Symmetry and partial order reduction techniques in model checking Rebeca

Article *in* Acta Informatica · February 2010

DOI: 10.1007/s00236-009-0111-x · Source: DBLP

CITATIONS
32

READS
195

5 authors, including:

Mohammad Mahdi Jaghoori
Centrum Wiskunde & Informatica
30 PUBLICATIONS **360** CITATIONS
SEE PROFILE

Marjan Sirjani
Malardalen University
174 PUBLICATIONS **1,785** CITATIONS
SEE PROFILE

Mohammad Reza Mousavi
University of Leicester
168 PUBLICATIONS **1,457** CITATIONS
SEE PROFILE

Ehsan Khamespanah
University of Tehran
43 PUBLICATIONS **289** CITATIONS
SEE PROFILE

Acta Informatica manuscript No.
(will be inserted by the editor)

Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca

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

Received: date / Accepted: date

Abstract Rebeca is an actor-based language with formal semantics which is suitable for modeling concurrent and distributed systems and protocols. Due to its object model, partial order and symmetry detection and reduction techniques can be efficiently applied to dynamic Rebeca models. We present two approaches for detecting symmetry in Rebeca models: One that detects symmetry in the topology of inter-connections among objects and another one which exploits specific data structures to reflect internal symmetry in the internal structure of an object. The former approach is novel in that it does not require any input from the modeler and can deal with the dynamic changes of topology. This approach is potentially applicable to a wide range of modeling languages for distributed and reactive systems. We have also developed a model checking tool that implements all of the above-mentioned techniques. The evaluation results show significant improvements in model size and model-checking time.

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

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

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

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

E. Khamespanah
University of Tehran, Tehran, Iran
E-mail: e.khamespanah@ece.ut.ac.ir

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

1 Introduction

Rebeca [55] (reactive objects language) is an actor-based language [31,2] that can be used at a high level of abstraction for modeling concurrent and distributed reactive systems. Different interpretations, dialects and extensions of actor models are proposed in several domains and are claimed to be the suitable model of computation for the most dominating applications, like in embedded system, multi-core programming, and web services [32]. Actors have been proposed as models of computation for designing embedded systems [45] and wireless sensor networks [12]. They are also used for designing web services [11,10].

The asynchronous message-passing paradigm in Rebeca allows for efficient modeling of loosely-coupled distributed systems. Rebeca is designed to suit model checking [56, 58], which is an algorithmic approach to verification [14]. Due to concurrency, which is usually modeled by interleaving semantics, model checking is subject to state-space explosion, namely the exponential growth of the number of states with respect to the number of system components. Among the methods for overcoming this problem are symbolic verification [47], partial order reduction [60,27,50], and symmetry reduction [21,39,15,48]. These techniques are sometimes combined to gain even more compact representations of the system under analysis [1,22,24].

In this paper, we show how the asynchronous object model of Rebeca leads to an efficient application of symmetry and partial order reduction. Unlike most of the traditional notations of formal languages, Rebeca has a simple Java-like syntax that makes it easy to learn for software practitioners. Despite its simplicity, Rebeca is powerful enough to enable verification of distributed software systems and protocols that are asynchronous by nature, e.g., [57,33]. In addition, Rebeca has been successfully applied in model checking security protocols [54] and system-level hardware design [43,4].

Reactive objects (called *rebecs* in Rebeca) have message servers for processing the incoming messages. A rebec has a single thread of execution and thus resembles a process in a language like Promela [34]. The execution of a message server is performed in one atomic step, therefore, it corresponds to one action. Due to this coarse granularity in interleaving, partial order reduction can be very efficient. We described in [41] how to apply partial order reduction to Rebeca. In this paper, we formalize the correctness proof of the approach.

The state space of a system can be viewed as a graph: the states are the vertices and the transitions are the edges. The intuitive idea of the symmetry reduction technique [21,39,15,48] is to partition the state space into equivalence classes corresponding to isomorphic graphs and use one state as the representative of each class. Calculating the representative state, known as the *constructive orbit problem* is NP-hard [13]. The problem is usually alleviated by first detecting or specifying the symmetry among higher-level constructs (such as processes or objects) and then using it in the orbit problem. For example, the notion of scalar sets were proposed by Ip and Dill in [39], and later used by others (e.g., [7], [29]) to allow the modeler to (explicitly) specify the symmetry in the system.

We previously [42] gave a polynomial-time solution for detecting structural symmetry without relying on any symmetry-related input from the modeler. Since rebecs instantiated from the same reactive class exhibit similar behavior, inter-rebec symmetry in a Rebeca model can be detected by analyzing its communication structure (which is similar to the analysis method discovered independently in [19]). In this paper, we extend the approach by allowing the modeler to specify intra-object symmetries,

which can be added to the analysis of the communication structure. We give a proof of soundness for these techniques. We also give an efficient heuristic for computing a representative by combining our technique with the ideas of Bosnacki et al. [7]. Our techniques can in principle be adopted in other similar models of computation.

The *model checking engine of Rebeca* (Modere) was introduced in [41] to apply partial order reduction on Rebeca. We explain in this paper how symmetry reduction is added to it. 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 using these techniques separately can be useful in model checking Rebeca, but, whenever possible, their combination yields even more reduction due to their complementary nature.

In summary, this paper describes how partial order and symmetry reduction techniques can be applied in model checking Rebeca, summarizing, integrating and extending the results of [41,42]. In the following, we illustrate the complete work, while mentioning the contributions of this paper:

- The main contribution of our work is a symmetry detection technique based on the object-based nature of Rebeca. A polynomial time algorithm for inter-rebec structural symmetry detection is proposed in [42]. To the best of our knowledge, it is novel in the sense that it needs no symmetry-related input from the modeler. A similar approach (without support for dynamic process creation) has been discovered independently and reported in [17]. In this paper, we extend the method to incorporate intra-rebec symmetries, which enables the automatic detection of structural symmetry in more topologies. Additionally, we show how to apply these techniques in computing the representative of a given state while model checking.
- To establish the formal proofs, we gave in [42] a refined formal semantics of Rebeca (compared to [55]), in which dynamic rebec creation and dynamic change of topology are neatly handled by introducing rebec variables (variables holding rebec identifiers). In this paper, we slightly improved the presentation of this refined semantics.
- An algorithm for applying static partial order reduction to model checking Rebeca is proposed in [41]. In this paper, we formalize the correctness proof of the algorithm.
- 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 [41], 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.

1.1 Related Work

Symmetry

One of the most widely used techniques for symmetry detection has been using scalarsets to explicitly specify the symmetry among the processes (see, e.g., [39,40]). We refer to [48], for a comprehensive survey of various other symmetry detection and reduction techniques. In our inter-rebec symmetry detection, we automatically extract

symmetric structures in the communication among rebecs, without any symmetry-related constructs like scalar sets. The closest work to ours is that of Donaldson, Miller and co-authors, who independently from our work, have proposed several techniques for detecting symmetry in models of computation (mainly Promela) and reducing the state space using the detected symmetries [8, 17–20].

In [17] the (initial) communication topology of a Promela code is abstracted into a *static channel diagram*. Symmetries present in the static channel diagram are further exploited to generate a maximal sub-group of valid symmetries for the Promela code. This approach resembles our inter-rebec symmetry reduction technique. Bosnacki et al. [8] introduce scalar sets (à la [40]) into Promela in order to capture the global symmetric structure of Promela processes. In [46], the same technique is applied to the B method [53]. Donaldson et al. [20] suggest a generalization of scalar sets for detecting and exploiting non-full symmetries of the topology of communication channels.

To detect symmetries in B, Leuschel and Massart [46] do not extend the syntax (as in inter-rebec symmetry) because the *deferred sets* construct gives rise to symmetric data values similar to scalar sets. This is in itself advantageous over the techniques proposed before. However, the modeler is still involved in the process, as the detected symmetry depends on the proper use of deferred sets.

