking for computing the representative states. Whenever a state is reached during state exploration, by lexicographical analysis of the local states of different processes (similar to the approach in [6]), it is determined which automorphism to use to compute the representative state.

The current implementation of Modere is based on NDFS. Therefore, we adopted the method in [5] for exploiting symmetry. This method, under fairness assumptions, does not necessarily produce all the counter examples. Nevertheless, any result produced by this method *is* a counter example of the system. Therefore, the current implementation can still be useful for debugging very big systems that cannot be analyzed otherwise.

5.4 Exploiting Partial Order Reduction

Naive combination of static partial order reduction with NDFS confronts us with 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 extra bit per state to show whether each state is 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 visiting a state on the stack of the first DFS [23].

For using static partial order reduction, the safe actions of each reactive class are identified statically before Modere is executed (cf. Section 4.1), i.e., in the translator component. 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.

To employ partial order reduction, the first DFS in Modere is altered to use the `safety` array for finding a *safe* rebec. A rebec is safe if its enabled action is safe, and the states obtained by executing the transitions labeled by this action are not on the stack (stack proviso). To do so, a rebec with a safe enabled action is first chosen. Then the stack proviso is checked by executing each transition resulting from that action. If none of these transitions lead to a state on stack, the execution of all other rebecs is ignored (in other words, postponed to the succeeding states) – causing the reduction. If the stack proviso is not fulfilled, the execution of that rebec is terminated and the algorithm checks the safety of the next rebec. If no safe rebec exists, the state is marked as 'fully expanded', and all of the enabled rebecs are executed. In other words, no reduction is possible in that particular state.

The second DFS simply checks whether current state is already marked as 'fully expanded'. Note that every state visited by the second DFS is already visited by the first DFS (due to the post-order nature of NDFS). If the fully-expanded bit is not set, it means that the first DFS has explored only a subset of the enabled rebecs, or more precisely one of the safe rebecs. Since we cannot determine exactly which rebec was explored, we will need to execute all the safe rebecs (those with a safe enabled action). This ensures that the second DFS, at least, explores all the states visited by the first DFS. On the other hand, if the algorithm tries to visit a state that was not explored by the first DFS, it is automatically ignored, because the second DFS cannot store new states - avoiding exploration of unnecessary states.

5.5 Combining Partial Order and Symmetry Reduction

The combination of partial order reduction and symmetry reduction techniques was first studied by Emerson, et.al., in [13]. Partial order techniques exploit the independence of actions, while symmetry based reduction techniques are based on structural symmetries in the system. Based on this fact, [13] provides an abstract framework for combining these two techniques. For this purpose, partial order reduction can be applied on the quotient structure obtained by exploiting the symmetry in a model. To do so, we need to find an ample function that works on the quotient structure (ample function computes the ample set for a given state, cf. Section 4).

On the other hand, it is also possible to apply partial order reduction, while constructing the quotient structure on-the-fly. An algorithm that supports this method is proposed in [25], which is also demonstrated in Figure 7. In this algorithm, the ample function works on the original state graph of the system. In Modere, the quotient structure is constructed on-the-fly. Therefore, the latter approach is more appropriate. Since the ample function should work on the original state graph, the same approach as explained in the previous sub-section (the *safety* array) can be easily adopted for this method.

Practically, the `C++` code of Modere (generated for a given Rebeca model) contains the appropriate code for both symmetry and partial order reduction methods. The pieces of code related to these techniques are surrounded by proper compiler directives. To select each of these reduction methods,

```
DFS(s)
  add (rep(s), Statespace)
  push(rep(s), Stack)
  for each l in ample(s) do
    let t such that s $\xrightarrow{l}$ t
    if (rep(t) not in Statespace) then DFS(t)
  od
  pop (Stack)
end DFS
```

Fig. 7 Applying partial order while constructing the quotient structure [25]

	Time (s)	Memory	States
No Spec	1	27,520	46,882
No Starv.	38	45,048	251,871
Safety	2	27,520	46,882

