Modeling and Efficient Verification of Broadcasting Actors

Behnaz Yousefi, Fatemeh Ghassemi, Ramtin Khosravi

Modeling and Efficient Verification of Broadcasting Actors

Behnaz Yousefi, Fatemeh Ghassemi, Ramtin Khosravi

School of Electrical and Computer Engineering, University of Tehran, Iran
{b.yousefi, fghassemi, r.khosravi}@ut.ac.ir

Abstract. Many distributed systems use broadcast communication for various reasons such as saving energy or increasing throughput. However, the actor model for concurrent and distributed systems does not directly support this kind of communication. In such cases, a broadcast must be modeled as multiple unicasts which leads to loss of modularity and state space explosion for any non-trivial system. In this paper, we extend Rebeca, an actor-based model language, to support asynchronous anonymous message broadcasting. Then, we apply counter abstraction for reducing the state space which efficiently bypasses the constructive orbit problem by considering the global state as a vector of counters, one per each local state. This makes the model checking of systems possible without further considerations of symmetry. This approach is efficient for fully symmetric system like broadcasting environments. We use a couple of case studies to illustrate the applicability of our method and the way their state spaces are reduced in size.

Keywords: state space reduction, broadcast, Rebeca, actor-based language, model checking, verification

1 Introduction

The actor model [2,13] is one of the pioneers in modeling of concurrent and distributed applications. It has been introduced as an agent-based language by Hewitt [13] and then extended by Agha as an object-based concurrent computation model [2]. An actor model consists of a set of actors that communicate through asynchronous message passing. Communication in actor models is based on unicast, i.e. in each message the receiver has to be specified. On the other hand, broadcast communication is a simple model of parallel computation [28] and a large number of algorithms in distributed networks use broadcast, such as consensus agreement [20,4,7,5,22], leader election [23,16,21], and max finding [9,17,27,18]. In addition, wireless channels have a broadcast nature as when a node sends a message, it can be received by any other node that lies within its communication range, which leads to power saving and throughput improvement [8]. Modeling these algorithms with actor model would cause some complexities both in modeling and analysis. In the modeling aspect, a broadcast has to be replaced with multiple unicasts, which leads to loss of modularity and cluttering of the model code. The (unnecessary) interleaving of these unicast messages

causes state space explosion during analysis, the main obstacle in model checking of nontrivial systems. Using broadcasts instead of multiple unicasts, enables efficient use of counter abstraction technique [3] to overcome this problem.

In this paper, we extend the actor-based modeling language Rebeca [31] with broadcast communication. Rebeca is an operational interpretation of the actor model with the aim of bridging the gap between formal verification techniques and the real world software engineering of concurrent and distributed applications. This is achieved by its simple Java-like syntax and extensive tool support [1,32], including a modeling environment and a model checker employing well known reduction techniques [14]. The resulting modeling language provides a suitable framework to model mobile ad hoc networks (MANETs). Having broadcast as the main communication mechanism in the broadcasting actor language, we have applied counter abstraction to efficiently reduce the size of the state space. To the best of our knowledge, there is no actor-based language with direct support for broadcast communication. In [29], Rebeca is extended with *components* to provide a higher level of abstraction and encapsulation and broadcast has been used for communication between the components of actors and not within a component.

In the original actor model, message delivery is guaranteed and each actor has a mailbox to maintain messages while it is busy processing another message. However, due to unpredictability of networks, the arrival order of messages are assumed to be arbitrary and unknown [2]. To prevent state space explosion, Rebeca makes use of FIFO queues as a means of message storage [30] so that messages will be processed based on the order that they have been received. In our extended model, queues are replaced by *bags* (unordered multi-sets of messages).

The paper is structured as follows. Section 2 briefly introduces Rebeca and provides an overview on the counter abstraction technique. Section 3 presents our extension to Rebeca to support broadcast. In Section 4, we show how we have implemented counter abstraction to generate the state space compactly. To illustrate the applicability of our approach, we bring two case studies in Section 5. Finally, we review some related work in 6 before concluding the paper.

