Efficient Symmetry Reduction for an Actor-Based Model

M.M. Jaghoori[1], M. Sirjani[2], M.R. Mousavi[3], and A. Movaghar[1]

[1] Sharif University of Technology, Tehran, Iran
[2] University of Tehran and IPM, Tehran, Iran
[3] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Symmetry reduction is a promising technique for combatting state space explosion in model checking. The problem of finding the equivalence classes, i.e., the so-called *orbits*, of states under symmetry is a difficult problem known to be as hard as graph isomorphism. In this paper, we show how we can automatically find the orbits in an actor-based model, called Rebeca, without enforcing any restriction on the modeler. The proposed algorithm solves the orbit problem for Rebeca models in polynomial time. As a result, the simple actor-based Rebeca language can be utilized efficiently for modeling and verification of systems, without involving the modeler with the details of the verification technique implemented.

1 Introduction

Model checking is the automatic and algorithmic way for the verification of system correctness. State space explosion is a major obstacle in exploiting model checking in practice. The problem arises when we try to explore all the possible states of a system to see whether a specific property is met or not. To overcome this problem, numerous methods have been proposed in order to avoid the construction of the complete state graph [6]. Among these methods are symbolic verification, partial order reduction, modular (parameterized) model checking, and symmetry reduction [8, 13, 18]. These techniques are sometimes combined to achieve even more compression in the representation of the system under analysis [1, 9, 11].

The symmetry technique is based on the fact that many systems are composed of similar and symmetric parts. These symmetric parts yield a similar behavior and have similar state graphs. The sub-graphs of these parts are usually interchangeable with respect to some permutation on the states. Therefore, it is possible to divide the state graph into symmetric graph quotients. One of these quotient graphs, annotated with the corresponding permutations, is shown to be enough for checking a general class of properties for the whole system [8]. However, for the technique to be useful, we need to find these permutations without constructing the total state space.

In concurrent systems, we can make use of the notion of *processes* running in parallel. Theoretically, a process is responsible for the behavior of some part of the system. We can consider the symmetry among processes and look for the permutations of processes, as suggested in [10]. Compared to permutations on states, permutations on processes are easier to find and maintain. However, checking all possible permutations for finding the ones that reveal the symmetries is not computationally efficient. Therefore, heuristic methods should be utilized for this purpose.

Alternatively, designer's insight may be use to reveal symmetry. Some tools, such as Murphi [13, 14], SMC [17] and SymmSpin [4], use the notion of scalar sets or a similar concept. Scalar sets are fully symmetric indices that are added to the model by the modeler to expose the symmetry of the system, so that the compiler can detect the symmetries automatically. In this paper, an algorithm is presented for finding the symmetry in Rebeca models automatically, with no changes to the syntax of Rebeca.

Rebeca [15] is an actor-based language, which can be used at a high level of abstraction for modeling concurrent systems. Using an object-based approach, and the asynchronous message-passing paradigm, Rebeca provides a basis that naturally fits in modular verification methods. Reactive objects are instantiated from reactive classes as templates. This suggests the idea that there is an inherent symmetry among the reactive objects of the same type (instantiated from the same reactive class).

It is preferred that the modeler is only involved in modeling issues, rather than the details of verification techniques. The interesting characteristic of Rebeca is that the only communication mechanism among the rebecs is through asynchronous message passing. This helps us find the symmetric permutations in polynomial time in the number of processes, without any extra work in modeling. We show that symmetry can be utilized in the presence of dynamic object creation and a special kind of changing topology.

In the rest of this paper, we first provide an overview of the symmetry reduction technique in Section 2. In Section 3, we introduce Rebeca and its semantics. Section 4 shows how symmetry reduction can be applied to Rebeca models and demonstrates the algorithm proposed for automatically detecting the symmetry in a Rebeca model. Section 5 extends our approach to the setting with dynamic creation of rebecs and dynamic topology. In Section 7, we present a brief comparison of our approach with related work. The concluding remarks and future work are presented in Section 8.

2 The Symmetry Reduction Technique

In this section, we explain the symmetry reduction technique [13, 8, 18]. The aim of this technique is to find the parts of the system that yield similar behavior. Intuitively, it is enough to run the model checking algorithm on one of these similar parts.