Similar to our inter-rebec symmetry detection, Donaldson et al. [17] can detect symmetry automatically without depending on particular syntactic notations in the language (scalar sets, deferred sets or the like). Moreover, arbitrary structural symmetries can be captured and thus, the approach is not restricted to full- or ring-based symmetries. The advantage of our inter-rebec symmetry detection compared to those reported above is that our approach works for a model of computation with dynamic creation of processes. Furthermore, we extend this approach to consider intra-rebec symmetries.

In intra-rebec symmetry, we propose to use scalar-sets but for a different purpose from its usual use. Our use of scalar sets is in line with the modular modeling encouraged by the actor model. We use scalar sets locally for each class to specify the symmetric behavior of that class with respect to its known rebecs (when applicable), rather than specifying the symmetry in the whole system. This is an extension of inter-rebec symmetry allowing us to consider the internal symmetry of rebecs along with their communication structure. Similar to [40, Section 3.8], we allow modulo arithmetic operations on the scalar sets and thus the rebec does not need to have a fully symmetric internal behavior.

Our symmetry reduction method, i.e., computing a representative state during model checking, is inspired by the techniques introduced in [39] and further explored in [6, 7]. We use a variant of the sorted heuristic in [7] to work around the orbit problem.

In order to model check temporal logic formulae on the quotient structure, one needs to make sure that the formula under consideration is also invariant under the applied automorphisms. We assume that our formulae are constructed from symmetric sub-formulae. An interesting future direction is to investigate this problem further, along the lines of [52], where the symmetry in the temporal logic formulae is exploited in order to detect more (partial) symmetries in the system. The generalized symmetry groups as proposed in [52], use the structural symmetries in the temporal formulae, in order to detect symmetries that are otherwise impossible to detect since symmetric components satisfy different (yet symmetric) propositions.

Partial Order Reduction

Many model checking tools take advantage of partial order reduction. The approach in SPIN [34] is the closest to ours, because it is based on explicit state enumeration and uses stack proviso. PV [49] uses an alternate proviso, and VIS [3] and COSPAN [44] are based on implicit state exploration. Our approach is based on statically finding safe actions. The idea is that a safe action can be executed before other enabled actions without being interleaved (see Section 4 for details).

In SPIN, assignments to local variables are 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 (resp. write) depends on the used channel not being empty (resp. 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 5 in the Appendix). In addition, unbounded queues in Rebeca eliminate the need for conditional safety (as explained in Section 2, queues are bounded in the actual implementation, 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 [58]. In that tool, Promela processes represent rebecs, and channels are used to substitute the 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. This code can be fed to SPIN to be model checked. However, using global variables renders all assignments unsafe. 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. More importantly, SPIN cannot handle the special case of the 'initial' message server (cf. Section 4) which can in turn produce much reduction (see experimental results in Section 6).

Paper Structure. In the next Section, we explain the syntax and refined semantics of Rebeca. Section 3 introduces symmetry reduction and in Section 4, partial order reduction is described. The details of applying symmetry and partial order reduction techniques to Rebeca are also given. In Section 5, we elaborate on the model checking engine of Rebeca (Modere) and how we incorporate the reduction techniques in it. 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 [58,55,56] is a modeling language with formal semantics based on an operational interpretation of the actor model [31,2]. A Rebeca model is a closed system defined as the parallel composition of a set of concurrent rebecs (reactive objects), written as $R = \|_{i \in I}\ r_i$, where I is the set of the indices used for identifying each rebec. Rebeca inherits from actor languages the dynamically changing topology and dynamic creation of objects; thus, the number of rebecs, and hence I, may change in the course

CL ::= **reactiveclass** $C(Nat)$ $\{KRs\ Vars\ Mtd^*\}$
KRs ::= **knownrebecs** $\{\ \langle Vdcl; \rangle^*\ \}$
$Vars$ ::= **statevars** $\{\ \langle Vdcl; \rangle^*\ \}$
$Vdcl$ ::= $T\ \langle v \rangle_{,}^{+}$
Mtd ::= **msgsrv** $M(\langle T\ v \rangle_{,}^{*})\ \{St^*\}$

$Call$ ::= $v.M$ | **self**.M | **sender**.M
St ::= $v = e$;
| $v =$ **new** $C(\langle e \rangle_{,}^{*})$;
| $Call(\langle e \rangle_{,}^{*})$;
| **if** (e) $\{St^*\}$ $\langle$ **else** $\{St^*\}$ $\rangle^{?}$

Fig. 1 BNF grammar for Rebeca classes. Angle brackets $\langle ... \rangle$ are used as meta parentheses, superscript ? for optional parts, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using $\langle ... \rangle_{,}$ with repetition denotes a comma separated list. Identifiers C, T, M and v denote class, type, method and variable names, respectively; Nat denotes a natural number; and, e denotes an (arithmetic, boolean or nondeterministic choice) expression.

of execution. 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 *only* through asynchronous message passing and have unbounded buffers for automatically storing the incoming messages, i.e., there is no statement in Rebeca syntax to explicitly wait for receiving a message. When a rebec is scheduled to run, the message at the head of the queue is taken out and processed. Each message that can be serviced by rebec r_i has a corresponding message server, which is given in the definition of the reactive class. 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 (like 'constructors' in object oriented programming languages). Each rebec receives this message implicitly upon creation.