2 Preliminaries

2.1 Rebeca

Rebeca [31] is an actor- based modeling language which has been proposed for modeling and verification of concurrent and distributed systems. It aims to bring the formal verification techniques into the real world of software engineering by means of providing a Java-like syntax familiar to software developers and also providing tool support via an integrated modeling and verification environment [1]. A design principle behind Rebeca is to enable domain-specific extensions of the core language [30]. Examples of such extensions has been introduced in various domains such as probabilistic systems [33], real-time systems [25], and software product lines [26].

In Rebeca, actors are the units of computation, called rebecs (short for reactive objects) which are instances of the defined *reactive classes* in the model. Rebecs communicate with other rebecs only through message-passing which is fair and asynchronous. A rebec can send messages only to its *known rebecs* mentioned in its definition and also to itself using "self" keyword. The local state of a rebec is represented by its *state variables* as well as the contents of its message queue. The *message servers*, which indicate how received messages must be processed, are also other parts of a rebec definition. Each rebec has at least one message server called "initial" which acts as a constructor in object-oriented language and is responsible for initialization tasks, and it is always put in every rebec's queue initially.

A rebec is *enabled* if and only if there is at least one message in its queue. The computation takes place by removing a message from the head of the queue and executing its corresponding message server atomically, after which the rebec proceeds to process the next message in the queue (if exists). Processing a message may have the following consequences:

- the value of the state variables of the executing rebec may be modified,
- new rebecs may be created,
- some messages may be sent to other rebecs or the executing rebec itself.

Besides the definition of the reactive classes, the *main* part of a Rebeca model specifies the instances of the reactive classes initially created along with their known-rebecs. The parameters of initial message server, if there is any, will also be specified.

As an example, Fig.1 illustrates a simple leader election algorithm modeled in Rebeca, aiming to select a node with the highest id as the leader. The nodes are organized in a (directed) ring. Each node sends its id to its neighbor and upon receiving a message compares the received id with its own id. If it is greater than its own id, it passes the number to its neighbor. So, when a number passes through the ring and is received by the node which its id is equal to the received id, it means that node has the greatest id and will be elected as the leader.

2.2 Counter Abstraction

When analyzing complex systems, their state space is prone to grow exponentially in space, known as the state space explosion problem, which is common in the realm of model checking. Counter abstraction is one of the proposed approaches to overcome this difficulty [24,3]. Its idea is to record the global state of a system as a vector of counters, one per local state, tracking how many of the n components currently reside in that local state. In our work, "components" refer to actors in the system. Let n and m be the number of components and local states respectively. This technique turns the n-component model of a size exponential in n, i.e., m^n, into one of a size polynomial in n, i.e., $\binom{n+m-1}{m}$. Counter abstraction can be seen as a form of symmetry reduction [10]. Two

```
reactiveclass Node
{
  knownrebecs
  {
    Node neighbour;
  }
  statevars
  {
    boolean isLeader;
    int myInt;
  }
  msgsrv initial(int num)
  {
    isLeader = false;
    myInt = num;
    neighbour.receiveInt(myInt);
  }
  msgsrv receiveInt(int num)
  {
    if (num > myInt)
      neighbour.receiveInt(num);
    if (num == myInt)
    {
      isLeader = true;
      self.isLeader();
    }
  }
  msgsrv isLeader()
  {
    // elected as the leader,
    // continue the
         computation
  }
}

main
{
  Node node0(node1):(1);
  Node node1(node2):(2);
  Node node2(node0):(3);
}
```

Fig. 1: Simple leader election algorithm: an example of a Rebeca model

global states S and S' are identical up to permutation if for every local state s, the same number of components reside in s is the same in the two states S and S', only the order of elements change through permutation. For example, consider a system which consists of 3 components each with only one variable v_i of type of Boolean. The global states of (T, T, F), (F, T, T) and (T, F, T) are equivalent and can be represented as $(2T, F)$.

3 Broadcasting Rebeca

In this section, we present a modeling language based on Rebeca, by replacing the unicast communication mechanism by broadcast. We name the language *bRebeca* and will describe its syntax and formal semantics in the following subsections.