Consider a system M, consisting of n concurrently executing processes that communicate through shared variables.[1] Let I be the set $[1..n]$ of natural num-

[1] In later sections, we will use a restricted form of shared variables to represent asynchronous message passing in Rebeca.

bers. We assume that each process is identified by a unique index from I. The variables in M are also subscripted by an index set, which denotes the processes that access the variable, e.g. the variable $v_{1,3}$ is accessed by processes 1 and 3, and the variable w_2 is a local variable of process 2. Each process is defined as a set of actions, where each action is a conditional assignment of values to some of the variables. A specific action is said to be enabled at some state, if the respective condition evaluates to true. The whole system, called an *indexed transition system or briefly a* program, is viewed as the interleaving of the processes. In other words, each process may have zero or more actions enabled at each particular state, and which move the system to (probably) another state. Figure 2.(a) (taken from [8] with minor changes) shows a simple system composed of n identical processes that start in a non-critical state and try to enter their critical section (by executing action a_i), and then leave the critical section (by executing action b_i). There is a variable associated to each process, which shows whether it is in its critical section or not.

The formal definition of an *indexed transition system* M is given as a 4-tuple $\langle S, A, T, s_0\rangle$. S denotes the set of *global states*, where a global state is a valuation of all the variables; and s_0 is the *initial state*. The *transition relation* is defined as $T \subseteq S \times A \times S$, where A denotes the set of the *actions* of different processes. The transitions in T represent the behavior of the system; i.e. $(s, a_i, t) \in T$ when the action a from process i is enabled in state s and its execution leads to state t. We may write $s \xrightarrow{a_i} t$ for (s, a_i, t).

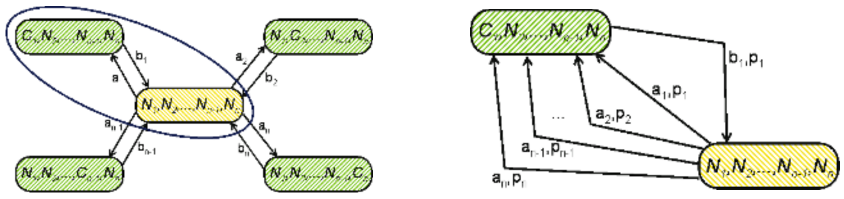
A permutation $\pi : I \to I$ is a bijection on the index set I. Recall that I is the set $[1..n]$. We write a permutation as $\pi = (i_1, \ldots, i_n)$, which means $\forall_{1\leq x\leq n}\pi(x) = i_x$. The set of all permutations on I is denoted by *SymI*.

The application of a permutation π on a global state s should result in the global state $\pi(s)$, which is defined as follows. For every variable $v_{i_1,\ldots,i_k}$, its value in the state s is given to the variable $v_{\pi(i_1),\ldots,\pi(i_k)}$, in the state $\pi(s)$. If $v_{\pi(i_1),\ldots,\pi(i_k)}$ does not exist, then $\pi(s)$ is undefined and π is said to be *inconsistent*. In addition, the application of π on an action a_i is the action $a_{\pi(i)}$, which must be a valid action in the system; otherwise, π is inconsistent. We say that a consistent permutation π is an *automorphism* of the indexed transition system M, when $\pi(s_0) = s_0$, and π preserves the transition relation, i.e., $s \xrightarrow{a_i} t \in T$ when $\pi(s) \xrightarrow{\pi(a_i)} \pi(t) \in T$.

The *set of automorphisms* of M is denoted by *AutM* and is a subgroup of *SymI* [8]. Given any subgroup $G \in AutM$, we can define an equivalence relation on S. The states s and s' are equivalent with respect to G when there is a $\pi \in G$ such that $\pi(s) = s'$. Each equivalence class is called an *orbit.* Intuitively, for model checking M, it is sufficient to construct the state space for the representatives of each orbit.

The system shown in Figure 2.(a) is an example of a symmetric state graph. Representatives of the two orbits of this system are distinguished by an ellipse around them; for example, the states $(C_1, N_2, \ldots, N_n)$[2] and $(N_1, C_2, \ldots, N_n)$

[2] Assume that each process i has a variable v_i that can be either C or N; then, by C_i or N_i we mean that v_i has the value C or N, respectively.



(a) Selecting a Representative From Each Orbit (b) Annotated Quotient Structure

Fig. 1

are equivalent, because applying the automorphism $(2, 3, \ldots, n, 1)$ on the former produces the latter.

The *annotated quotient structure (AQS)* for M is $\overline{M} = \langle \overline{S}, \overline{T}, s_0 \rangle$, where $\overline{S}$ is the set of the representative states (which contains exactly one state from each orbit) and $\overline{T} \doteq \{\overline{s} \overset{a_i,\pi}{\rightarrow} \overline{t} \mid \pi \in G, \overline{s} \in \overline{S} \wedge \overline{t} \in \overline{S} \wedge \overline{s} \overset{a_i}{\rightarrow} \pi(\overline{t}) \in T\}$. Figure 2.(b) shows the annotated quotient structure of the previous example. Note that in this graph, there is only one edge with action b. Each p_i shows the permutation $(i, i+1, \ldots, n, 1, 2, \ldots, i-1)$, which maps the i'th process to the first one (this notation for permutations is explained earlier in this section).

