Software ENgineering

Extending Rebeca with synchronous messages and reusable components

M. Sirjani, F.S. de Boer, A. Movaghar, A. Shali

REPORT SEN-R0505 FEBRUARY 2005

ISSN 1386-369X

Extending Rebeca with synchronous messages and reusable components

ABSTRACT
In this paper, we propose extended Rebeca as a tool-supported actor-based language for modeling and verifying of concurrent and distributed systems. We enrich Rebeca with a formal concept of components which integrates the message-driven computational model of actor-based languages with synchronous message passing. Components are used to encapsulate a set of internal active objects which react asynchronously to messages by means of methods and which additionally interact via a synchronous message passing mechanism. Components themselves interact only via asynchronous and anonymous messages. We present our compositional verification approach and abstraction techniques, and the theory corresponding to it, based on formal semantics of Rebeca. These techniques are exploited to overcome state explosion problem in model checking.

2000 Mathematics Subject Classification: 68N19
1998 ACM Computing Classification System: D.2.4 ; D.3.1 ; F.3.2
Keywords and Phrases: the actor model; reactive systems; Rebeca; components; modular verification

Extending Rebeca with Synchronous Messages and Reusable Components

Marjan Sirjani[a,b] Frank de Boer[b] Ali Movaghar[a] Amin Shali[c]
[a]Department of Computer Engineering
Sharif University of Technology
Azadi Ave., Tehran, Iran
[b]Department of Software Engineering
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands
[c]Department of Electrical and Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
msirjani@cwi.nl *f.s.de.boer@cwi.nl* *movaghar@sharif.edu* *shali@ece.ut.ac.ir*

Abstract. In this paper, we propose extended Rebeca as a tool-supported actor-based language for modeling and verifying of concurrent and distributed systems. We enrich Rebeca with a formal concept of components which integrates the message-driven computational model of actor-based languages with synchronous message passing. Components are used to encapsulate a set of internal active objects which react asynchronously to messages by means of methods and which additionally interact via a synchronous message passing mechanism. Components themselves interact only via asynchronous and anonymous messages. We present our compositional verification approach and abstraction techniques, and the theory corresponding to it, based on formal semantics of Rebeca. These techniques are exploited to overcome state explosion problem in model checking.
Keywords: the actor model, reactive systems, Rebeca, component, modular verification.

1 Introduction

With the increasing use of concurrent and distributed systems, establishing a reliable approach in developing such systems is a critical problem. This problem can only be solved in a rigorous manner by verification methods which are based on a formal semantics. In a formal verification approach, we need a modeling language to represent the behavior of the system, a specification language to embody the required properties, and an analysis method to verify the behavior against the required properties.

Despite the recent successful application of formal methods, further research is still required. One of the major problems in practice is the difference in abstraction level and the resulting semantic gap between the modeling and programming languages used by software designers and programmers and the modeling languages used by the verification tools. Deductive approaches in formal verification are mostly based on mathematical oriented languages which cannot be easily used by software engineers, and modeling languages supported by model checker tools usually are not very close to programming languages used by practitioners. On the other hand, languages used by practitioners, as modelers or programmers, are too informal or too heavy to be analyzed. Another problem lies in the analysis methods. Deductive methods need a high expertise and interaction with theorem provers, and model checkers suffer from the state explosion problem, when the number of system components grows.

Rebeca ($\underline{Re}active\ O\underline{b}j\underline{ec}ts\ L\underline{a}nguage$) is an actor-based language with a formal foundation, introduced in [15, 17] and which is designed in an effort to bridge the gap between formal verification approaches and real applications. Rebeca is supported by a front-end tool for the translation of Rebeca codes into existing model-checker languages [16, 18]. Compositional verification and abstraction techniques are introduced to reduce the state space and make it possible to verify complicated reactive systems.

In this paper, we introduce an extended version of Rebeca, by enriching the model of computation with a formal concept of a component. The motivation is to provide a general framework which integrates in a formally consistent manner, both synchrony and asynchrony; introducing components to encapsulate tightly coupled reactive objects which may have synchronous communication; and present a tool-supported formal verification approach which provide us with open components whose behavior are verified. Certain properties are proven to be preserved when these model checked components are composed with other arbitrary components. Components are introduced for integrating different communication patterns (synchronous and asynchronous), at different levels of abstraction. At the highest level of abstraction, components only interact asynchronously via broadcasting anonymous messages. At a lower level of abstraction (within a component), computations, on the one hand are driven by asynchronous messages, and on the other hand can be synchronized by a handshaking communication mechanism. We present a formal operational semantics of the extended Rebeca language and show how components can be used in a modular verification approach. The modular verification approach, not only provide us with reliable components, but also is useful to overcome the state explosion problem in model checking.