(a) No reduction

	Time (s)	Memory	States
No Spec	1	25,628	27,240
No Starv.	24	37,544	150,255
Safety	1	25,628	27,240

(b) Partial order reduction

	Time (s)	Memory	States
No Spec	1	25,596	27,013
No Starv.	27	31,652	148,252
Safety	2	25,596	27,013

(c) Symmetry reduction

	Time (s)	Memory	States
No Spec	0	24,524	15,758
No Starv.	17	31,536	88,225
Safety	0	24,524	15,758

(d) Combined symmetry and PO

Fig. 8 Dining philosophers checked with Modere

the modeler can select to include the relevant code in compilation. This means that no decision about the reduction method should be made during run-time, which results in faster execution.

6 Empirical Results

6.1 Dining Philosophers

The dining philosophers problem was briefly introduced in Example 3 (Section 3.2). Two typical specifications can be checked for this model. The starvation freedom property means that each philosopher should always find a chance for eating (after a limited time for thinking). The safety property actually refers to the correctness of the model, and ensures that a `Fork` can't be picked up by two `Philosophers` simultaneously. In addition, the pure model (with no specification) can be fed to Modere to determine the size of the state-space.

The information in Table (a) (Figure 8) show that adding the starvation freedom property to the model increases the number of states, but the safety property leaves it unchanged (compared to using no specification). This is because the automaton representing safety has only two states, where the second state shows the undesired behavior. Since the model behaves correctly, the undesired state is never reached in this example. Therefore, it only increases the verification time, but not the number of states.

6.1.1 Symmetry

Binding rebecs in the same way as shown in Example 3 makes the model symmetric (inter-rebec symmetry). This symmetry can be detected by the algorithm of Section 5.3. Both algorithms in Figures 5 and 6 produce the same result. Applying the `check()` algorithm on every pair of rebecs, finds the following equivalence classes (orbits):

```
Orbit:
    phil0:0 phil2:5
Orbit:
    phil1:3 phil3:7
Orbit:
    fork0:2 fork2:4
Orbit:
    fork1:1 fork3:6
```

6.1.2 Partial Order

For applying partial order reduction, the safe actions (message servers) of the model must be identified. The only safe action in this model is the initial message server of Fork, which contains only assignments. As explained in Section 4, the assignments in the initial message server are assumed invisible, and hence the initial message server is safe.

In Figure 8, Table (b) shows that applying partial order reduction causes almost 45% reduction in the number of states. The result of applying symmetry is presented in Table (c), while Table (d) demonstrates the effect of combining partial order and symmetry reduction techniques. It can be seen that the combination of these techniques results in considerable reductions both in the number of states and the verification time.

6.2 Load Balancer

In this section, we model the ‘load balancer’ problem (from [11]). In this problem, there are six identical clients that need some specific service, which is provided by three identical servers. Instead of communicating directly with the servers, the clients send their requests to the load-balancer entities. The responsibility of a load-balancer is to distribute the incoming requests evenly among the servers. As a result, the servers receive an (almost) equal number of service requests. The servers, after finishing the requested service, reply directly to the clients. Only then, the clients may again ask for service.

In our model, the round-robin policy is used for load balancing, i.e., each load balancer forwards the incoming requests to the servers in a round-robin manner. In order to keep track of the originator of each request, the load balancers send to the server, the `sender` of each request, along with the request itself. Therefore, each server can determine the client, to which it should address the response. The complete Rebeca code for this example is demonstrated in Figure 10.

As explained in Section 2.2, the queue of each rebec is bounded to obtain a finite model. If a (bounded) queue is full, and it is to receive a new message, model checking is terminated with a ‘queue overflow’ message (plus an execution trace showing the path leading to queue overflow). We make use of this fact to evaluate the load balancing policy. Since there are six clients, there are at most 6 requests at a time. With an optimal load balancing policy each server would see at most 2 requests in its queue. In our model (with round-robin policy), we observe that only for queue sizes of greater than or equal to 7 for `Server`, no queue overflow is reported. It means that some server may receive 6 requests, while the `initial` message is still in its queue. We conclude that round-robin is not a proper policy for load balancing.