Emerson, et.al, in [8], show that $\overline{M}$ can be used in the automata theoretic approach to model check M against formulas that respect the symmetry of the system. This approach is extended in [10] for efficient model checking under fairness conditions. Bosnacki in [3] shows how symmetry reduction can be combined with the *nested depth-first search* algorithm. In these methods, it is always assumed that the orbit relation is previously known. However, the problem of finding the equivalence relations (orbits), known as the orbit problem, is in its general form shown to be as hard as graph-isomorphism [8, 5]. In the following sections, we introduce Rebeca as an actor based language and explain how we can automatically compute the equivalence relation without engaging the modeler in the details of the verification method used. The algorithm proposed in the later sections solves the orbit relation for a Rebeca model in polynomial time (with respect to the number of the processes in the system).

3 Rebeca: An-Actor Based Model

The actor model was originally introduced by Hewitt [12] as an agent-based language. It was later developed by Agha [2] into a concurrent object-based model. Rebeca (Reactive objects language) [15] is based on the actor model with an operational semantics.

3.1 Basic Definitions

A Rebeca model is constructed by the parallel composition of a set of *rebecs*, written as $R = ||_{i \in I}\, r_i$, where I is the index set that is used to identify each rebec. For the sake of simplicity, we ignore the dynamic features of Rebeca in this section, and hence, assume that I is a fixed set for a given model. We relax this assumption in Section 5, where dynamic behavior in Rebeca models is addressed.

The concurrent execution of rebecs is modeled by interleaving, i.e., rebecs are given turns for execution. For model checking a Rebeca model, all fair sequences of execution are considered. An infinite sequence is considered fair when all the rebecs are infinitely often executed or disabled.

The rebecs communicate by sending asynchronous messages. The messages that can be serviced by the rebec r_i are denoted by the set M_i. There is a message server corresponding to each element of M_i. Each rebec has an unbounded queue for storing its incoming messages. In each state, the message at the head of the queue of a rebec specifies which one of its message servers is enabled. Each rebec, in its turn, removes one message from the top of its queue and atomically executes the corresponding message server.

The local state of a rebec r_i is distinguished by the valuation of its local variables and its queue. The global state of the model is obtained by the combination of the local states of all rebecs. For each rebec r_i, an ordered list of known rebecs is introduced, whose indices are collected in K_i. The rebec r_i can only send messages to its known rebecs. Since a rebec can also send messages to itself, we always have $j \in K_j$. The known rebecs of all rebecs are specified statically. As a result, we can derive the *communications graph* of a Rebeca model, from the known rebecs lists. In this directed graph, nodes are rebecs, and there is an edge from r_i to r_j when $j \in K_i$.

The behavior of a Rebeca model is defined as the interleaving of the enabled rebecs in each state. A rebec is enabled, if its message queue is not empty. There is at least a message server 'initial' in each rebec, which is responsible for the initialization tasks, and the corresponding message is assumed to be in the queues of all rebecs in the initial state. The execution of a message server is defined as the atomic sequential execution of its statements. Statements may be either '(nondeterministic) assignments' or 'send' operations. An assignment changes the values of the local state variables. In the case of a nondeterministic assignment, a set of values is used to specify the next value of the variable. A rebec can send messages to its known rebecs. The messages may be accompanied by parameters. The sent messages, together with their parameters are placed (immediately) in (the tail of) the queue of the receiving rebec. The execution of statements may be restricted by some conditions (on the values of the local variables, the sender or the parameters of the message).

3.2 The Formal Semantics of Rebeca

The semantics of a Rebeca model is expressed with an indexed transition system $\langle S, A, T, s_0 \rangle$ (introduced in Section 2). Each state in the system is identified by the values assigned to the local variables of the rebecs, together with the messages (and their parameters and sender) in the queues of the rebecs. Without loss of generality, we assume that all local variables take values from the domain set D. This domain set includes the *undefined* value represented by $\perp$.