Plan of the paper. In the next section we first discuss related work, including related work on actor models and some other concurrent modeling languages, and also automatic verification tools. In Section 3, we show syntax and formal semantics of Rebeca, extended by synchronous messages and broadcast communication. We start with the local configuration within a component and then move to components and global configuration as the higher level structures in our operational semantics; a simple example is included to explain Rebeca syntax and semantics. Section 5 explains our approach for verifying properties of components, based on a formal structural operational semantics, in order to offer reliable off-the-shelf components. We use our simple example again, to show our model and module checking approach. In Section 6, we have a short conclusion and a description of our future work.

2 Related Work

Different object-oriented models and languages for concurrent systems have been proposed since the 1980s. The *actor* model was originally introduced by Hewitt [11] as an agent-based language. It was later developed by Agha [5] into a concurrent object-based model. The actor model is proposed as a model of concurrent computation in distributed, open systems. Some interesting work has been done on formalizing the actor model [19, 10].

The actor model was first explained as a simple functional model [4, 5], but several imperative languages have also been developed based on it [20]. Besides its theoretical basis, the actor model and languages provide a very useful framework for understanding and developing open distributed systems.

Input-output automata for modelling asynchronous distributed systems is introduced by Lynch and Tuttle in [14]. They showed how to construct modular and hierarchical correctness proofs for their models. Alur and Henzinger proposed RML (Reactive Modules Language) for modelling a system and used a subset of linear temporal logic, alternating-time temporal logic, to specify its properties [8]. RML supports compositional design and verification.

Many models, including those we mentioned above, have tools for facilitating their analysis ([7]). There are also model checkers which are developed with their own modeling languages, such as NuSMV [1] and Spin [3]. However, to the best of our knowledge, our work presents a first component-based ipmerative actor-language which integrates synchronous message passing and which is supported by compositional verification techniques. Furthermore, the design of the Rebeca language is based on a powerful yet simple paradigm; providing the basic necessary constructs in a Java-like syntax which is easy to use for practitioners. The event-driven nature of its semantics, leads to straightforward approaches which decrease the state space significantly. Abstraction techniques which preserve LTL-X and ACTL properties, can be applied automatically. In our previous work [17], components are sub-models which are the result of decomposing a closed model in order to apply compositional verification, but here the concept of components is what we have in component-based modeling which are independent modules with well-defined interfaces. Once verified, a component can be used as a reliable off-the-shelf module. Hence, in the modular verification approach presented here, although the strategy in abstraction techniques is the same, but the technical details are quite different. A similar approach in using abstraction technique for model checking open SDL systems is used in [12].

3 Rebeca

A model in Rebeca consists of a set of *rebec*s ($\underline{re}active\ o\underline{b}j\underline{ec}t$) which are concurrently executed. Rebecs are encapsulated active objects, with no shared variables. Each rebec is instantiated from a *class* and has a single thread of execution which is triggered by reading messages from an unbounded queue. Each message specifies a unique method to be invoked when the message is serviced. When a message is read from the queue, its method is invoked and the message is deleted from the queue. Note that reading messages, thus, drives the computation of a rebec. Rebecs do not provide an explicit control over the message queue. In order to increase the modeling power of actor-based languages, we extend the asynchronous communication mechanism of Rebeca with synchronous message passing and a mechanism for broadcasting anonymous messages.

Synchronous messages are specified only as a signature specifying the name of the message and the types of its parameters.

For sending a synchronous or asynchronous message to an internal rebec, we specify its name. An anonymous send statement represents a broadcast to other components. In order to introduce the extended version of Rebeca we need the following definition.

Definition 1 (Basic definitions).

- The predefined types T: *Int* for integers, *Bool* for Booleans, and *Reb* for rebec names, i.e., identities of the active object in Rebeca.
- The set Var is the set of variables of type T with typical elements $x_1, x_2, \ldots, x_n$, including instance variables and also local variables. We show local variables by $u_1, \ldots, u_n$, values by $v_1, \ldots, v_n$, and rebec names by $r, r', \ldots$.
- The set Val is the union of all the values for all the types, i.e., all the values for type *Int*, $\{True, False\}$ for type *Bool*, and all the rebec names for type *Reb*.
- The set Mes is the set of messages with typical elements $m, m_1, \ldots, m_n$.