3.1 Syntax

In bRebeca, rebecs communicate with each other only through broadcasting: the sent message will be received by all rebecs of the model (as specified in the main part). After taking a message from its bag, the receiving rebec simply discards the message if no corresponding message server is defined in its reactive class. Since every message will be received by all existing rebecs, unlike Rebeca, there is no need for declaring the known rebecs in the reactive class definitions. Furthermore, there is no need to specify the receiver of a message in a send

statement. Every `initial` message server at least have one parameter, named `starter`. The value of `starter` is only true for the rebec which initiates the algorithm by broadcasting the first message.

The grammar of bRebeca is presented in Fig. 2.

$$
\begin{aligned}
\text{Model} &::= \text{ReactiveClass}^{+}\ \text{Main}\\
\text{Main} &::= \textsf{main}\ \{\text{RebecDecl}^{+}\ \}\\
\text{ReactiveClass} &::= \textsf{reactiveclass}\ C\ \{\ \text{StateVars}\ \text{MsgServer}^{*}\ \}\\
\text{StateVars} &::= \textsf{statevars}\ \{\ \text{VarDecl}^{*}\ \}\\
\text{MsgServer} &::= \textsf{msgsrv}\ M(<T\ V>^{*})\ \{\ \text{Statement}^{*}\ \}\\
\text{VarDecl} &::= T\ V;\\
\text{Statement} &::= \text{Assign} \mid \text{Broadcast} \mid \text{Conditional}\\
\text{Assign} &::= V = Expr;\\
\text{Broadcast} &::= M(<V>^{*});\\
\text{Conditional} &::= \textsf{if}\ (Expr)\{\ \text{Statement}^{*}\ \}\ \textsf{else}\ \{\ \text{Statement}^{*}\ \}\\
\text{RebecDecl} &::= C\ R(<V>^{*});
\end{aligned}
$$

Fig. 2: bRebeca language syntax: Angle brackets (< >) are used as meta-parentheses. Superscript ? denotes that preceding part is optional, superscript + is used for more than once repetition, and * indicates the zero or more times repetition. The symbols C, T, M, and V denote class, type, method and variable names respectively. The symbol E denotes an expression, which can be an arithmetic or Boolean expression.

3.2 Semantics

The formal semantics of bRebeca is expressed as a labeled transition system (LTS), defined by the quadruple $\langle S, \rightarrow, L, s_0\rangle$, where S is a set of states, $\rightarrow$ a set of transitions between states, L a set of labels, and s_0 the initial state.

Let I denote the set of all existing rebec identifiers, ranged over $1..n$, V a set of all possible values for the state variables, and M the set of all message servers identifiers in the model. All rebecs of the model which execute concurrently form a closed model $R = \|_{i\in I} r_i$. Each rebec with identifier $i \in I$, is described by a tuple $r_i = \langle V_i, M_i\rangle$, where V_i is the set of its state variables and M_i the set of messages it can respond to. As said earlier, a rebec in bRebeca holds its received messages in an unordered bag (unlike Rebeca, in which maintains such messages in a FIFO queue).

Definition 1. *(Local State) The local state of a rebec r_i is an element of $S_i = Val_i \times Bag_i$, where Val_i is the set of all valuations for the state variables of r_i*

(functions from V_i into V), and Bag_i is the set of all possible bags of messages to rebec r_i.

Definition 2. *(Global State) A global state S is defined as the combination of all local states of rebecs in the model:*

$$S = \prod_{i \in I} S_i$$

An initial state s_0 consists of all rebecs initial state, where all rebecs have executed their initial messages. In fact, an initial message server can be seen as a constructor in object-oriented languages.

To formally define the transitions between the states, we assume there are two sets of auxilary functions defined as follows:

- $Update_i(v_i, m) : V_i \times M_i \to V_i$ receives a valuation v_i for the state variables of r_i and a message m, and returns the updated valuation for the state variables of r_i. This function abstracts the effect of processing m by r_i on its state variables.
- $Sent_i(m) : M_i \to M$ specifies the set of messages broadcasted by r_i as a result of processing message m.