Rebecs have local variables making up their states but there are no shared variables. 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 the known rebecs must be provided upon creation (this is enforced by the syntax). We use K_i to denote the list of the initial values of the known rebecs of r_i. Instead of using [name,value] pairs for each known rebec, we use an ordered list to keep the values only, in the same order as they appear in the reactive class definition. 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 the 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. Although each rebec has one queue, we can model multiple reception queues between different rebecs by adding extra rebecs representing each queue. Figure 1 shows the syntax for defining classes in Rebeca, the details of which is clarified in the following example.

```
reactiveclass Fork(3) {
   knownrebecs {
     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) {
   knownrebecs {
     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. 2 Rebeca code for the dining philosophers problem

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 2 shows the Rebeca code of this example.

The parameter '3' passed to the reactive classes denotes the upper bound on the queue length, provided by the modeler. This upper bound is used to avoid infinitely large states due to unbounded queues. The model checker will produce a proper message if queue overflow occurs. Then the modeler needs to increase this upper bound. 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 deadlock and consequently starvation, for every other philosopher we bind the left and right forks in the reverse order. For example, in Figure 2, both `phil0` and `phil1` try to take `fork0` first, and only the one who succeeds will try to take its second fork.

2.1 The Formal Semantics of Rebeca

In [55], 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.

An LTS consists of states and the transitions between them. Each transition is labelled by an action, which corresponds to execution of a message server from one rebec. Since instances of similar reactive classes have similar actions, actions of each rebec are indexed by the identifier of that rebec. We may write $s \xrightarrow{a_i} t$ for a transition from s to t labeled by action a_i. In the case of a nondeterministic choice in an expression, it is possible to have two or more transitions with the same action from a given state (a nondeterministic choice of the form `?(d1,...,dn)` chooses an arbitrary value from the list `(d1,...,dn)` to be used in the corresponding expression). A formal definition of LTS for Rebeca is given at the end of this section.

Definition 1 (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 2 (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 equality;
- used to specify the receiver of a send statement; or,
- assigned (the index of) a dynamically created rebec.

Rebec variables can be passed around as arguments to messages, resulting in a dynamic topology. As mentioned in the previous subsection, some of these variables are statically initialized as the known-rebecs, reflecting the initial configuration of the rebecs.

Definition 3 (Queue) 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 write $r_j.Q$ to refer to the queue as a whole. Note that the first element in queue has index 1.

The domain of message queue variables is $M_j \cup \perp$, where M_j is the set of the names of message servers defined in r_j and $\perp$ 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.

In the semantics, we assume unbounded queues for rebecs, however, in order to make model checking feasible, we need to put an upper bound on the queue of a rebec. This upper-bound is in practice supplied by the modeler and must be increased in case of a queue overflow. Thus, it can simulate the unbounded queue semantics of Rebeca.

Definition 4 (Global State) Given a set I_s of live rebec indices, a global state is defined as the combination of the local states of all rebecs: $s = \prod_{j \in I_s} s_j$. Note that I_s may change in the course of transitions due to dynamic rebec creation.[1]
The local state of a rebec r_j consists of its local (data and rebec) variables plus the queue, formally written as: $s_j = (r_j.v_1, \ldots, r_j.v_{w_j}, r_j.Q)$, where $w_j \geq 0$ shows the number of the 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*'; $r_j.s[1]$ is set to j; the variables corresponding to the known rebecs are initialized according to the model; and, if the *initial* message server of r_j accepts n_j $(n_j \leq h_j)$ parameters other than the known rebecs, the variables $r_j.p_1[1]$, $r_j.p_2[1]$, $\ldots$, $r_j.p_{n_j}[1]$ are also initialized accordingly. All other (local and queue) variables are assigned the value $\perp$.

Definition 6 (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 (cf. Definition 4).
- s_0 is the *initial state*.
- A denotes the set of *actions* (message servers) of different reactive classes.
- The *transition relation* $T \subseteq S \times A \times S$ is defined as follows.

We write a_j for action a in rebec r_j. There is a transition $s \xrightarrow{a_j} t$ in the system, iff the action a_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 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 the values of variables in s, and is assigned to the variable on the left-hand side, in state t.

1. *Message removal*: This is an *implicit* sub-action that exists in all actions. It includes removing the first element of message, sender and parameter queues, plus shifting other elements of the queues. This sub-action automatically happens as the last sub-action of any action. Note that parameters are available in the message server by using their names (as in Java).

[1] The subscript s of I_s may be omitted when no ambiguity arises.

$\forall_{i>0}$. $r_j.m[i] \leftarrow r_j.m[i+1]$
$\forall_{i>0}$. $r_j.s[i] \leftarrow r_j.s[i+1]$
$\forall_{i>0}$. $\forall_{0<k\leq h_j}$. $r_j.p_k[i] \leftarrow r_j.p_k[i+1]$

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 (which may include normal arithmetic on integers and boolean operations on booleans) 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); for instance, d cannot be a literal to represent an explicit rebec index. As a result of this assignment, the value of d is assigned to w in state t.
3. *Rebec creation*: This statement has the form '*new* $rc(kr_1, \ldots, kr_m) : (p_1, \ldots, p_z)$' 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 means that $I_t \leftarrow I_s \cup \{v\}$ in state t. 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 $p_1, \ldots, p_z$ are placed in $r_v.p_1[1], \ldots, r_v.p_z[1]$, 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 the rebec 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: for y such that $r_k.m[y] = \perp \wedge \forall_{0<u<y} r_k.m[u] \neq \perp$, we have the following:

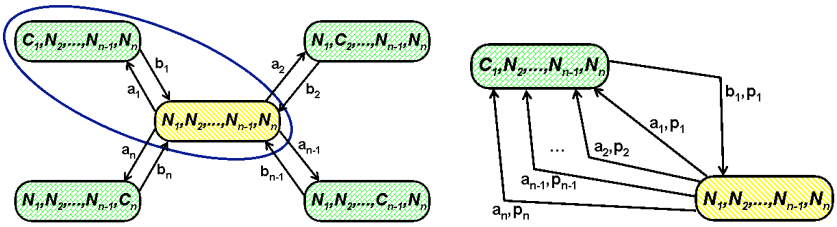
   $r_k.m[y] \leftarrow m$,
   $r_k.s[y] \leftarrow j$,
   $\forall_{1\leq i\leq z} r_k.p_i[y] \leftarrow n_i$ (other elements keep their $\perp$ value).

3 The Symmetry Reduction Technique

The state space of a system can be viewed as a graph: the states are the vertices and the transitions are the edges. The intuitive idea of the symmetry reduction technique [21,39,15,48] is to partition the state space into equivalence classes corresponding to isomorphic graphs and use one state as the representative of each class. In this section, we explain formally how to use symmetry reduction in model checking.

3.1 Preliminaries

Consider a system M, consisting of n concurrently executing processes, represented by a labeled transition system $M = \langle S, A, T, s_0 \rangle$. A global state $s \in S$ is composed of local states of the processes: $s = \prod_{i \in I} s_i$ where I is the set of process indices. A permutation $\pi : I \rightarrow I$ is defined as a bijection (1-1 and onto function) on the set I of process indices, but it can be lifted to global states and actions in a straightforward way by permuting the indices of the local states and actions. A formal definition of



(a) Selecting representatives (b) Annotated quotient structure

Fig. 3 An example of a symmetric state-space

this lifting for Rebeca is given in the next subsection. The set of all permutations on I is shown by $SymI$. In the following, we use π to denote the lifting of the same permutation onto states and actions.