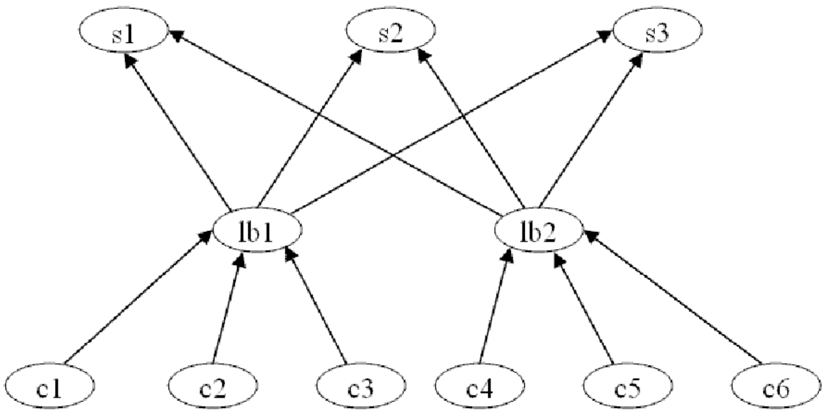


Fig. 9 Initial communication graph for the load-balancer example

```
reactiveclass LoadBalancer (4) {
  knownobjects {
    Server srv[scs:1,2,3];
  }
  statevars {
    scs srvNo;
  }
  msgsrv initial(){
    srvNo = ? {1, 2, 3};
  }
  msgsrv request(){
    srv[srvNo].service(sender);
    srvNo = srvNo +% 1;
  }
}

reactiveclass Server (7) {
  knownobjects {}
  statevars {}
  msgsrv initial() {}
  msgsrv service(Client rec){
      rec.serviceComplete();
  }
}
```

```
reactiveclass Client (2) {
  knownobjects {
    LoadBalancer lb;
  }
  statevars {}
  msgsrv initial(){
    self.requestService():
  }
  msgsrv requestService(){
    lb.request();
  }
  msgsrv serviceComplete(){
    self.requestService();
  }
}
main{
  Client c1(lb1):(), c4(lb2):(),
         c2(lb1):(), c5(lb2):(),
         c3(lb1):(), c6(lb2):();
  Server s1():(), s2():(),
         s3():();
  LoadBalancer lb1(s1,s2,s3):(),
               lb2(s1,s2,s3):();
}
```

Fig. 10 Rebeca code for the load-balancer example

6.2.1 Symmetry

Figure 9 shows the known rebec relation of this model. As shown here, the clients are not introduced as known-rebecs of the servers. Instead, the servers receive, as the parameter to the `service` method, the ID of the client to which they should address the reply. Furthermore, each load balancer knows all the three available servers. Due to the symmetric behavior of the load balancers, their known rebecs are introduced using a scalar set, thus allowing to use intra-rebec symmetry in this model.

The algorithm for inter-rebec symmetry (Figure 5) shows little symmetry in this model (only (`c1`,`c2`,`c3`) and (`c4`,`c5`,`c6`) form non-singleton equivalence classes). However, with the extended algorithm (Figure 6), we will observe that all instances of each type form one equivalence class, namely one orbit per reactive class. Note that this is neither ring nor star, and shows an example of other topologies where our algorithm can be applied.

	Time (s)	Memory	States
No Reduction	143	543,520	7,667,577
Partial order	103	394,628	5,223,939
Symmetry	123	485,524	5,898,883
PO and Symmetry	65	313,988	3,171,658

Fig. 11 Load Balancer checked with Modere

6.2.2 Partial Order

In this model, rebec variables are used as the parameter to the `service` method and hence the safety of 'send' statements cannot be determined statically. However, we can observe that the `initial` message servers of `LoadBalancer` and `Server` are safe (because they contain one assignment and no sub-action, respectively). This means two out of seven actions are safe.