To keep our semantic description simple, we do not give the details of the above two functions, since the semantics of control statements and expressions are the same as those in Rebeca. We also ignore the parameters in the messages. For a detailed semantic description, the reader may refer to [31].

Definition 3. *(Transition Relation) Let L denote to the set of all messages which can be passed between rebecs in the model R, $L = \bigcup_{i \in I} M_i$. Also assume $s = \langle s_1, s_2, \ldots, s_n \rangle$ and $s' = \langle s'_1, s'_2, \ldots, s'_n \rangle$ be two states in S such that $s_i = \langle v_i, b_i \rangle$ and $s'_i = \langle v'_i, b'_i \rangle$. The transition relation $\to \subseteq S \times L \times S$ is defined such that $(s, m, s') \in \to$ (written as $s \xrightarrow{l} s'$) if and only if*

$$\begin{aligned} \exists i \in I \cdot\ & m \in b_i \wedge \\ & v'_i = Update_i(v_i, m) \wedge \\ & b'_i = b_i \cup Sent_i(m) - \{m\} \wedge \\ & (\forall j \in I, j \neq i \cdot v'_j = v_j \wedge b'_j = b_j \cup Sent_i(m)). \end{aligned}$$

Note that in the definition above, the operators $\cup$ and $-$ are assumed to be applied to message bags, which are multi-sets. In other words, if m is already in b, $\{m\} \cup b$ adds another copy of m to b, and $b - \{m\}$ removes one instance of m from b.

4 Implementing Counter Abstraction

To apply the counter abstraction technique on bRebeca, we consider each global state as a vector of counters, one per each distinct local state, and keep track of how many rebecs have that local state: the same state variable values and bag.

The reduction takes place on-the-fly while constructing the state space. Whenever a new global state is reached, we create a temporary global state by comparing the local states and count how many are equal. Then the temporary global state is compared with existing temporary global states and if it is a new state, then it would be added to the set of the reached states. Fig. 3 shows an example of applying counter abstraction on a global state. The global state consists of three rebecs with only one state variable i and one update message The local states, i.e. the state variables and bags of rebecs r1 and r2 are equal and they can be considered the same while ignoring their identities.

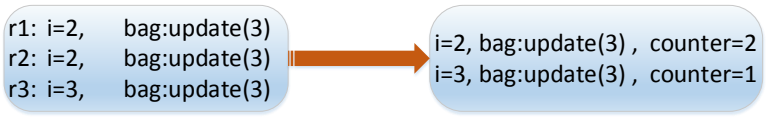

Fig. 3: Example of applying counter abstraction on a global state

To implement the state space builder, we follow a model-driven approach implemented in C#. A reusable layer of abstract classes are defined to provide the basic entities and services needed to generate the state space, such as *State* and *Message*. Also, the basic mechanism of generating the state space using a Depth-First Search (DFS) is implemented in this layer (named *StateSpaceBuilder*). The DFS is implemented in a multi-threaded way to exploit multi-core processing power. This class takes the responsibility of handling nondeterminism in message scheduling: in each state, a separate task is created for scheduling each *enabled* rebec with distinct local state where rebecs are fairly run by keeping track of rebecs in each state.

The proposed model supports only broadcast communications, as rebecs have no ids to be distinguished from each other it would be impossible to add unicast to such a model. Though with some minor modifications of current model, as follows, very limited unicast would be feasible. A node can unicast a message only to itself or the sender of prepossessing message, otherwise it will jeopardize the soundness of counter abstraction technique by considering two states equal while they are not. In the other words, only relative addresses (i.e. the sender of a message) are allowed as ids have no absolute meaning.

We need to keep somehow the ids of rebecs instead of just counters in some level of state space exploration. While constructing the state space, by processing messages, we keep the ids of those rebecs together in a group which have the same local state, regardless of their ids. The resulting global state is called `middle global state`.To store reachable states, the `final global state`s are computed from `middle global state`s by counting the number of ids in each group.

To make unicast unconditionally possible we need to consider the permutation of ids and use the known-rebec concept like symmetry reduction[14].