Example 2 (taken from [21] with minor changes) Consider a system 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 3-a shows the state-space of this system. Consider the index set $I = [1..n]$, and the permutations defined in I. Henceforth, we write a permutation π as an ordered list, the i'th element of which denotes $\pi(i)$. For example, if $p = /2, 1, 3, \ldots, n-1, n/$, then $p(1) = 2$, $p(2) = 1$ and other indices are mapped to themselves. This permutation p maps $\{C_1, N_2, \ldots, N_n\}$ to $\{N_1, C_2, \ldots, N_n\}$, while applying any permutation to the state $\{N_1, N_2, \ldots, N_n\}$ results in the same state.

Definition 7 (Automorphism [21,39]) A permutation π is said to preserve the transition relation when $s \xrightarrow{a} t \in T$ implies $\pi(s) \xrightarrow{\pi(a)} \pi(t) \in T$. 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 $SymI$ (and similarly $AutM$) forms a group [30] with respect to functional composition, i.e., it includes the identity permutation and is closed under composition and the inverse operation. Any subgroup G of $AutM$ induces an equivalence relation on the states, such that two states s and s' are equivalent iff $\exists_{\pi \in G} \pi(s) = s'$. The resulting equivalence classes are called *orbits*. For example in Figure 3-a, every permutation on the index set is an automorphism of M, i.e., $AutM = SymI$. With respect to $SymI$, one orbit contains $\{N_1, N_2, \ldots, N_n\}$, 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 3-b).

Definition 8 (AQS [21]) Given a subgroup G of $AutM$, the *annotated quotient structure (AQS)* for M with respect to G 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 an AQS of 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 us 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 the states.

In Figure 3-a the (arbitrarily) chosen representatives of its two orbits are distinguished with a line around them. Figure 3-b shows the 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.[2] Emerson et al. [21] show that using $\overline{M_G}$ instead of M is enough in the automata theoretic approach to model checking. This approach is extended in [23] for model checking under fairness conditions. Bosnacki in [5] shows how symmetry reduction can be employed in *Nested Depth-First Search* (NDFS) [36]. To apply these methods, the representative for each state should be determined while exploring the state-space, known as the *constructive orbit problem*, which is NP-Hard [21,13]. In the following, we show how we can detect symmetries in a Rebeca model efficiently. Exploiting the detected symmetries in order to efficiently construct the quotient structure, i.e., a polynomial-time heuristic solution to the constructive orbit problem, is described in Section 5.1.

3.2 Inter-rebec Symmetry in Rebeca

Recall the dining philosophers problem in Example 1. 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 this intuition, r_0 (phil0) and r_2 (phil2) should have symmetric behaviors. In other words, one automorphism should be a permutation π, where $\pi(0) = 2$ and $\pi(2) = 0$. In general, a Rebeca system is symmetric if the communication pattern among rebecs, i.e., the binding of the known-rebecs, is symmetric. In the following, we define this formally.

3.2.1 Formal Definitions

Consider a system $R = \langle S, A, T, s_0 \rangle = ||_{i \in I}\, r_i$ of a Rebeca model, where $I = [1..n]$ is the index set of the rebecs. To exploit symmetry, we use the permutations acting on I. Since rebecs of the same type (i.e., instances of the same reactive-class) consist of the same message servers, they exhibit similar behavior. We limit the permutations to those preserving rebec types.

Definition 9 (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.

[2] We do not address the problem of detecting symmetry in formulas. One solution is to use *indexed temporal logics* [21] or to detect the symmetries in temporal logic formulas [52].

Next, we define formally how to lift a permutation π defined on the index set I to be applied on states and actions. For an action $a_i \in A$, we define $\pi(a_i) = a_{\pi(i)}$. Obviously, $a_{\pi(i)} \in A$ if π preserves the rebec types.

Definition 10 (Permutation on states) 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 (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., $\pi(K_i) = (\pi(t_1), \pi(t_2), \ldots, \pi(t_{P_i}))$.

Lemma 1 *Given a permutation π, $\pi(s_0) = s_0$ if π preserves rebec types and known rebec relation and the parameters to the initial message servers are symmetric, i.e., if $\pi(i) = j$, and p is a parameter to the initial message server of r_i, the corresponding parameter for r_j should*

- *have the same value as p, if p is a data variable; or,*
- *be equal to $\pi(p)$, if p is a rebec variable.*

The proof is straightforward from the definition of initial state (Def. 5) and the application of permutations on states given above (Def. 10).

Theorem 1 (Soundness) *If a permutation π preserves rebec types and $\pi(s_0) = s_0$, then π is an automorphism of R (cf. Definition 7).*

Theorem 2 *The set of all permutations satisfying Theorem 1 form a group.*

The proofs of these theorems are given in the Appendix. As an example of using Theorem 1, we check if the permutation $\pi = /2, 3, 0, 1, 6, 7, 4, 5/$ is an automorphism of the dining philosophers model as was discussed at the beginning of this subsection. Notice that π satisfies the intuitive condition we expect, i.e., $\pi(0) = 2$ and $\pi(2) = 0$. First of all, 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. Lemma 1). 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.

In Section 5.1, we present an algorithm for checking whether a permutation preserves rebec types and known-rebec relation; and we show how to use the group of these permutations (cf. Theorem 2) for computing a representative during symmetry

KRs ::= **knownrebecs** { $\langle KRdcl; \rangle^*$ }
$KRdcl$ ::= $Vdcl$ | $T\ \langle v\ [i : Nat..Nat] \rangle^+_,$
$Vdcl$ ::= $T\ \langle vGr \rangle^+_,$
vGr ::= $v\ \langle [i] \rangle^?$

$Call$ ::= $vGr.M$ | **self**.M | **sender**.M
St ::= $vGr = e;$
| $vGr =$ **new** $C(\langle e \rangle^*_,);$
| **forEachValueOf** (i) $\{St^*\}$

Fig. 4 Extended syntax including scalar sets. The declarations here include only what is different or added to Figure 1. Expressions (denoted by e) can now include scalar expressions. An identifier i denotes a scalar set.

reduction. 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. This relieves the modeler from the task of using symmetry-related constructs (e.g., scalar-sets) in order to exhibit the automorphisms (see related work in Section 1.1).

3.3 Intra-rebec Symmetry in Rebeca

In systems with loosely coupled objects, symmetry is usually caused by the symmetric composition of objects, thus, inter-rebec symmetry 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 the internal structure of some objects also needs to be considered in order to reveal the symmetric composition in the system, e.g., of the object in the center of a *star* network. 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.

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. In this model, 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 provides mutual exclusion and schedules the trains such that starvation is avoided. For the controller, the trains have equal priorities and are treated equally.

Suppose we do not consider the internal behavior of the rebecs, and only rely on the communication structure (as required by Theorem 1). 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 intuitively expect $\pi = /2, 1, 3/$, which swaps `t1` and `t2`, to be an automorphism. However, π does not preserve the known rebec relation, because: $\pi(K_3) = (\pi(1), \pi(2)) = (2, 1) \neq K_{\pi(3)}$. We extend inter-rebec symmetry to allow swapping the order of the known rebecs when the rebec has symmetric internal behavior.

3.3.2 Scalar Sets in Rebeca

To enable the modeler to exhibit internal symmetries in rebecs, we add *scalar sets* [39] to the syntax of Rebeca (see Figure 4). An abridged Rebeca model of the *Bridge Controller* system using scalar sets is shown in Figure 5. We use this as the running example in this section.