As it is seen in Figure 11, even when only `initial` message servers are safe, there is still a considerable amount of reduction. The results in this table show the case that the upper bound on the queues of servers is 7. Furthermore, combining partial order and symmetry is again very effective in reducing the state space size.

7 Conclusions and Future Work

Rebeca is an actor-based language for modeling and verification of reactive systems where reactive objects can communicate only through asynchronous message passing. Dynamic rebec creation and dynamic change of topology are also possible with the so called *rebec variables*. In this paper, we addressed the problem of model checking Rebeca by applying symmetry and partial order reduction methods to Rebeca.

With that goal in mind, we gave a detailed account of the formal semantics of Rebeca. Then, based on the asynchrony in the model and the actor-based nature, we proposed an efficient approximate solution to the orbit problem for Rebeca. To the best of our knowledge, there is no counterpart in the literature for this approach to detecting symmetry. As a result the symmetry reduction technique can be efficiently applied to Rebeca.

For applying symmetry, we first proposed inter-rebec symmetry, which can be applied without making any changes to the syntax of Rebeca. The main advantage of this approach is that the modeler does not need to be concerned about the reduction techniques being used. This method detects the symmetry in topologies like ring, where the internal symmetry in rebecs is not important. If there is any rebec with internally symmetric behavior (including but not limited to star topology), we can make use of intra-rebec symmetry. We added scalar sets to the syntax of Rebeca to enable the modeler to show the symmetric behavior of these specific rebecs. This is still much simpler than the traditional approaches that require the modeler to exhibit the symmetry in the *whole* system. We proved the soundness of our algorithms.

We also applied static partial order reduction to Rebeca. Static partial order reduction, the most practical variant of the partial order reduction techniques, is based on detecting safe actions statically by overviewing the model. We showed how we can determine the safety of actions in a given Rebeca model. The special case of assignments in the 'initial' message server, which was also helpful in the presented case studies, makes partial order reduction a promising reduction technique for Rebeca. As shown in the practical results, the combination of partial order and symmetry can yield even better reductions in model checking Rebeca models.

The Model checking Engine of Rebeca (called Modere) uses the automata theoretic approach to model checking. Currently, specifications can be written in LTL and are automatically translated to

Büchi automata. Modere, besides symmetry and partial order reduction techniques, takes advantage of different tricks to reduce memory usage, for example by handling search stack manually.

Compositional verification is another technique, which is studied in [36] and is appropriate for Rebeca. In future, this technique can be added to Modere as well, and its combination with partial order and symmetry should be studied. CTL model checking for Rebeca is also being studied and can be added to Modere. More experiments with this tool in the future can demonstrate the power of model checking with Rebeca while applying partial order and symmetry reduction techniques (besides compositional verification).

We are also studying the generalization of our symmetry detection algorithms to Promela. We hope that we can improve upon the previous work on Promela (e.g., [6,10]) and provide the algorithms and tools for efficient automatic symmetry detection and reduction for Promela models, too.

A Proof of Theorem 1

The proof is given here for the more general case of intra-rebec symmetry. To obtain the simpler proof for inter-rebec symmetry only (without the notion of scalar sets involved), one can remove the parts related to scalar sets. We first prove some lemmas to make the proof of the theorem easier.

Lemma 4 *Given a permutation π, if a scalar expression evaluates to e in state s then it evaluates to $\psi(e)$ in $\pi(s)$, where ψ is the rotary permutation for the cluster type associated to the expression (cf. Def. 20).*