A model in rebeca is a number of class definitions followed by rebecs instantiated from them. Components are declared as sets of rebecs. Each class, consists of an interface, declaration of instance variables and its body which is a set of method definitions. The following describes the syntax of the basic actions, and the methods in Rebeca. In the following definition, a shows the basic actions, and A stands for a name of a class (sometimes refer to as rebec template). S is the body of a method that includes local variable declarations and a sequential statement composed of the basic actions. A method definition, mtd, consists of a method signature and method body (S).

Definition 2 (Syntax of extended Rebeca).

The basic actions, and the methods in Rebeca are defined by the following BNF-grammar (we abstract from the syntax of expressions e_i, and brackets ([]) show the optional parts).

$$\begin{array}{l} a ::= \;\; x = e \;\mid\; x = new(A) \;\mid\; [x.]m(e_1, ..., e_n) \;\mid\; receive(m_1, ..., m_n) \\ S ::= \;\; a; S \;\mid\; a \\ mtd ::= \;\; m(u_1 : t_1, \ldots, u_n : t_n)[: S] \end{array}$$

An *assignment* statement, $x = e$, assigns the value resulting from the evaluation of the expression e to variable x. A *create* statement $x = new(A)$, creates a new rebec as an instance of class A and assigns its unique identity to the variable x. A *class* A, is a template that rebecs are instantiated from.

A *send* statement, can be sending a message to a rebec, specifying its name; or it can be an anonymous send. An anonymous *send* statement $m(e_1, ..., e_n)$, which does not indicate the name of the receiver, causes an asynchronous broadcast of the message m with actual parameters e_1 to e_n. This broadcast in fact will involve all the components of the system as described in the following section on the semantics. Within a component, this in turn, will cause sending an asynchronous message to all its rebecs.

Execution of a *send* statement, $r.m(e_1, ..., e_n)$, consists of sending a message m with actual parameters e_1 to e_n to the rebec r. Message passing can be both synchronous as well as asynchronous. Asynchronous messages define a corresponding message-handler S, also called a method, and there is no explicit receive statement for them. An asynchronous message will be stored in unbounded message queue of the callee, after which the caller proceeds with its own computation. When this message is read by the callee the corresponding statement is executed.

Synchronous messages are specified only in terms of their signature, they do not specify a corresponding handler S. Synchronous message passing involves a 'hand-shake' between the execution of a send-statement by the caller and a receive statement by the callee in which the (synchronous) message name specified by the caller is included. A *receive* statement, $receive(m_1, ..., m_n)$, denotes a nondeterministic choice between receiving messages m_1 to m_n. This kind of synchronous message passing is a two-way blocking, one-way addressing, and one-way data passing communication. It means that both sender and receiver should wait at the rendezvous point, only sender specifies the name of the receiver, and data is passed from sender to receiver.

When a sender sends a synchronous message it is blocked, waiting for the receiver to reach to the corresponding receive statement (which includes the message sent by the sender as an option). But this happen only if the receiver is not already waiting for that message, in the latter case the sender and the receiver meet, the data is passed to the receiver and they both got unblocked and continue their execution. If there are more than one sender waiting for a receive statement, then arriving to that receive statement, the receiver makes a nondeterministic choice between the coming messages. According to that choice the corresponding sender got unblocked, passes the data, and continue its execution. Other senders stay in their blocked state.

The body of each method, S, is a sequential statement composed of the basic actions. A *method definition*, mtd, defined as $m(u_1 : t_1, \dots, u_n : t_n)[: S]$, denotes the method that is correspondent to message m with virtual parameter u_1 to u_n of type t_1 to t_n, and the body S. The definition of method body S is optional, and we have the convention that $m(u_1 : t_1, \dots, u_n : t_n) : S$ corresponds to an asynchronous message, and $m(u_1 : t_1, \dots, u_n : t_n)$ corresponds to a synchronous message.

4 Operational Semantics

We will define the semantics of extended Rebeca in terms of a labeled transition system.

Semantics is defined in a structured manner which reflects the hierarchy of rebecs, component and component system: First we introduce a labelled transition system which describes the behavior of a rebec in isolation. This transition system forms the basis for a labelled transition system which descibes the behavior of a component as a set of rebecs. Finally, the latter system is used as a basis for describing the overall behavior of a system of components

Definition 3 (Local configuration).

Assuming a model with rebec template definitions: $A_1 = B_1, \dots, A_n = B_n$, where B_i is the body of the class, rebecs are instantiated from these templates. A local configuration l for a rebec is defined as a tuple $l =< r, \sigma, S, q >$ where

– r denotes the rebec identity,

- $\sigma \in Var \to Val$ assigns values to the variables of the rebec,
- S is the statement to be executed next, and
- q denotes the unbounded FIFO queue containing asynchronous messages.