```
reactiveclass Controller (6) {
  knownrebecs {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();
        } else {
           waiting[i] = true;
} } } } }
```

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

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

Fig. 5 Using scalar sets in Rebeca

Scalar Sets. A scalar set is a set of consecutive *scalar values*. Scalar values are natural numbers, on which only the operations for adding modulo the size of a set (+%) and checking equality and non-equality are allowed. 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 is a non-scalar integer.

Grouped Known-rebecs. 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 group them using scalar sets. The model checker can then consider this internal symmetry. New scalar sets can be introduced only when declaring grouped known-rebecs. For example, in Figure 5, `Controller` declares the scalar set `i` with the range [1,2] along with the array-like definition of two known rebecs of type `Train`. This also helps the modeler avoid repeating the symmetric part of the code in that reactive class (explained in the sequel). The syntactic constraints on using scalarsets ensure that the containing reactive class (e.g., `Controller`) has symmetric behavior towards these known rebecs.

Variables. The scalar sets defined for grouping known rebecs can be used as the data type for defining other variables, called *scalar variables*; or, again in an 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 the controller are examples of grouped variables. Scalar variables are distinguished by a subscript 's' when necessary.

We define a *cluster* as the set of grouped known rebecs, grouped state variables, and scalar variables that are defined using the same scalar set. The scalar set identifying each cluster denotes the type of that cluster. In Figure 5, `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. A cluster is a logical concept only used in the proofs.

Scalar Expressions. 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 (using +%) 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 Figure 5, '`i +% 1`' is a scalar expression (of type `i`) that is used for indexing `signal`.

For simpler manipulation of grouped known rebecs and variables, we define the construct `forEachValueOf` that associates a number of statements with a scalar set. 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). 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 of `i`, which can be 1 or 2.

The behavior of these blocks is similar to the *for* loop construct in programming languages such as C and Java, or similar to the `ALL` construct in SMC [59]. 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 [39] 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, because `trn[1]` will be called both when `i` is 1 and 2.

3.3.3 Using Scalar Sets for Detecting Symmetry

In this section, we write variables as $r_i.v_g[e]$, where $r_i.v_g$ represents a group of variables of r_i that share a name v_g (e.g., `signal` in Figure 5), and e is the scalar value used as the index. Without loss of generality, for non-grouped variables and known rebecs, we assume a cluster typed with the singleton scalar set [1].

Definition 12 (Grouped KR lists) The grouped known rebecs list of r_i is defined as $L_i = (L_{i1}, L_{i2}, \ldots, L_{ib_i})$, where 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 5, `Controller` has two known rebecs of type `Train` sharing the name `trn`. So its grouped known rebecs list contains one sub-list with two elements, namely $L_3 = (L_{31})$, where $L_{31} = (1, 2)$, assuming that 1 and 2 are the indices of the `Train` instances `t1` and `t2`.

Definition 13 (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 14 (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}.

This definition is illustrated at the end of this section. 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 15 (Permutation on states - extension of Def. 10) Given a 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.

The idea here is to allow permuting the grouped known rebecs internally using rotary permutations whenever the strict ordering is irrelevant. This is indeed due to the internal symmetry of the rebec. Considering the new definition of applying permutations on states, Theorems 1 and 2 presented in the previous sub-section still hold (by taking the proper rotary permutations into account). However, we need to rephrase Lemma 1 as follows.

Lemma 2 *Given a permutation π, $\pi(s_0) = s_0$ if π preserves rebec types and grouped known rebec relation and the parameters to the initial message servers of symmetric rebecs are symmetric.*

The proof is again straightforward from the definition of initial state (Def. 5) and the application of permutation on states considering grouped known rebecs and variables (Def. 15). Note that the same definition of symmetric parameters is applicable, because scalar variables cannot be passed as parameters to the initial message server. That is due to the fact that the scope of scalar variables is inside reactive classes.

Recall that the permutation $\pi = /2, 1, 3/$ does not preserve the known rebec relation of the bridge controller example and so Lemma 1 is not applicable. 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, Lemma 2 ensures that $\pi(s_0) = s_0$ and hence Theorem 1 states that π is an automorphism of the model.

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) [60,50,26–28]. We use *static* partial order reduction (as explained in [35]), which is

considered to be the most practical variant of the partial order reduction techniques. The idea here is that instead of considering all enabled actions at a given state s (denoted by $enabled(s)$ in the following), it is enough to explore only a subset based on static characteristics of the actions. Recall from Section 2.1 that action a_j is enabled in s, if the corresponding message is at the queue head of rebec r_j.

Definition 16 (Invisibility) An action that does not affect the satisfiability of the (propositions used in the) specification to be checked is called invisible. A transition labelled with an invisible action is also called invisible.

Definition 17 (Independence) A symmetric irreflexive relation $I \subseteq A \times A$ on actions is said to be an independence relation iff for all $(a_i, b_j) \in I$ and for each s such that $\{a_i, b_j\} \subseteq enabled(s)$, the following conditions hold:

- if there exists t_1 such that $s \xrightarrow{a_i} t_1$ then $b_j \in enabled(t_1)$.
- 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 $t_1' = t_2'$.

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

Definition 19 (Safety) An action is called safe if it is invisible and globally independent. All the transitions labeled with a safe action are also called safe.

To apply static partial order reduction, the safety of actions must be known a priori, i.e., must be determined by a 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.

This strategy alone may cause some postponed enabled actions to be ignored forever. This so called *ignoring* problem may occur in the case of a loop. Generally, the solutions to the ignoring problem are called *provisos* (e.g., stack proviso [35], alternate proviso [49] or provisos for safety and liveness properties in [25]) whose need was first recognized by Valmari [60]. 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 can be only one message at the queue head). This action may result in more than one transition (if it involves nondeterministic choices). 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.

Theorem 3 *If a message server is only composed of safe assignment and safe send statements, its corresponding action is safe.*

The proofs of the theorems are given in Appendix C. According to Theorem 3, in order to determine the safety of message servers statically, we need to determine the safety of assignments and send statements. In addition as shown in Appendix C, the assignments in the `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 [41] (see Lemma 7 in Appendix C). 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, known-rebecs can be statically bound to real rebecs. Then by a static analysis of the send statements in the model, it is possible to find the rebecs that access each queue (i.e., send messages to the rebec that owns the 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. As a result, it cannot disable or be disabled by other actions.

5 The Rebeca Model Checking Tool

The Rebeca model checking tool consists of two components: a translator and an engine.[3] 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 [61, 62]. In this approach, the system and the specification (of the negation of the desired property) are each specified 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 a reachable accepting cycle (a cycle reachable from the initial state and containing at least one

[3] `http://www.reykjavikuniversity.is/icerose/applying-formal-methods/tools/` or `http://ece.ut.ac.ir/fml/rmc.htm`

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.

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++` objects. The files containing these objects are placed automatically beside the engine (Modere). The whole package is then compiled to produce an executable for model checking the given Rebeca model.

In Modere, the local states of rebecs are stored locally. A 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 is similar to the "collapse" compression method used in SPIN [34]. Modere uses Nested Depth First Search (NDFS) [36] for computing the product automaton and performing model-checking on-the-fly. For better memory management, Modere uses a non-recursive implementation of NDFS, and handles the search stack manually. Furthermore, Modere considers only the fair sequences of execution. An infinite sequence is considered (weakly) fair when all the rebecs are infinitely often executed or disabled.

5.1 Exploiting Symmetry

In Section 3, we showed that a permutation should satisfy three conditions in order to be an automorphism: preserve rebec types, preserve (grouped) known rebec relation and preserve the symmetry of the parameters to the initial message server. The algorithm in Figure 6 checks if a permutation (π or `pi`) exists that maps r_i to r_j while preserving rebec types and known rebec relation. If the algorithm returns true, `pi` contains this permutation. The algorithm is extended in Figure 7 to consider grouped known rebec relation. Checking the third condition is straightforward.

In this subsection, the names in `type-writer` font refer to variables in the algorithms. First, $\pi($`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 lists 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 add $\pi($`x`$) =$ `y` to the permutation, their known rebecs must be checked, too. 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. If the computed permutation `pi` is not complete, we can repeat the algorithm for checking the equivalence of the missing rebec indices until a complete permutation is found (cf. Section 5.1.1).

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 7). 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 (cf. Definition 14). 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 7 that must match via ψ_t. There are three possibilities. First, suppose $\pi($`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`