Proof Considering the definition of scalar expressions in Def. 16, there are three possibilities:

a. If the expression is simply a scalar variable, it can either be a scalar state variable or a scalar set used inside a `forEachValueOf` block. For scalar state variables, this is straightforward from Def. 20. In the case of a `forEachValueOf` block, since the block is executed once per each value of the scalar set, and the execution of different iterations are independent, we can map iteration e in a_j (in s) to iteration $\psi(e)$ in $a_{\pi(j)}$ (in $\pi(s)$).
b. A scalar expression can also be a scalar variable added to an integer z. The value z is based only on data variables, so it is the same value in s and $\pi(s)$. Suppose that $\psi(x)$ is defined as $x +\% c$ (cf. Def.18). If the scalar expression evaluates to $e = x +\% z$ in s, the same expression in $\pi(s)$ evaluates to $\psi(x) +\% z = x +\% c +\% z = \psi(x +\% z) = \psi(e)$.
c. Finally, a scalar expression can be a nondeterministic choice from all values of the scalar set denoting its type. We can again simply map the transition due to choosing e in state s, to the transition due to choosing $\psi(e)$ in $\pi(s)$.

Lemma 5 *Given a permutation π that preserves rebec types and some state s such that a transition $s \xrightarrow{a_j} t$ exists, a transition $\pi(s) \xrightarrow{a_{\pi(j)}} t'$ exists and the same sub-actions can be executed in a_j and $a_{\pi(j)}$.*

Proof The fact that action a_j is enabled at state s means that the message corresponding to action a is at the queue head of r_j, i.e., $r_j.m[1] = a$. According to Def. 10, in $\pi(s)$, we have $r_{\pi(j)}.m[1] = a$. This means that there is a transition $\pi(s) \xrightarrow{a_{\pi(j)}} t'$ in the model. Since π preserves the rebec types, a refers to the same message server in both r_j and $r_{\pi(j)}$.

To show that the same sub-actions will be executed in a_j and $a_{\pi(j)}$, we analyze *conditions* in Rebeca (which may be used inside message servers as guards for sub-actions). Conditions can be categorized as follows, based on the type of the variables used:

1. Data variables: Since data variables hold similar values in s and $\pi(s)$, any expression (including logical expressions and conditions), based only on data variables, also evaluates to the same value in s and $\pi(s)$.
2. Rebec variables: Rebec variables (including the `sender` keyword) are only allowed to be compared for (in)equality with other rebec variables. Recall that if a rebec variable holds the value x in s, in $\pi(s)$ the corresponding variable evaluates to $\pi(x)$. Since π is a bijective function (c.f. Definition 6), the (in)equality of two rebec variables is preserved by π.
3. Scalar variables: Scalar expressions can only be compared for (in)equality with other scalar expressions. Since by applying π on s, the values of scalar expressions are mapped using ψ, the (in)equality result remains unchanged.

As a result, conditions in a_j and $a_{\pi(j)}$ evaluate to the same values, which implies that the same sub-actions will be executed in these actions.

Theorem 1. *If a permutation π preserves rebec types and $\pi(s_0) = s_0$, then π is an automorphism of R.*

Proof Consider the definition of an automorphism in Rebeca (Def. 11). We have $\pi(s_0) = s_0$; therefore, we need to prove that if $s \xrightarrow{a_j} t$ is a transition in T, there exists a transition $\pi(s) \xrightarrow{a_{\pi(j)}} \pi(t)$ in T, too. Instead, we show that for every reachable state t, $\pi(t)$ is also reachable from s_0 (and the desired property of the theorem will also be proven). Throughout the proof, whenever we refer to a cluster (cf. Def. 15), we use ψ to denote the rotary permutation associated with that cluster (according to Def. 20).

We use an induction on the length of the shortest path from s_0 to s, denoted by $l(s)$. It is assumed among the hypotheses of the theorem that $\pi(s_0) = s_0$, thus, the basis of the induction follows. Now, assume that for any state q with $l(q) < k$, the induction hypothesis holds, that is $\pi(q)$ is reachable from s_0. To complete the proof, we have to show that for any reachable t with $l(t) = k$, $\pi(t)$ is also reachable.