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

Assuming that there are w_j local variables in r_j, a local state of r_j can be represented formally as $s_j = (r_j.v_1, \ldots, r_j.v_{w_j}, r_j.m_1, \ldots, r_j.m_{x_j}, r_j.s_1, \ldots, r_j.s_{x_j}, r_j.p_{11}, \ldots, r_j.p_{h_j x_j})$, where $h_j \geq 0$, $x_j \geq 1$ and $w_j \geq 0$. A global state of the system is defined as the combination of the local states of all rebecs: $s = \prod_{i \in I} s_i$. The set S denotes the set of all the global states. In the initial state s_0, $r_i.m_1 =' initial'$ for all rebecs r_i. If the initial message server of r_i accepts i_j parameters, the variables $r_j.p_{11}$, $r_j.p_{12}$, $\ldots$, $r_j.p_{i_j 1}$ are also initialized as specified in the model. All other (local and queue) variables are assigned the value $\perp$.

Since message servers are executed atomically, each message server is equivalent to an action, unless there are nondeterministic assignments, in which case, one action is defined per each nondeterministic choice. The set A denotes the set of all actions resulting from the message servers. Therefore, the transition relation $T \subseteq S \times A \times S$ is defined as follows. There is a transition $s \xrightarrow{a_j} t$ in the system, if the value of $r_j.m_1$ in the state s is equal to the message corresponding to the action a, and the execution of a results in the state t.

In the following, we define different possible types of sub-actions that a transition $s \xrightarrow{a_i} t$ may contain. In the formulas below, the variables on the left hand side of $\leftarrow$ refer to variables in t and the ones on the right hand side refer to their values in s.

1. Message removal: This sub-action includes the removal of the first element of message, sender and parameter queues. By removing the first element, we mean shifting other elements of the queue toward the queue head. This sub-action exists in all actions. It can be written as:

   $\forall_{0<i<x_j} r_j.m_i \leftarrow r_j.m_{i+1}$, and $r_j.m_{x_j} \leftarrow \perp$, and
   $\forall_{0<i<x_j} r_j.s_i \leftarrow r_j.s_{i+1}$, and $r_j.s_{x_j} \leftarrow \perp$, and
   $\forall_{0<i<x_j, 0<k\leq h_j} r_j.p_{ki} \leftarrow r_j.p_{k(i+1)}$, and $r_j.p_{k(x_j)} \leftarrow \perp$.

2. Assignment: An assignment can be a statement like '$w \leftarrow d$', where w is the i'th local variable in r_j and $d \in D \setminus \perp$. This statement simply means: $r_j.v_i \leftarrow d$. The right hand side of an assignment may also be a more complex expression based on the local variables of r_j. In such cases, the expression can be evaluated with the values of the local variables in state s, and finally

a value like d is obtained. Therefore, for the sake of simplicity, we can assume that the right hand side of an assignment is always an explicit value.

3. Send: The rebec r_j may send a message m to r_k, where $m \in M_k$ and $k \in K_j$. As stated earlier, by K_j, we mean the ordered list of (the indices of) the known rebecs of r_j. The message m is assumed to have h_k parameters, say $n_1, \ldots, n_{h_k}$, where $n_i \in D$, $1 \leq i \leq h_k$. Like an assignment, a parameter may also be represented by an expression, which finally resolves into an explicit value from D. Recall that rebec r_k has h_k parameter queues. Note that n_i may be $\perp$; and for $i < h_k$, if n_i is $\perp$, then n_{i+1} must also be $\perp$. In addition, the number parameters that are not $\perp$ must agree with the number of arguments that the message server corresponding to m accepts. The result of this sub-action is:

   If $\exists_{0<y\leq x_k}(r_k.m_y = \perp \wedge \forall_{0<z<y} r_k.m_z \neq \perp)$, then
   $r_k.m_y \leftarrow m$, $r_k.s_y \leftarrow j$, $\forall_{1\leq i\leq h_k} r_k.p_{iy} \leftarrow n_i$

   Otherwise, x_k must be increased and the transition system of the Rebeca model cannot be constructed.

4 Symmetry in Rebeca

To exploit symmetry in an indexed transition system associated to a Rebeca model, we need to find a permutation group acting on the index set $[1..n]$ of the rebecs. With the permutation group, the state space is partitioned into orbits (equivalence classes). Since the rebecs of the same type (i.e. they are instances of the same reactive-class) exhibit similar behavior, it is reasonable to limit the permutations to those that preserve rebecs types. Theorem 1 helps to derive the symmetry in Rebeca models in a straightforward way. It simplifies the orbit problem by helping to obtain possible permutations prior to the real construction of the state space.

From now on, consider a system $R = \langle S, A, T, s_0\rangle = ||_{i\in I}\, r_i$ of a Rebeca model. Here, we redefine the application of a permutation on a global state. Definition 2 is repeated for easier reference.