```
00  int[] pi; // permutation
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. 6 Inter-rebec symmetry detection algorithm

```
00  int[] pi; // permutation
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. 7 General symmetry detection algorithm (inter- and intra-rebec symmetry)

to 0. Third, if for some u, $\pi(\mathtt{x}[u])$ 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.

5.1.1 Applying Symmetry Reduction

Applying symmetry reduction during model checking amounts to computing the representative of every state we reach. One way to tackle this problem is by computing all the automorphisms of the model before starting the model checking. To this end, we can run the `check` algorithm for every pair of rebecs and thus find the equivalence groups of rebecs. Then we can compute the group of automorphisms that correspond to this partitioning of the rebecs. Then, during model checking we need to find the proper automorphism, such that unique representatives are obtained for the equivalence classes induced on states; for example, the automorphism resulting in the smallest

	(phil0)	(phil1)	(phil2)	(phil3)	(fork0)	(fork1)	(fork2)	(fork3)
rebec index	0	1	2	3	4	5	6	7
local state	5	7	3	4	1	8	4	6

Fig. 8 A global state in the dining philosophers example

state representation w.r.t. lexicographic ordering (cf. [13,7]). A naive computation of this, i.e., comparing all automorphisms, may be computationally expensive because in full symmetry the number of automorphisms is exponential. The advantage of the naive algorithm is that it results in more reduction (at the cost of more execution time).

Alternatively, we propose to use the `check` algorithm at model checking time to compute the proper automorphism on-the-fly. The idea is again to select as representative the smallest state (w.r.t. the lexicographic ordering) from the equivalence class induced by the symmetry group resulting from Theorem 2. In this approach, first we (lexicographically) sort the local states of the rebecs. Then, using the `check` algorithm, we find the automorphism that maps (the index of) the rebec with the smallest local state to 0. If no such automorphism exists we try the rebec with the second smallest local state and so on. If the `check` does not produce a complete permutation, we complete the permutation by calling `check` again to assign the rebec with the smallest remaining local state to the first empty element of the permutation. The process is repeated until the proper permutation is found (which may very well be the identity permutation).

For example assume Figure 8 shows a global state of the dining philosophers example, where the local states are represented by numbers. Since fork0 (rebec with index 4) has the smallest local state, we first try to map $\pi(4) = 0$ by calling `check(4,0)`. This is not possible because rebecs `0` (i.e., phil0) and `4` (i.e., fork0) are not of the same type. The rebec with the next smallest local state is phil2. Calling `check(2,0)` returns `true` and it results in `pi` holding the permutation $/2, 3, 0, 1, 6, 7, 4, 5/$.

We took the idea of using lexicographic ordering on the states from Bosnacki et al. [7]. As they mention, in some cases, this heuristic may result in multiple representatives when more than a rebec with the same local states exist. The algorithm can be made more accurate at the cost of comparing all orderings of such rebecs, which in the worst case is again exponential. Nevertheless, the empirical results in Section 6 are encouraging. In that section, we compare the on-the-fly approach with the naive one.

5.1.2 Complexity Analysis

We show now that the on-the-fly algorithm for finding the representative states works in polynomial time (in the number of rebecs in the given model). First consider the `check` algorithm. In its both versions, 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 `check` algorithm is $O(n^2)$.

For computing a representative state using the on-the-fly approach, first it takes $O(nlog(n))$ to sort the local states. In the worst-case, we may then compare every pair of rebecs, which results in an execution time of $O(nlog(n) + n^2 \times n^2) = O(n^4)$ in total.

5.1.3 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 necessarily complete. In fact, all practical approaches to using symmetry reduction (cf. related work in Section 1.1) suffer from the same inaccuracy. That is what we lose, as we want to avoid the 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 (except the line of work by Miller, Donaldson et al. [17–20], see related work in Section 1.1), as it has usually been the modeler's task to specify the symmetry explicitly, or general graph-isomorphism algorithms are employed. In Rebeca, we can find the automorphisms by analyzing the known rebec relation, because we have encapsulated objects that communicate only via asynchronous message passing. We expect our techniques to be generalizable to other high-level models of concurrency of which encapsulation and asynchronous message passing form the basis.

The current implementation of Modere is based on NDFS. Therefore, we adopted the method in [5] for exploiting symmetry. As shown by Bosnacki [5] with a simple example in terms of a state graph, under fairness assumptions, this method does not necessarily produce all the counter examples. Nevertheless, any property violation 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, or for the verification of safety properties that do not require fairness.

5.2 Combining Partial Order and Symmetry Reduction

Modere, as described in our previous work [41], also supports partial order reduction. The combination of partial order reduction and symmetry reduction techniques was first studied by Emerson et al. [22]. 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). Iosif [38] proposes an algorithm for applying partial order reduction while constructing the quotient structure on-the-fly. Since Modere does indeed construct the quotient structure on-the-fly, the latter approach is more appropriate. Furthermore, in this algorithm, the ample function works on the original state graph of the system.

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 compiler directives (like the code for partial order reduction in SPIN). To select each of these reduction methods, the modeler can decide to include the relevant code in compilation. This means that no decision about the reduction method is made during run-time resulting in faster execution.

6 Empirical Results

In this section, we provide the results from model checking some case studies, by applying symmetry and partial order reduction separately and in combination. In all these cases, we checked the models for deadlock on a 3.2GHz Intel processor with 2GB main memory. Although these case studies do not make full use of Modere's features like LTL model checking and fairness, they do demonstrate an effective application of the developed reduction techniques in model checking.

Table 9 compares the effect of the implemented reduction techniques. A plus (+) in this table implies that the model checker has used up all the available physical memory and thus model checking could not be finished. A minus (-) is used when a reduction technique was not applicable, i.e., would cause no reduction. Note that the reduction in model checking time is not proportional to the reduction in states, due to the overhead of using symmetry and partial-order reduction.

Table 10 compares the results of on-the-fly reduction based on inter-rebec and intra-rebec symmetry detection with a naive algorithm. The naive algorithm computes statically the symmetry groups based on intra-rebec symmetry detection technique (when applicable) and at each state compares the application of all automorphisms to find the lexicographically smallest state. Table 10 also shows the size of the automorphism groups computed statically (in the column under 'Aut Size').

Dining Philosophers. The dining philosophers problem was introduced in Section 2. The size of this model is shown as the number of philosophers in the model. We checked this model for 2 to 5 philosophers. Model checking 6 philosophers was not possible with any reduction technique. With four philosophers, binding rebecs in the same way as shown in Figure 2 (in Section 2) makes the model symmetric (with respect to inter-rebec symmetry). In order to avoid deadlock, no symmetry can be used for other numbers of philosophers. However, partial order reduction can be applied to all sizes. The only safe action in this model is the initial message server of Fork, which contains only assignments (cf. Section 4). Even one safe action can result in considerable reduction. This is seen better when the two reduction techniques are combined.

Trains - Bridge controller. This model was used as the motivating example for intra-rebec symmetry in Section 3.3. This example is extended so that it can be modeled with different configurations by increasing the number of trains. The number of trains is used to show the size of these models in Table 9. Partial order reduction in this model is due to the safety of the initial and release message servers of the bridge controller. Partial order reduction is not effective with models of size 8 or greater. Using scalar sets of the proper size in the bridge controller makes it possible to use intra-rebec symmetry. This technique is more effective than partial order reduction in this example and can handle up to 8 trains. The combination of the two techniques can even handle 9 trains, in which case 21 million states are stored.

		No Reduction		POR		SYMM		POR + SYMM	
	size	Time [sec]	# of states	Time [sec]	# of states	Time [sec]	# of states	Time [sec]	# of states
Phil	2	0	285	0	70	-	-	-	-
	3	0	3062	0	2985	-	-	-	-
	4	5	374K	2	196K	5	237K	1	62K
	5	9	676K	6	316K	-	-	-	-
Train	2	0	94	0	90	0	57	0	41
	3	0	788	0	755	0	340	0	253
	4	0	6344	0	5720	0	2165	0	1093
	5	0	51K	0	49K	0	15K	0	9538
	6	5	432K	4	419K	3	107K	1	54.8K
	7	58	3.8M	46	3.7M	23	837K	7	501K
	8	+	+	+	+	282	6.7M	47	4.2M
	9	+	+	+	+	+	+	1140	21M
LB	4/2	0	21K	0	10.8K	0	7520	0	1664
	6/2	25	1.34M	7	676K	3	65.4K	0	8814
	4/3	2	106K	0	46K	3	69.6K	0	11.2K
	6/3	268	9.8M	42	3.74M	46	893K	1	59.4K
	4/4	10	436K	1	172K	10	233K	1	42K
	6/4	+	+	+	+	+	+	1	90K
2PC	2	0	324	-	-	0	166	-	-
	3	8	617K	-	-	3	105K	-	-

Fig. 9 The effect of partial order and symmetry reduction techniques. This table is based on intra-rebec symmetry except for dining philosophers.

		No Reduction		On-the-fly detection				A priori detection		
				Inter-rebec		Intra-rebec		Naive reduction		
	size	Time [sec]	# of states	Time [sec]	# of states	Time [sec]	# of states	Time [sec]	# of states	Aut size
Phil	4	5	374K	5	237K	-	-	5	187K	4
Train	2	0	94	-	-	0	57	0	50	2
	3	0	788	-	-	0	340	0	288	3
	4	0	6344	-	-	0	2165	0	1712	4
	5	0	51K	-	-	0	15K	0	10.9K	5
	6	5	432K	-	-	3	107K	1	76.4K	6
	7	58	3.8M	-	-	23	837K	7	580K	7
LB	4/2	0	21K	0	10.6K	0	7520	1	4833	16
	6/2	25	1.34M	4	104K	3	65.4K	40	40.2K	144
	4/3	2	106K	2	59K	3	69.6K	10	33.2K	24
	6/3	268	9.8M	38	840K	46	893K	667	201K	216
	4/4	10	436K	11	312K	10	233K	65	121K	32
	6/4	+	+	+	+	+	+	4840	889K	288
2PC	2	0	324	0	166	0	166	0	166	2
	3	9	617K	6	208K	3	105K	4	103K	6

Fig. 10 Comparing symmetry reduction heuristics.

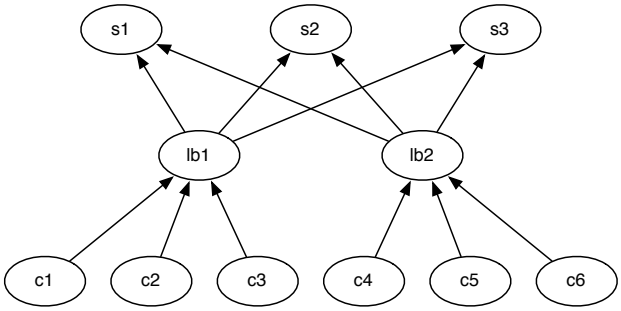


Fig. 11 Initial communication graph for the load-balancer example

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

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