For processing a bRebeca model, we use a bRebeca parser to generate a C# code from the source model. Each reactive class is translated into two basic classes: one subclass of *State* to represent the local states of the rebecs of that class, and another class which holds the implementation of the message servers. A translated message server, when executed, generates all possible "next states". Note that due to the existence of "non-deterministic assignment" in Rebeca, there may be more than one next state.

When generating a subclass of *State* for a rebec, the code generator is responsible for implementing an abstract *hash function*, which is used to compare the local states. This is essential to implement an efficient comparison for global states which is encoded as a mapping from (the polymorphic) State into integers, implemented by a Dictionary class in C#.

5 Case Studies

To illustrate the applicability of the proposed modeling language, two algorithms are modeled and the amount of reduction in state space size is shown.

5.1 Max-algorithm

Consider a network where every node stores an integer value and they are going to find the maximum value in the network in a distributed manner. The bRebeca code for the model is illustrated in Fig.4 for a simple network of three nodes.

The nodes in the network are modeled by the reactive class `Node` with two state variables `my_i` to store the node value and `done` which indicates whether the node has already sent its value or there is no need to send it, based on the values of the other nodes received so far. The goal is to find the maximum value of `my_i` among nodes. One node initiates the algorithm by broadcasting its value to other nodes. Upon receiving a value from others, each node compares its value with the received one. If its value is less than the received one, it updates its value to the received one and waits for receiving the next messages while giving up on sending by setting its `done` to true. Otherwise, it broadcasts its value to other rebecs, if it has not already and then sets its `done` to true. The algorithm terminates when there is no further message to be processed. It means that everyone has either transmitted its value or given up. In this case, all state variables have been updated to the maximum value. This algorithm is referred to as "Max-Algorithm" [17].

For a network consists of three rebecs, if we start with rebec `rebec2` which has the maximum value (3), each node gives up transmitting after receiving the maximum and procedure has only one step. The reduced state space obtained from the execution of max-algorithm in this network is shown in Fig.5.

```
1  reactiveclass Node
2  {
3    statevars
4    {
5        int my_i;
6      boolean done;
7    }

9    msgsrv initial(int j,
         boolean starter)
10   {
11       my_i = j;
12     if(starter) {
13       done = true;
14       send(my_i);
15     } else
16       done = false;
17   }

19   msgsrv send(int i)
20   {
21     if (i < my_i) {
22       if (!done) {
23         done = true;
24         send(my_i);
25       }
26     } else {
27       my_i = i;
28       done = true;
29     }
30   }
31 }
32 main
33 {
34       Node rebec0(1,false);
35       Node rebec1(2,false);
36       Node rebec2(3,true);
37 }
```

Fig. 4: Max-algorithm with 3 nodes

5.2 Leader Election

One way of electing leader in a distributed network is through flipping a coin [11]. The algorithm consists of several rounds. In each round, all competitors, nodes with coin value of true, participate by flipping their coin and broadcasting the observed results to the others. A round is completed whenever all competitors have flipped their coin and received other nodes' observation. At the end of each round:

- If there is only one node with coin value of true, then it is elected as the leader.
- If there is no node with coin value of true, it means that the round should be repeated. So all coin values of the previous competitors will be set to true.
- If the number of nodes with coin value of true is more than one, nodes with coin value of true will participate in the next round and flip the coin again.

Fig. 6 shows one execution scenario of the leader election algorithm in a network of five nodes.

The bRebeca code for this algorithm is represented in Fig. 7. There are two reactive classes, named `Node` and `Barrier`, in the model. We use `Barrier` to synchronize nodes before starting a new round, in order to prevent mixing up the messages between different rounds. The `Barrier` is to make sure that the current round is completed, and all nodes are aware of each other observation, and ready to start a new round. Reactive class `Barrier` has only one state variable which counts the number of nodes has completed their round. Whenever all nodes