Definition 1. *The application of a permutation π on a global state s, denoted by $\pi(s)$, is defined as follows:*

1- Variables that are not of 'rebec index' type (i.e., don't get their value from the set I), like $r_j.v_i$, $r_j.m_i$ and $r_j.p_{ki}$: Their values in state s, is assigned to the local or queue variables $r_{\pi(j)}.v_i$, $r_{\pi(j)}.m_i$ and $r_{\pi(j)}.p_{ki}$ in state $\pi(s)$, respectively.
2- Variables that are of 'rebec index' type, like $r_j.v_i$ and $r_j.s_i$ (sender queue): Suppose their value is state s is x. In state $\pi(s)$, the value $\pi(x)$ is assigned to variables $r_{\pi(j)}.v_i$ and $r_{\pi(j)}.s_i$, respectively.

For static Rebeca models, the latter case shrinks only to the case of sender queue variables. However, the more general case applies to dynamic Rebeca models, which is discussed in the next section.

Definition 2. *A permutation π, defined in I, is said to preserve the transition relation when $[s \xrightarrow{a_i} t \in T] \rightarrow [\pi(s) \xrightarrow{a_{\pi(i)}} \pi(t) \in T]$. Such a permutation is called an automorphism of R, if $\pi(s_0) = s_0$.*

Definition 3. *A permutation π is said to preserve rebec types, if for all i,j such that $\pi(i) = j$, the rebecs r_i and r_j are instances of the same reactive-class.*

Definition 4. *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: $\forall_{i \in I} K_{\pi(i)} = (\pi(t_1), \pi(t_2), \ldots, \pi(t_{P_i}))$.*

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

Given an automorphism of a Rebeca model, we can partition the rebecs into equivalence classes. To examine whether $\pi(s_0) = s_0$, the initialization of the system must be checked to ensure that the parameters sent with the *initial* message do not break the symmetry; i.e. equivalent rebecs receive similar values for the normal parameters to *initial*, and symmetric values for rebec parameters.

The obtained equivalence relation on rebecs can be used to derive a symmetry group on the states of the underlying structure. It shows how the simple natural object-based syntax of Rebeca helps us find the symmetry automatically. Next section presents an efficient algorithm that finds the symmetry groups of a given Rebeca model, if there is any.

4.1 Implementation

In this section, we present an algorithm for detecting symmetry in Rebeca models based on Theorem 1 of the previous section. In the following, we demonstrate how symmetry can be detected from normal Rebeca models, i.e., with no change in the syntax of Rebeca.

Theorem 1, implies that checking for equivalence of two rebecs, is reduced to finding a permutation that maps one to the other and preserves the known-rebec relation. The ordering among the known rebecs of each rebec helps us implement a polynomial time algorithm for this purpose. First, we show that checking whether two given rebecs belong to the same equivalence class can be done in linear time. It is performed in the *check* algorithm given below.

```
check (i, j) : boolean;
 if (i.type != j.type) return false;
 define pi as an empty array of size n;// pi[i] = permutation acting on i
 Let pi[i] := j;   // suppose permutation of i is j
 Let p1 := K(i); // the ordered known rebecs of i
 Let p2 := K(j); // the ordered known rebecs of j
 while p1 not empty do
    x := removeFirstElementOf (p1);
    y := removeFirstElementOf (p2);
    if (pi[x] is undefined)
      Let pi[i] := j;
      p1 += K(x);  // add to the end of the list
      p2 += K(y);  // add to the end of the list
    else if(pi[x] != y) // knownrebec relation is not preserved
      return false;
  od
  return true;
end
```

The inputs to *check*, i and j, are the indices of two rebecs. In this algorithm, we try to find a permutation π that maps i to j, and also respects the known-rebec relation. For this purpose, we take a constructive approach. The permutation is represented by an array of size n. The i'th element of this array shows the result of the permutation for rebec i. The algorithm starts with defining $\pi(i) = j$. Then it tries to find the other elements of the permutation. It is expected that i and j are rebecs of the same type. Therefore, they have equal number of known rebecs, which are also of similar types. Since the permutation must respect the known-rebec relation, it must also map the known rebecs of i to the known rebecs of j. It is assumed that $K(i)$ returns the ordered list of the indices of the known rebecs of rebec i. In the algorithm, $p1$ and $p2$ are the lists of rebec indices that must be checked for equivalency. Therefore, $K(i)$ and $K(j)$ are added to $p1$ and $p2$. Then at each step, one element from $p1$ and $p2$ are removed and checked against previous values of π. If π has another value then the algorithm returns false and terminates. If a new pair has been added to π, the indices of their known rebecs are added to $p1$ and $p2$. The algorithm continues until a contradiction is encountered, or there are no other rebecs to be checked. The return value shows whether they belong to the same equivalence class. As we explained in section 3, we can construct a communication graph of a Rebeca model. If this graph is not connected, the permutation of rebecs not connected to i and j, are not important to the equivalence of i and j. This algorithm in the worst case (i and j are equivalent), gives the answer in time linear in the number of rebecs in the system.