Consider some state s with $l(s) = k - 1$, such that $s \xrightarrow{a_j} t$. Since $l(s) < k$, $\pi(s)$ is reachable. Based on Lemma 5, a transition $\pi(s) \xrightarrow{a_{\pi(j)}} t'$ exists, in which the same sub-actions as a_j are executed. In the following, by comparing the effect of executing the enabled sub-actions of a_j and $a_{\pi(j)}$, we demonstrate that the variables in t' hold the same values as in $\pi(t)$. The possible sub-actions to be considered are:

1. Message removal: The transition $s \xrightarrow{a_j} t$ always contains this sub-action, which results in:
   $\forall_{0<i<x_j} r_j.m[i] \leftarrow r_j.m[i+1]$, and $r_j.m[x_j] \leftarrow \bot$, and
   $\forall_{0<i<x_j} r_j.s[i] \leftarrow r_j.s[i+1]$, and $r_j.s[x_j] \leftarrow \bot$, and
   $\forall_{0<i<x_j, 0<k\leq h_j} r_j.p_k[i] \leftarrow r_j.p_k[i+1]$, and $r_j.p_k[x_j] \leftarrow \bot$.

   By Definition 10, it is deduced that the transition $\pi(s) \xrightarrow{a_{\pi(j)}} t'$ contains:
   $\forall_{0<i<x_{\pi(j)}} r_{\pi(j)}.m[i] \leftarrow r_{\pi(j)}.m[i+1]$, and $r_{\pi(j)}.m[x_{\pi(j)}] \leftarrow \bot$, and
   $\forall_{0<i<x_{\pi(j)}} r_{\pi(j)}.s[i] \leftarrow r_{\pi(j)}.s[i+1]$, and $r_{\pi(j)}.s[x_{\pi(j)}] \leftarrow \bot$, and
   $\forall_{0<i<x_{\pi(j)}, 0<k\leq h_{\pi(j)}} r_{\pi(j)}.p_k[i] \leftarrow r_{\pi(j)}.p_k[i+1]$, and $r_{\pi(j)}.p_k[x_{\pi(j)}] \leftarrow \bot$.
2. Note that according to Lemma 4, if a scalar expression evaluates to e in s, it evaluates to $\psi(e)$ in $\pi(s)$. There are three cases:
   (a) Assignment to a data variable (say $r_j.v_{di}[e]$): Assume that y represents the result of evaluating an expression, which is only based on data variables in state s. The same expression in $\pi(s)$ evaluates to the same value y (because data variables retain their values). Therefore, if a_j assigns y to $r_j.v_{di}[e]$ then $a_{\pi(j)}$ has a sub-action of the form $r_{\pi(j)}.v_{di}[\psi(e)] \leftarrow y$.
   (b) Assignment to a rebec variable ($r_j.v_{ri}[e]$): In this case, the right hand side can only be a rebec variable[4]. Assume that this variable contains the value x in state s. Therefore, such an assignment in a_j can be written as $r_j.v_{ri}[e] \leftarrow x$. Considering Definition 10, the same variable has the value $\pi(x)$ in the state $\pi(s)$. In other words, $a_{\pi(j)}$ has a sub-action of the form $r_{\pi(j)}.v_{ri}[\psi(e)] \leftarrow \pi(x)$.
   (c) Assignment to a scalar state variable: If it is assigned e by a_j, based on lemma 4, the value $\psi(e)$ is assigned to the variable by $a_{\pi(j)}$.
3. Send: Suppose that a_j contains a send sub-action, in which the message m with parameters $n_1, \ldots, n_{h_k}$ is sent from r_j to r_k. This necessitates that k is the value of a rebec variable, e.g. $r_j.v_g[e]$. According to Definition 20, $r_{\pi(j)}.v_g[\psi(e)]$ contains the value $\pi(k)$ in $\pi(s)$. Consequently, it is easy to see that the action $a_{\pi(j)}$ results in the message m with *symmetric* parameters being sent from $r_{\pi(j)}$ to $r_{\pi(k)}$. By symmetric parameters, it is meant that one of the following cases applies, based on the type of the parameter:
   (a) If the parameter is a data variable, the same value as in s is used.
   (b) If the parameter is a rebec variable, the permutation π is applied to the value in s.
   (c) If the parameter is a scalar expression, the proper rotary permutation ψ is applied to its value when applying π.