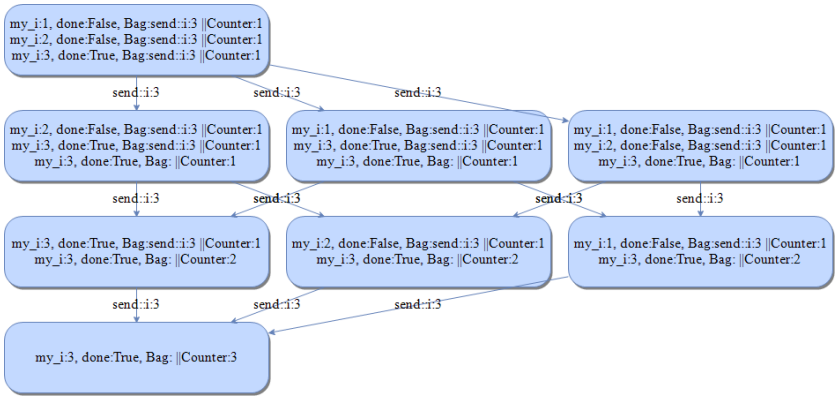

Fig. 5: max-algorithm state space

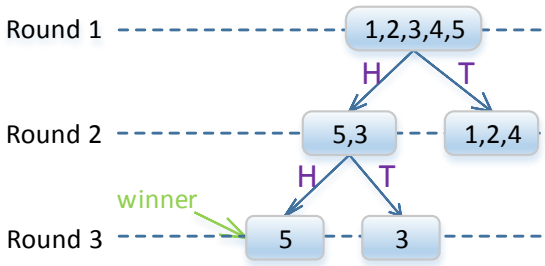

Fig. 6: An execution scenario of leader election algorithm with five nodes

complete their round, it would broadcast `start_next_round` to all nodes to start a new round.

In reactive class `Node`, two state variables `head` and `tail` are used to store the number of heads and tails have been observed in the current round. The state variable `comp` indicates whether the node was a competitor in the previous round. Therefore if the number of competitors in the current round is equal to zero, no head observed in the previous round, we would be able to repeat the round by restoring the previous competitors (lines 32-35). In every round if currently there is more than one competitor and node is one them, its coin value is true, it flips its coin and after updating its counters accordingly broadcasts the result to the other nodes (lines 31-45). The number of previous and current competitors also need to be saved. Each node needs to keep the number of current competitors to decide when it has completed its round so it can inform the `Barrier` (line 46). The number of previous competitors is needed to specify when we can clean our counters and move to the next round (lines 25-30). Note

that as the delivery order of messages is not guaranteed, a node may process its `rec_coin` message before `start_next_round` message. Hence, both message servers must check whether the counter variables, such as `head` and `tail`, belong to the previous round and need to be reset before using them.

```
reactiveclass Node {
  statevars {
    int head;
    boolean my_coin;
    int tail;
    int current_comp;
    int prev_comp;
    boolean comp;
    boolean is_leader;
  }

  msgsrv initial(boolean starter) {
    if(starter==true)
      start_next_round();
    my_coin=true;
    head=3;
    tail=0;
    current_comp=3;
    prev_comp=3;
    comp=true;
    is_leader=false;
  }

  msgsrv start_next_round() {
    if (head+tail == prev_comp) {
      prev_comp = current_comp;
      current_comp = head;
      head = 0;
      tail = 0;
    }
    if (current_comp != 1 && comp) {
      if (current_comp == 0) {
        my_coin = true;
        current_comp = prev_comp;
      }
      if (my_coin) {
        int ch=?(0,1);
        if (ch == 0) {
          my_coin = false;
          tail = tail+1;
        } else {
          my_coin=true;
          head=head+1;
        }
        rec_coin(my_coin);
        if (head+tail == current_comp)
          rec_barrier();
      } else
        comp=false;
    }
    if (my_coin && current_comp == 1)
      is_leader = true;
  }

  msgsrv rec_coin(boolean c) {
    if (head + tail == prev_comp) {
      prev_comp = current_comp;
      current_comp = head;
      head = 0;
      tail = 0;
    }
    if (c)
      head = head+1;
    else
      tail=tail+1;
    if(head+tail == current_comp)
      rec_barrier();
  }
}

reactiveclass Barrier {
  statevars {
    int barrier;
  }

  msgsrv initial(boolean starter) {
    barrier=0;
  }

  msgsrv rec_barrier() {
    barrier = barrier + 1;
    if (barrier == 3) {
      start_next_round();
      barrier=0;
    }
  }
}
main {
  Node rebec0(true);
  Node rebec1(false);
  Node rebec2(false);
  Barrier bar(false);
}
```

Fig. 7: Leader election algorithm

Table 1 compares the number of states resulted with and without applying counter abstraction. Note that with increasing the number of nodes, the opportunity of collapsing nodes together grows.