For finding the biggest equivalence classes of rebecs, which yields the most reduction in the state space, we first assume that each rebec by itself constitutes an equivalence class. Then at each step, we take representatives of two different equivalence classes, and check their equivalence. If they are equivalent, their corresponding equivalence classes can be combined. In the worst case (which is the case of an asymmetric system), every pair of rebecs of the same type are checked for equivalence. It means $O(n^2)$ times calling of check, which is in turn linear; and in total find is $O(n^3)$ in the number of rebecs in the system.

```
find ()
    classes := empty list;
    for every rebec r in the system add {r} to classes;
    for each m,n classes such that m != n  do
        if check (m.rep, n.rep) // check representatives of m and n
            replace m and n by the union of (m, n);
    od
end
```

The *find* algorithm computes the equivalence classes of rebecs in a Rebeca model. With these equivalence classes, the algorithms introduced in [17] or [3] can be used to model check Rebeca models while exploiting the symmetry of the models.

5 Dynamic Features in Rebeca

In this section, we deal with the dynamic features of a Rebeca model. Then we show that Theorem 1 applies to dynamic models, too. In a dynamic Rebeca model, rebecs may be created dynamically, i.e., during the execution of other rebecs. We allow the definition of variables of rebec type, which can hold references to rebecs (i.e., the index of the rebec). Due to dynamic creation, I is no longer fixed and (only) changes upon creation of new rebecs. We use $I(s)$ to denote the set of (indices of) rebecs in state s. In addition, rebec references can be passed as parameters to messages. Therefore, the set of rebecs that receive messages from a given rebec includes its known rebecs, plus the rebecs dynamically assigned to the rebec variables. Remember that the known rebec list of a rebec must be determined upon creation, and may not change during the execution.

5.1 Formal Semantics

The behavior of a dynamic Rebeca model $R = ||_{i \in I}\, r_i$, where I is the (dynamic) set of rebec indices, is defined as an indexed transition system $\langle S, A, T, s_0 \rangle$. The set of states S contains all the global states and s_0 is the initial state. The set of actions, which are indexed by indices from I, is denoted A, and is the set of all transitions. We use the notion $I(s)$ to mean the index set in state s. We may drop the s argument and just write I when s is irrelevant or clear from the context. By a (rebec) reference, we mean an index from the set I. The domain of rebec variables is $I \cup \bot$. Furthermore, like local variables, parameter queue variables can also be divided into two groups of normal parameters, and rebec parameters. The domain of normal parameters is D, and the domain of rebec parameters is I.

The apparent difference here is the introduction of some new sub-actions. Consider a transition $s \xrightarrow{a_j} t$ in a dynamic Rebeca model. In the following, the new sub-actions of a_j are introduced. In addition, the changes to some of the sub-actions with respect to Section 3 are also explained.

1. Assignment: Assignment to local variables of rebec index type is only possible in the form of $w \leftarrow z$ where w is a local variable and z is either a local variable or an argument of the containing message server. Both w and z take values from I, the set of rebec indices. As a result of this assignment, the value of z in state s is assigned to w in state t. Assignment to normal variables does not change compared to static models.

2. Rebec creation: A statement of the form '*new* $rc(kr_1, kr_2, \ldots, kr_m)$: $(p_1, \ldots, p_d)$', where rc is the name of a reactive-class, and kr_u represents an index from the current set I, and shows that u'th known rebec of the newly created rebec must be bound to r_{kr_u} , and p_u shows the u'th parameter to the

initial message. The execution of this sub-action in a_j, results in a new index v being added to I. This index is assigned to the newly created rebec. The effect of this new index is that the global state t, which is defined as $\prod_{i\in I} t_i$, will also include the local state t_v. The local state t_v of rebec r_v is defined in the same way as other rebecs, i.e. based on the variables and (message, sender and parameter) queues of the reactive-class rc. The valuation of the local variables of r_v in t is defined as follows. The message *initial* is placed in $r_v.m_1$, and the parameters p_1, p_2, ..., p_d are placed in $r_v.p_{11}$, $r_v.p_{21}$, ..., $r_v.p_{d1}$, respectively, and $r_v.s_1$ is assigned the value j (the index of its creator or parent). All other (local and queue) variables of r_v are undefined ($\perp$).