   Assuming that η_i represents the symmetric value corresponding to n_i, in t' we have:
   $\exists_{0<y\leq x_{\pi(k)}} (r_{\pi(k)}.m[y] = \bot \wedge \forall_{0<z<y} r_{\pi(k)}.m[z] \neq \bot)$; and,
   $r_{\pi(k)}.m[y] \leftarrow m, \quad r_{\pi(k)}.s[y] \leftarrow \pi(j), \quad \text{and} \quad \forall_{1\leq i\leq h_{\pi(k)}} r_{\pi(k)}.p_i[y] \leftarrow \eta_i.$
4. Rebec creation: If a_j results in the creation of a rebec with the new index v, the result of the execution of $a_{\pi(j)}$ would be the creation of a rebec with the new index, say w. We only need to extend π to include the pair (v, w), i.e. $\pi(v) = w$. Placing the `initial` message and its parameters at the queue head of the new rebec, is just the same as sending the `initial` message.

By analyzing the values of the variables in state t', it can be deduced that $t' = \pi(t)$. So, we have both shown that $\pi(t)$ exists, and $\pi(s) \xrightarrow{a_{\pi(j)}} \pi(t) \in T$. □

[4] It can also be the `sender` keyword, which represents the head of the sender queue, and is also a rebec variable.

References

1. Abdulla, P.A., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification (extended abstract). In: Hu and Vardi [24], pp. 379–390
2. Agha, G.: The structure and semantics of actor languages. In: Proc. the REX Workshop, pp. 1–59 (1990)
3. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: O. Grumberg (ed.) Proc. Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, *LNCS*, vol. 1254, pp. 340–351. Springer (1997)
4. Bosnacki, D.: Enhancing state space reduction techniques for model checking. Ph.D. thesis, Technische Universiteit Eindhoven (2001)
5. Bosnacki, D.: A light-weight algorithm for model checking with symmetry reduction and weak fairness. In: T. Ball, S.K. Rajamani (eds.) Proc. Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, *LNCS*, vol. 2648, pp. 89–103. Springer (2003)
6. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer (STTT) **4**(1), 92–106 (2002)
7. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Hu and Vardi [24], pp. 147–158
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
9. Clarke, E.M., Kurshan, R.P. (eds.): Proc. Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, *LNCS*, vol. 531. Springer (1990)
10. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) Proc. Formal Methods, International Symposium of Formal Methods Europe (FM '05), Newcastle, UK, July 18-22, *LNCS*, vol. 3582, pp. 481–496. Springer (2005)
11. Donaldson, A.F., Miller, A., Calder, M.: Finding symmetry in models of concurrent systems by static channel diagram analysis. Electr. Notes Theor. Comput. Sci. **128**(6), 161–177 (2005)
12. Emerson, E., Sistla, A.: Symmetry and model checking. Formal Methods in System Design **9**(1–2), 105–131 (1996)
13. Emerson, E.A., Jha, S., Peled, D.: Combining partial order and symmetry reductions. In: E. Brinksma (ed.) Proc. Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, *LNCS*, vol. 1217, pp. 19–34. Springer (1997)
14. Emerson, E.A., Sistla, A.P.: Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In: P. Wolper (ed.) Proc. Computer Aided Verification, 7th International Conference, Liege, Belgium, July, 3-5, *LNCS*, vol. 939, pp. 309–324. Springer (1995)
15. Emerson, E.A., Wahl, T.: On combining symmetry reduction and symbolic representation for efficient model checking. In: D. Geist, E. Tronci (eds.) Proc. the 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME '03, *LNCS*, vol. 2860, pp. 216–230. Springer (2003)
16. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke and Kurshan [9], pp. 176–185
17. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to UPPAAL. In: K.G. Larsen, P. Niebert (eds.) Proc. Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS '03, Marseille, France, September 6-7, *LNCS*, vol. 2791, pp. 46–59. Springer (2003)
18. Herstein, I.: Topics in Algebra. Xerox (1964)
19. Hewitt, C.: Procedural embedding of knowledge in planner. In: Proc. the 2nd International Joint Conference on Artificial Intelligence, pp. 167–184 (1971)
20. Hojjat, H., Nokhost, H., Sirjani, M.: Formal verification of the IEEE 802.1D spanning tree protocol using extended Rebeca. In: Proc. the First International Conference on Fundamentals of Software Engineering (FSEN'05), *ENTCS*, vol. 159, pp. 139–159. Elsevier (2006)
21. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
22. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: D. Hogrefe, S. Leue (eds.) Proc. the 7th IFIP WG6.1 International Conference on Formal Description Techniques, vol. 6, pp. 197–211. Chapman & Hall (1994)
23. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth first search. In: Proc. the Second SPIN Workshop, pp. 23–32. American Mathematical Society (1996)
24. Hu, A.J., Vardi, M.Y. (eds.): Proc. Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, *LNCS*, vol. 1427. Springer (1998)
25. Iosif, R.: Symmetry reduction criteria for software model checking. In: D. Bosnacki, S. Leue (eds.) Proc. Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, *LNCS*, vol. 2318, pp. 22–41. Springer (2002)
26. Ip, C., Dill, D.: Better verification through symmetry. Formal methods in system design **9**(1-2), 41–75 (1996)

27. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: H. Haddad (ed.) Proc. ACM Symposium on Applied Computing (SAC '06), Dijon, France, April 23-27, pp. 1810–1815. ACM (2006)
28. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: G. Chakraborty (ed.) Proc. Distributed Computing and Internet Technology, Second International Conference, ICDCIT '05, Bhubaneswar, India, December 22-24, *LNCS*, vol. 3816, pp. 494–507. Springer (2005)
29. Kakoee, M.R., Shojaei, H., Sirjani, M., Navabi, Z.: A new approach for design and verification of transaction level models. In: Proc. IEEE International Symposium on Circuit and Sytems (ISCAS '07) (2007). To appear.
30. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: B. Steffen (ed.) Proc. Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, *LNCS*, vol. 1384, pp. 345–357. Springer (1998)
31. McMillan, K.: Symbolic Model Checking. Kluwer Academic, Boston, MA, USA (1993)
32. Nalumasu, R., Gopalakrishnan, G.: An efficient partial order reduction algorithm with an alternate proviso implementation. Formal Methods in System Design **20**(3), 231–247 (2002)
33. Peled, D.: All from one, one for all: on model checking using representatives. In: C. Courcoubetis (ed.) CAV, *LNCS*, vol. 697, pp. 409–423. Springer (1993)
34. Shahriari, H.R., Makarem, M.S., Sirjani, M., Jalili, R., Movaghar, A.: Modeling and verification of complex network attacks using an actor-based language. In: Proc. the 11th Annual International CSI Computer Conference, pp. 152 – 158 (2006)
35. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamamenta Informaticae **63**(4), 385–410 (2004)
36. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Model checking, automated abstraction and compositional verification of Rebeca models. Journal of Universal Computer Science (JUCS) **11**(6), 1054–1082 (2005)
37. Sirjani, M., SeyedRazi, H., Movaghar, A., Jaghoori, M.M., Forghanizadeh, S., Mojdeh, M.: Model checking CSMA/CD protocol using an actor-based language. WSEAS Transactions on Circuit and Systems **3**(4), 1052–1057 (2004)
38. Sirjani, M., Shali, A., Jaghoori, M.M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: Proc. International Conference on Application of Concurrency to System Design (ACSD '04), 16-18 June, Hamilton, Canada, pp. 145–150. IEEE Computer Society (2004)
39. Sistla, A.P., Gyuris, V., Emerson, E.A.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Transactions on Software Engineering Methodology **9**(2), 133–166 (2000)
40. Valmari, A.: A stubborn attack on state explosion. In: Clarke and Kurshan [9], pp. 156–165