Table 1: Comparing the state spaces size with and without applying counter abstraction

	No. of nodes	No. of states	No. of states with reduction
Max-Algorithm	3	64	53
	4	3216	1675
	5	719,189	185,381
Leader election	3	3792	752
	4	308,553	15,905
	5	$>$ 1,200,000	521,679

6 Related Work

In order to avoid state space explosion, different approaches have been proposed such as symbolic model checking [19], symmetry reduction [6], partial order reduction [12] and counter abstraction [3].

Counter abstraction has been studied in several other works (e.g., in [10,3,24]). The proposed approach in [24] aims to abstract an unbounded parameterized system into a finite-state system and then verify various liveness properties efficiently. The authors in [24] use limited abstracted variables to count for each local state of a process how many processes currently reside in. However, counters were saturated at a static value of c, meaning that c or more processes are at local state s. In [3], counter abstraction is used to achieve efficiency in BDD-based symbolic state space exploration of concurrent Boolean programs while unlike [24] it makes use of exact counters where in a global state only non-zero counters are stored. The idea of counting have also been used in [15] to record the local states of a biological system, in which each local state is represented as a vector of counters, each element denotes to the corresponding number of species.

In this paper counters are unbounded, similar to [3], to show the exact number of Rebecs having the specific local state, and abstracted local states are not limited either.

As mentioned before, there are several techniques for reducing the state space such as symmetry reduction which aims to reduce the state space by partitioning the state space into equivalence classes which are represented by one state[6] as their representative. Since finding the unique representative of state while exploring the state space, known as *constructive orbit problem*, is NP-Hard [6], some heuristics have been proposed to avoid this problem, which may result in multiple representatives. In [14], a polynomial-time heuristic solution is proposed to exploit symmetry in Rebeca, computing a representative state using the on-the-fly approach takes $O(n^4)$ in the worst-case. The complexity of the proposed algorithm is due to the role of "known rebecs" that should be pre-

served during the permutation. Since in the broadcast environment there is no notion of "known rebecs" and the system is fully symmetric, we can skip paying such a price by applying counter abstraction which is suitable for such systems. The complexity of finding the equivalent of each state is linear in the number of states.

7 Conclusion

In this paper we extended the syntax and semantics of Rebeca to support broadcast efficiently. On one hand, it makes modeling easier, by replacing a set of unicast statements by a single broadcast statement, there is no need to define each rebec as a known-rebec to every other rebec to make the broadcast possible. On the other hand, as all rebecs instantiated from one reactive class are identical, their indexes are irrelevant and can be ignored while constructing the state space. This property makes counter abstraction applicable which is more efficient in fully symmetry systems as discussed in section 6.

The broadcasting actors model provides a suitable framework to model wireless sensor (WSNs) and mobile ad hoc networks (MANETs). In these networks, broadcast is restricted by locality of nodes, meaning that a node receives a message if it is located close enough to a sender, so called connected. Connectivity of nodes defines the topology concept which should be modeled as a part of semantics. Due to energy consumption of nodes and their mobility, the underlying topology changes arbitrary. Therefore, to address local broadcast and topology changes, bRebeca can be extended at the semantics level to allow verification of WSNs and MANETs. To this aim, we pair the global state with the topology of networks and generate the state space for permutations of a topology. We merge states with identical structures of topology while applying counting abstraction which makes automatic verification of such networks susceptible.