```
reactiveclass Client (2) {
  knownrebecs {
    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. 12 Rebeca code for the load-balancer 6/3 (i.e., 6 clients and 3 servers)

Table 10 shows that a naive reduction behaves better than the on-the-fly reduction heuristic for this example. This model has a star topology with the controller in the middle. The only symmetry in this model is the symmetric internal behavior of the controller. Therefore, the size of the automorphism group is equal to the number of trains in the model, which is due to the rotary nature of the permutations used in intra-rebec symmetry. The small size of the symmetry group makes it practically possible to naively compare all automorphisms in order to compute a unique representative.

Load Balancer. In this problem (from [19]), there are some identical clients that need some specific service, which is provided by a number of 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, all 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. The size of this model is shown by the number of clients and servers (written as c/s).

Figure 11 shows the known rebec relation of the 6/3 model. As shown here, the clients are not introduced as known-rebecs of the servers. Instead, the servers receive, as 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 (see Figure 12), thus allowing us to use intra-rebec symmetry reduction in this model.

This model is subject to both inter-rebec and intra-rebec symmetry. Even without taking the symmetric behavior of the load balancers into account, we can freely permute the clients in each side (based on inter-rebec symmetry). Using intra-rebec symmetry, we can permute the servers, too; but only using rotary permutations. Theoretically, intra-rebec symmetry should produce more reduction because the symmetry group is bigger, but using on-the-fly detection, this is not always the case, as can be seen for 4/3 and 6/3 configurations. The reason is that the heuristics applied in finding the representative states do not always compute a unique representative. Table 10 shows that by the growth of the size of the automorphism group, the naive reduction gets very slow, but at the same time very effective in terms of memory usage. For the 6/4 model, the on-the-fly techniques used up all the memory and could not finish, while the naive method finished successfully after almost one and a half hours.

The application of partial order reduction is also quite effective in this model, although only the initial message server of load balancer is safe. As can be seen in Table 9, the only possibility for checking 6 clients and 4 servers is by combining partial order and symmetry reduction techniques.

Two-phase commit. The last case study we modeled is the two-phase commit protocol (2PC) for distributed databases. There are some number of similar nodes that may start a transaction, which can in turn be committed if all nodes come to a consensus. In our model, when a node nondeterministically decides to create a transaction, it starts it on all cooperating nodes. The response from the cooperating nodes indicates whether they commit or fail. There will be a consensus when the number of received responses is equal to the expected value and they are all true.

In this model, partial order reduction is not applicable because none of the message servers is safe. The communication graph of this model is a complete graph because each node is able to communicate with all other nodes. In a model with n nodes, inter-rebec symmetry can find only n automorphisms because of the order of the known-rebecs. Intra-rebec symmetry relaxes this limitation. For a model with two nodes, the results of inter-rebec and intra-rebec symmetry are the same because each node has only one known rebec. With three nodes, we can see that intra-rebec symmetry has a better performance (very close to optimal, i.e., the naive reduction). With 4 nodes, we couldn't model check this example.

```
reactiveclass Node(10) {
   knownrebecs {
      Node known[t:1..2];
   }
   statevars {
      boolean receivedResults;
      byte received;
      byte expected;
      boolean[t] cooperator;
   }
   msgsrv initial() {
      self.createTransaction();
   }
   msgsrv startTransaction() {
      sender.coResponse(?(true, false));
   }
   msgsrv coResponse(boolean result) {
      received = received + 1;
      if (!result)
         receivedResults = false;
      if (received == expected) {
         self.createTransaction();
      }
   }
   msgsrv createTransaction() {
      if (?(true,false)) {
         boolean result;
         forEachValueOf(t)
            cooperator[t] = false;
         received = 0;
         expected = 0;
         receivedResults = true;
         forEachValueOf(t) {
            cooperator[t] = true;
            expected = expected + 1;
            known[t].startTransaction();
         }
         expected = expected + 1;
         self.coResponse(?(true, false));
      }
      else {
         self.createTransaction();
} } }
main {
   Node node1(node2, node3):();
   Node node2(node3, node1):();
   Node node3(node1, node2):();
}
```

Fig. 13 Two-phase commit (2PC) protocol in Rebeca with three nodes

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 by using the so called *rebec variables*. In this paper, we addressed the problem of model checking Rebeca by applying symmetry and partial order reduction methods.

Considering the asynchrony in the model and the actor-based nature, we proposed efficient symmetry detection and reduction techniques for Rebeca. 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. We then proposed intra-rebec symmetry to be used when there are rebecs with internally symmetric behavior (including but not limited to star topology). 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 simpler than the traditional approaches that require the modeler to exhibit the symmetry in the *whole* system. We proved the soundness of our algorithms. Finally, we proposed a heuristic algorithm that employs these symmetry detection techniques in order to calculate the representative states on-the-fly.

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 empirical 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 the search stack manually.

Other reduction techniques applied to Rebeca in theory include compositional verification [56] and slicing-based reductions [51]. In future, these techniques can be added to Modere as well, and their combination with partial order and symmetry should be studied. Another future direction is the generalization of our symmetry detection algorithms to Promela. On the other hand, the ideas of scalar sets can be further studied for detecting data symmetries in Rebeca. For example, by reusing the ideas of intra-rebec symmetry, one may consider the application of scalar-sets in breaking the symmetry of parameters when applicable.

Acknowledgement

The work of the first author is funded by the European IST-33826 STREP project on Modeling and Analysis of Evolutionary Structures for Distributed Services (Credo). 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 is partially supported by the project "The Equational Logic of Parallel Processes" (nr. 060013021) of The Icelandic Research Fund.
We would like to thank the anonymous reviewers who helped us improve the quality of the paper with their rightful comments.

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 3 *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. 15).*

Proof Considering the definition of scalar expressions in Section 3.3.2, 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. 15. 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.13). 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 4 *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., $r_j.m[1] = a$. Therefore, 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 equality or non 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, the equality or non equality of two rebec variables is preserved by π.
3. Scalar variables: Scalar expressions can only be compared for equality or non equality with other scalar expressions. Since by applying π on s, the values of scalar expressions are mapped using ψ, the equality or non 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 (Def. 7). 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. Section 3.3.2), we use ψ to denote the rotary permutation associated with that cluster (according to Def. 15).

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 4, 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:
   $\forall_{i>0}$ ▪ $r_j.m[i] \leftarrow r_j.m[i+1]$
   $\forall_{i>0}$ ▪ $r_j.s[i] \leftarrow r_j.s[i+1]$
   $\forall_{i>0} \blacksquare \forall_{0<k\leq h_j}$ ▪ $r_j.p_k[i] \leftarrow r_j.p_k[i+1]$

   By Definition 10, it is deduced that the transition $\pi(s) \xrightarrow{a_{\pi(j)}} t'$ contains:
   $\forall_{i>0}$ ▪ $r_{\pi(j)}.m[i] \leftarrow r_{\pi(j)}.m[i+1]$
   $\forall_{i>0}$ ▪ $r_{\pi(j)}.s[i] \leftarrow r_{\pi(j)}.s[i+1]$
   $\forall_{i>0} \blacksquare \forall_{0<k\leq h_j}$ ▪ $r_{\pi(j)}.p_k[i] \leftarrow r_{\pi(j)}.p_k[i+1]$
2. Note that according to Lemma 3, 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 3, 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 15, $r_{\pi(j)}.v_g[\psi(e)]$ 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:
   – data variable: the same value as in s is used.
   – rebec variable: the permutation π is applied to the value in s.
   – scalar expression: the proper rotary permutation ψ is applied to the parameter value when applying π.