3. Send: In dynamic Rebeca models, messages can be sent both to known rebecs, and to local variables of rebec type. Like the case of a static model, the rebec r_j may send a message m to r_k with the parameters $n_1, \ldots, n_{h_k}$, where $m \in M_k$, and either k belongs to K_j or $r_j.v_g$ is a rebec variable and holds the value k. In addition, n_i may be a normal parameter ($n_i \in D$) or a rebec parameter ($n_i \in I$). In the case of a normal parameter, n_i can also be an expression that evaluates to some value from $D \setminus \perp$. However, in the case of a rebec parameter, n_i must be a local variable or an argument of the containing message server, and must be of rebec index type. This send operation, results in the message m being placed in the first empty slot of the queue of the receiving rebec. The result of sending $m(n_1, \ldots, n_{h_k})$ is: (recall that h_k is the number of parameter queues of r_k and for $i < h_k$, if n_i is $\perp$, then n_{i+1} must also be $\perp$):

If $\exists_{0<y\leq x_k}(r_k.m_y = \perp \wedge \forall_{0<z<y} r_k.m_z \neq \perp)$ then $r_k.m_y \leftarrow m, r_k.s_y \leftarrow j$, $\forall_{1\leq i\leq h_k} r_k.p_{iy} \leftarrow n_i$

Otherwise, x_k must be increased and the transition system of the Rebeca model cannot be constructed.

Passing a rebec reference as a parameter is treated the same as passing normal variables. Note again that the known rebecs of a rebec must be determined upon creation of that rebec.

5.2 Symmetry in Dynamic Rebeca Models

Detecting symmetry in the dynamic Rebeca models is possible in a similar way as in the static ones. Theorem 1 applies to dynamic Rebeca models without any changes. Note that Theorem 1 takes into consideration only the rebecs that are created in the initial state (and the known rebec relation among them). Theorem 2 carries over without any change to the extended setting.

Theorem 2. *If a permutation π preserves both rebec types and the known-rebec relation, and $\pi(s_0) = s_0$, then π is an automorphism of R.*

The interesting point is that since we made no changes to the theorem, the same algorithm is sufficient for detecting the symmetry in dynamic models.

6 Case Study

In this section, we give an example to show how our algorithm works. We use the 'load balancer' example from [7] with some changes. In this example, there are six identical clients that need some service, which is provided by three identical servers. Instead of communicating directly with the servers, the clients send their requests to load-balancers. The responsibility of the load-balancers is to distribute the load evenly among the servers. In our example, the round robin policy is used for load balancing, i.e., each load balancer sends the incoming requests to the servers in a round-robin manner. The servers, however, reply directly to the clients. In a static structure, the servers know all the clients beforehand; but in a dynamic model, the reference of the requesting client is passed to the server. The server uses that reference for sending its reply.

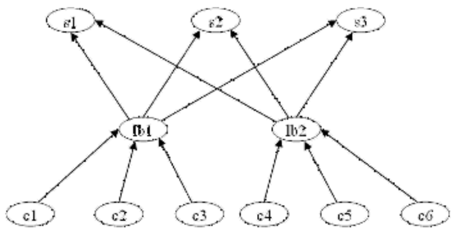


Fig. 2. The Load-Balancer

We first model it using only static features. In this case, the clients must be assigned a distinguishing identifier. This identifier is passed to their initial message server. Furthermore, all the clients are introduced to servers as known rebecs. The clients pass their identifier together with their request message, which is passed on by the load balancer. Thus the server knows to whom it should direct the answer.

The initialization of this system is shown below:

```
main {
    Client c1(lb1):(1),c2(lb1):(2),c3(lb1):(3),c4(lb2):(4),c5(lb2):(5),c6(lb2):(6);
    LoadBalancer lb1(s1,s2,s3):(), lb2(s1,s2,s3):();
    Server s1(c1,c2,c3,c4,c5,c6):(), s2(c1,c2,c3,c4,c5,c6):(),s3(c1,c2,c3,c4,c5,c6):();
}
```

In this model, the load-balancers and servers constitute two orbits. However, clients are not symmetric (each client adds up to one orbit). That is because of the symmetry-breaking identifiers passed to their initial message server. Using dynamic features of Rebeca, we can change the model, so that each load-balancer sends the reference of the sender of a request message to the servers. Therefore, the servers do not need to know the clients in advance. They just forward the reply to the rebec, whose reference is sent by the request.