References

1. Rebeca formal modeling language. http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/Afra
2. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
3. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Computer Aided Verification. pp. 64–78. Springer (2009)
4. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM 32(4), 824–840 (1985)
5. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Advances in Cryptology. pp. 524–541. Springer (2001)
6. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Computer Aided Verification. pp. 147–158. Springer (1998)
7. Correia, M., Veronese, G.S., Neves, N.F., Veríssimo, P.: Byzantine consensus in asynchronous message-passing systems: a survey. IJCCBS 2(2), 141–161 (2011)

8. Cui, T., Chen, L., Ho, T.: Distributed optimization in wireless networks using broadcast advantage. In: Decision and Control. pp. 5839–5844. IEEE (2007)
9. Dechter, R., Kleinrock, L.: Broadcast communications and distributed algorithms. Trans. Computers 35(3), 210–219 (1986)
10. Emerson, E.A., Trefler, R.J.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In: Correct Hardware Design and Verification Methods. pp. 142–156. Springer (1999)
11. Fill, J.A., Mahmoud, H.M., Szpankowski, W.: On the distribution for the duration of a randomized leader election algorithm. Ann. Appl. Probab pp. 1260–1283 (1996)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
13. Hewitt, C.: Viewing control structures as patterns of passing messages. Artif. Intell. 8(3), 323–364 (1977)
14. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Informatica 47(1), 33–66 (2010)
15. Katoen, J.: Model checking: One can do much more than you think! In: Fundamentals of Software Engineering. pp. 1–14. Springer (2011)
16. Larrea, M., Raynal, M., Arriola, I.S., Cortiñas, R.: Specifying and implementing an eventual leader service for dynamic systems. IJWGS 8(3), 204–224 (2012)
17. Levitan, S.P., Foster, C.C.: Finding an extremum in a network. In: 9th International Symposium on Computer Architecture. pp. 321–325. ACM (1982)
18. Martel, C.U.: Maximum finding on a multiple access broadcast network. Inf. Process. Lett. 52(1), 7–15 (1994)
19. McMillan, K.L.: Symbolic model checking. Kluwer (1993)
20. Melliar-Smith, P.M., Moser, L.E., Agrawala, V.: Broadcast protocols for distributed systems. Trans. Parallel Distrib. Syst. 1(1), 17–25 (1990)
21. Mostéfaoui, A., Raynal, M., Travers, C.: Crash-resilient time-free eventual leadership. In: 23rd International Symposium on Reliable Distributed Systems. pp. 208–217. IEEE Computer Society (2004)
22. Okun, M., Barak, A.: Efficient algorithms for anonymous byzantine agreement. Theory Comput. Syst. 42(2), 222–238 (2008)
23. Ostrovsky, R., Rajagopalan, S., Vazirani, U.V.: Simple and efficient leader election in the full information model. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing. pp. 234–242. ACM (1994)
24. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infty)-Counter Abstraction. In: Proceedings of the 14th International Conference on Computer Aided Verification. pp. 107–122. CAV '02, Springer-Verlag, London, UK, UK (2002)
25. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci. Comput. Program. 89, 41–68 (2014)
26. Sabouri, H., Khosravi, R.: Delta modeling and model checking of product families. In: Fundamentals of Software Engineering. pp. 51–65. Springer (2013)
27. Shiau, S., Yang, C.: A fast maximum finding algorithm on broadcast communication. In: Computing and Combinatorics. pp. 472–481. Springer (1995)
28. Shiau, S., Yang, C.: A fast sorting algorithm and its generalization on broadcast communications. In: Computing and Combinatorics. pp. 252–261. Springer (2000)
29. Sirjani, M., de Boer, F.S., Movaghar, A., Shali, A.: Extended Rebeca: A component-based actor language with synchronous message passing. In: Fifth International Conference on Application of Concurrency to System Design. pp. 212–221. IEEE Computer Society (2005)

30. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Formal Modeling: Actors, Open Systems, Biological Systems. pp. 20–56. Springer (2011)
31. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. 63(4), 385–410 (2004)
32. 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: 4th International Conference on Application of Concurrency to System Design. pp. 145–150. IEEE Computer Society (2004)
33. Varshosaz, M., Khosravi, R.: Modeling and verification of probabilistic actor systems using pRebeca. In: Formal Methods and Software Engineering. pp. 135–150. Springer (2012)