   Assuming that η_i represents the symmetric value corresponding to n_i, in t' we have: for y such that $r_{\pi(k)}.m[y] = \perp \wedge \forall_{0<z<y} r_{\pi(k)}.m[z] \neq \perp$) perform
   $$r_{\pi(k)}.m[y] \leftarrow m, \quad r_{\pi(k)}.s[y] \leftarrow \pi(j), \text{ and } \forall_{1\leq i\leq h_{\pi(k)}} r_{\pi(k)}.p_i[y] \leftarrow \eta_i.$$
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 $\pi(s) \xrightarrow{a_{\pi(j)}} \pi(t) \in T$. □

B Proof of Theorem 2

Theorem 2. *The set of all permutations satisfying Theorem 1 form a group.*

Proof The conditions in Theorem 1 are preserving rebec types, known rebec relation and the symmetry of the parameters to the initial message server. The identity permutation trivially satisfies these conditions.

We first show that the set of permutations satisfying these conditions is closed under composition of permutations. Assume that π and ϕ are in the set.

– Since π and ϕ preserve rebec types, it is easy to see that for any i, the rebecs r_i, $r_{\pi(i)}$ and $r_{\phi\circ\pi(i)}$ have the same type, therefore, $\phi \circ \pi(i)$ preserves rebec types.
– We have: $\phi\circ\pi(K_i) = \phi(\pi(K_i)) = \phi(K_{\pi(i)}) = K_{\phi(\pi(i))} = K_{\phi\circ\pi(i)}$, therefore, $\phi\circ\pi$ preserves known rebec relation.
– If p is a rebec parameter to the initial message of r_i, the same parameter for $r_{\pi(i)}$ and $r_{\phi\circ\pi(i)}$ has the value $\pi(p)$ and $\phi \circ \pi(p)$, respectively. In the case of a data parameter, it has the same value.

It remains to show that the set contains the inverse of all its elements. Assume that π is in the set, and for a given i assume that $j = \pi^{-1}(i)$.

– Since π preserve the rebec types, r_j and $r_{\pi(j)}$ have the same type, which are respectively equivalent to $r_{\pi^{-1}(i)}$ and r_i and thus π^{-1} preserves rebec types.

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

- We know that $\pi(K_j) = K_{\pi(j)} = K_i$. Hence, applying π^{-1} to the elements of K_i results in K_j. Therefore, we have $\pi^{-1}(K_i) = K_j = K_{\pi^{-1}(i)}$.
- The parameters of r_i and r_j are also trivially symmetric.

This proves the theorem for inter-rebec symmetry and simple known rebecs. We chose this for a simpler presentation, but the proof can be easily adapted to address grouped known rebecs by adding the rotary permutations when necessary. □

C Proof of Theorem 3

In this section, we study the safety of different sub-actions (cf. Section 2) and actions in Rebeca. In the following, note that actions that only change the local variables of a rebec, like assignments, are globally independent.

Lemma 5 *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 globally independent, because it does not affect the variables of other rebecs. □

Lemma 6 *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 7 *Sending a message from r_i to r_j is safe if r_i is the exclusive sender to r_j (similar to the idea of exclusive send/receive constructs,* `xs` *and* `xr`*, in Promela).*

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 3. *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. □

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 [37], 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. Behjati, R., Sabouri, H., Razavi, N., Sirjani, M.: An effective approach for model checking SystemC designs. In: Proc. International Conference on Application of Concurrency to System Design (ACSD'08), pp. 56–61 (2008)
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.: A heuristic for symmetry reductions with scalarsets. In: Proceedings of the International Symposium of Formal Methods Europe (FM'01), *Lecture Notes in Computer Science*, vol. 2021, pp. 518–533. Springer (2001)
7. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer (STTT) **4**(1), 92–106 (2002)
8. Bosnacki, D., Donaldson, A.F., Leuschel, M., Massart, T.: Efficient approximate verification of promela models via symmetry markers. In: Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07), *Lecture Notes in Computer Science*, vol. 4762, pp. 300–315. Springer (2007)
9. Bosnacki, D., Edelkamp, S. (eds.): Proc. Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, *LNCS*, vol. 4595. Springer (2007)
10. Chang, P.H., Agha, G.: Supporting reconfigurable object distribution for customized web applications. In: The 22nd Annual ACM Symposium on Applied Computing (SAC), pp. 1286–1292 (2007)
11. Chang, P.H., Agha, G.: Towards context-aware web applications. In: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), pp. 239–252 (2007)
12. Cheong, E., Lee, E.A., Zhao, Y.: Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. In: Proceedings of the 3rd international conference on Embedded networked sensor systems, SenSys 2005, pp. 302–302 (2005)
13. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Hu and Vardi [37], pp. 147–158
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
15. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design **9**(1/2), 77–104 (1996)
16. 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)
17. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Proceedings of the International Symposium of Formal Methods Europe (FM'05), *Lecture Notes in Computer Science*, vol. 3582, pp. 481–496. Springer (2005)
18. Donaldson, A.F., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. Electronic Notes in Theoretical Computer Science **185**, 63–76 (2007)
19. Donaldson, A.F., Miller, A., Calder, M.: Finding symmetry in models of concurrent systems by static channel diagram analysis. Electronic Notes in Theoretical Computer Science **128**(6), 161–177 (2005)
20. Donaldson, A.F., Miller, A., Calder, M.: Spin-to-Grape: A tool for analysing symmetry in Promela models. Electronic Notes in Theoretical Computer Science **139**(1), 3–23 (2005)
21. Emerson, E., Sistla, A.: Symmetry and model checking. Formal Methods in System Design **9**(1–2), 105–131 (1996)
22. 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)
23. 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)

24. 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)
25. Evangelista, S., Pajault, C.: Some solutions to the ignoring problem. In: Bosnacki and Edelkamp [9], pp. 76–94
26. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proc. the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 110–121. ACM Press, New York, NY, USA (2005)
27. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke and Kurshan [16], pp. 176–185
28. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bosnacki and Edelkamp [9], pp. 95–112
29. 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)
30. Herstein, I.: Topics in Algebra. Xerox (1964)
31. Hewitt, C.: Procedural embedding of knowledge in planner. In: Proc. the 2nd International Joint Conference on Artificial Intelligence, pp. 167–184 (1971)
32. Hewitt, C.: What is commitment? physical, organizational, and social (revised). In: Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II, Lecture Notes in Computer Science, pp. 293–307. Springer (2007)
33. 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)
34. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
35. 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 (1995)
36. 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)
37. 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)
38. 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)
39. Ip, C., Dill, D.: Better verification through symmetry. Formal methods in system design **9**(1-2), 41–75 (1996)
40. Ip, C.N.: State reduction methods for automatic formal verification. Ph.D. thesis, Department of Computer Science, Stanford University (1996)
41. 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)
42. 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)
43. Kakoee, M.R., Shojaei, H., Ghasemzadeh, 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), pp. 3760–3763 (2007)
44. 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)
45. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers **12**(3), 231–260 (2003)

46. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proc. of the International Symmetry Conference, Edinburgh, UK, pp. 71–85 (2007)
47. McMillan, K.: Symbolic Model Checking. Kluwer Academic, Boston, MA, USA (1993)
48. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. **38**(3) (2006)
49. 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)
50. 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)
51. Saboori, H., Sirjani, M.: Slicing-based reductions for Rebeca. In: Proc. FACS'08, ENTCS, pp. 57–71. Elsevier (2008). To be published
52. Saïdi, H.: Discovering symmetries. In: 11th International Workshop on Formal Methods: Applications and Technology (FMICS/PDMC'06), *LNCS*, vol. 4346, pp. 67–83. Springer (2006)
53. Schneider, S.: The B-Method: An Introduction. Palgrave (2001)
54. 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)
55. 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)
56. 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)
57. 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)
58. 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), pp. 145–150. IEEE Computer Society (2004)
59. 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)
60. Valmari, A.: A stubborn attack on state explosion. In: Clarke and Kurshan [16], pp. 156–165
61. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Proc. of the 8th VMCAI, LNAI 4349, pp. 137–150 (2007)
62. Vardi, M.Y., Wolper, P.: An automata theoretic approach to automatic program verification. In: D. Kozen (ed.) Proc. Symposium on Logic in Computer Science, pp. 322–331. IEEE Computer Society (1986)