The initialization of this system is shown below:

```
main {
    Client c1(lb1):(),c2(lb1):(),c3(lb1):(),c4(lb2):(),c5(lb2):(),c6(lb2):();
    LoadBalancer lb1(s1,s2,s3):(), lb2(s1,s2,s3):();
    Server s1():(), s2():(), s3():();
}
```

In the dynamic model, the clients also form one orbit. This shows that using dynamic features, we could model this example more naturally, which helps us find bigger orbits. This encourages the use of this technique in model checking symmetric Rebeca models. Figure 2 shows the static communication graph (defined in Section 3) of the dynamic load-balancer example.

7 Related Work

Symmetry reduction technique has been implemented in many model checking tools such as Murphi [13, 14] and SMC [17] and SPIN [4, 7]. Murphi is the first language (and tool) that provided support for symmetry reduction. If the modeled system is symmetric, the modeler must be aware of it, and use scalar sets properly to expose the symmetry of the system. SMC was developed by Sistla et.al., as a symmetry based model checker for verification of safety and liveness properties. SMC uses a notion of 'modules', which play the same role as scalar sets of Murphi. Other tools, like UPPAAL, SMV and SPIN, use the approach of Murphi for handling symmetry; namely, adding scalar sets to expose the symmetry of the system by the modeler. Using scalar-sets (or modules in SMC) is error prone and sometimes makes modeling a symmetric system more difficult. In our approach, no change to the syntax of Rebeca is made, and therefore the modeler does not need to know about the symmetry of the system. Instead, the symmetry in a Rebeca model is automatically detected, if there is any.

The work of [7] is similar to ours, in the sense that static graphs of channels are used to detect the symmetry automatically from (dynamic) Promela models. The dynamicity in their models is caused by sending the channels around. They do not consider the dynamic creation of processes. In our approach, rebec references (which can be interpreted as their inbox address) can be sent around, and rebecs can also be created dynamically.

8 Conclusions

Rebeca is an object based language for modeling and verification of reactive systems. Since rebecs of the same type show similar behavior, it is easy to find symmetry in Rebeca models. We showed in this paper that if the static communication graph of a Rebeca model is symmetric, then the whole model is symmetric. Furthermore, an algorithm is presented for solving the orbit problem for Rebeca models in polynomial time. The algorithm finds the orbits of rebecs by examining the known-rebec relation that defines the composition of the system. In contrast to most other symmetry-related tools, no new construct needs to be added to the syntax of Rebeca to be used by this algorithm. The same algorithm still works when dynamic features, such as 'the dynamic creation of rebecs' and 'the dynamic changing of topology', are added to the models. As a result, the symmetry reduction technique can be efficiently implemented in current Rebeca model checkers [16], or in the direct model checking of Rebeca.

References

1. P. A. Abdulla, B. Jonsson, M. Kindahl, and D. Peled. A general approach to partial order reductions in symbolic verification. In *Proceedings of CAV'98*, pages 379–390, 1998.
2. G. Agha. The structure and semantics of actor languages. In *Proceedings of the REX Workshop*, pages 1–59, 1990.
3. D. Bosnacki. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In *Proceedings of the SPIN Workshop*, pages 89–103, 2003.
4. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. *Software Tools for Technology Transfer*, 4(1):92–106, 2002.
5. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Proceedings of CAV'98*, pages 147–158, 1998.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
7. A. Donaldson, A. Miller, and M. Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. In *Proceedings of AVOCS'04*, pages 161–177, 2005.
8. E. Emerson and A. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1–2):105–131, 1996.
9. E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In *Proceedings of TACAS '97*, pages 19–34, 1997.
10. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In *Proceedings of CAV'95*, pages 309–324, 1995.
11. E. A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In *Proceedings of CHARME'03*, pages 216–230, 2003.
12. C. Hewitt. Procedural embedding of knowledge in planner. In Proceedings of IJCAI'71, pages 167–184, 1971.
13. C. Ip and D. Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.
14. C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murphi. In *Proceedings of CAV'96*, pages 147–158, 1996.
15. M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamamenta Informaticae*, 63(4):385–410, 2004.
16. M. Sirjani, A. Shali, M. M. Jaghoori, H. Iravanchi, and A. Movaghar. A front-end tool for automated abstraction and modular verification of actor-based models. In *Proceedings of ACSD'04*, pages 145–150, 2004.
17. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering Methodology*, 9(2):133–166, 2000.
18. A. P. Sistla. Employing symmetry reductions in model checking. *Computer Languages, Systems & Structures* 30(3-4):99–137, 2004.