Next, we introduce a labelled transition relation which describes the behavior of a rebec in isolation. The labels indicate the nature of the transition:

- the label τ indicates an internal computation step;
- a label $m(v_1, \ldots, v_n)$ indicates that the asynchronous message $m(v_1, \ldots, v_n)$ has been broadcasted;
- a label $r.m(v_1, \ldots, v_n)$ indicates that the asynchronous or synchronous message $m(v_1, \ldots, v_n)$ has been sent to the rebec r (which is required to be different from the executing rebec);
- a label $r.m(v_1, \ldots, v_n)$, where r denotes the executing rebec itself, indicates the reception of the message $m(v_1, \ldots, v_n)$.

For notational convenience, the parameters of a message are dropped in the following definitions when it does not cause loss of information, i.e., $m(v_1, \ldots, v_n)$ is shown simply by m.

Definition 4 (Local transition for processing message queue).

When the point of control is at the end of a method, its execution is finished which is denoted by nil. If there is a message at the top of the rebec's queue it is popped and the corresponding method is called for execution. The parameter values are substituted before execution. It is worthwhile to observe here that we don't have recursion in methods so we don't need to worry about fresh local variables. The above is formalized by the following transition:

$\langle r, \sigma, nil, q.m(v_1, \ldots, v_n)\rangle \xrightarrow{\tau} \langle r, \sigma', S, q\rangle$
where, given the method definition $m(u_1 : t_1, \ldots, u_n : t_n) : S$, $\sigma' = \sigma\{v_1/u_1, \ldots, v_n/u_n\}$ denotes the state resulting from assigning the values $v_1, \ldots, v_n$ to the formal parameters $u_1, \ldots, u_n$. Note that $\sigma\{v/u\}$ denotes the result of assigning the value v to u in the state σ.

Definition 5 (Local transition for assignment).

When the next statement to be executed is an assignment we have the following transition rule:

$\langle r, \sigma, x = e; S, q\rangle \xrightarrow{\tau} \langle r, \sigma', S, q\rangle$,
where $\sigma' = \sigma\{\sigma(e)/x\}$ and $\sigma(e)$ denotes the value of expression e in σ.

Definition 6 (Local transitions for send).

When the next statement to be executed is a send statement we distinguish between broadcast, sending to self, and sending to others :

- $\langle r, \sigma, m(e_1, \ldots, e_n); S, q\rangle \xrightarrow{m(\bar{v})} \langle r, \sigma, S, q\rangle$
  where $\bar{v} = (v_1, \ldots, v_n)$, and $v_i = \sigma(e_i)$.
- $\langle r, \sigma, x.m(e_1, \ldots, e_n); S, q\rangle \xrightarrow{r'.m(\bar{v})} \langle r, \sigma, S, q\rangle$
  where $\sigma(x) = r'$, $r \neq r'$, $\bar{v} = (v_1, \ldots, v_n)$, and $v_i = \sigma(e_i)$.
- $\langle r, \sigma, x.m(e_1, \ldots, e_n); S, q\rangle \xrightarrow{\tau} \langle r, \sigma, S, q.m(v_1, \ldots, v_n)\rangle$
  where $\sigma(x) = r$, and $v_i = \sigma(e_i)$.

The first case above describes the anonymous broadcast of an asynchronous message. The second case describes sending a synchronous or asynchronous message to another rebec. Finally, the last case describes sending of an asynchronous message to the rebec itself. Note that we do not allow sending synchronous messages to self, which will cause deadlock.

Definition 7 (Local transitions for receive).

We distinguish between the reception of synchronous and asynchronous messages:

- The following transition describes the reception of an asynchronous message for which the receiving rebec has a corresponding server:

  $\langle r, \sigma, S, q\rangle \xrightarrow{r.m} \langle r, \sigma, S, q.m\rangle$
- Asynchronous messages for which the receiving rebec does not have a corresponding receiver are simply discarded:

  $\langle r, \sigma, S, q\rangle \xrightarrow{r.m} \langle r, \sigma, S, q\rangle$
- Finally, we have the following transition which described the reception of a synchronous message:

  $\langle r, \sigma, receive(m_1, \ldots, m_n); S, q\rangle \xrightarrow{r.m(\bar{v})} \langle r, \sigma', S, q\rangle$
  where, given the method definition $m(u_1, \ldots, u_n)$ and $\bar{v} = (v_1, \ldots, v_n)$, $\sigma' = \sigma\{v_1/u_1, \ldots, v_n/u_n\}$.

Definition 8 (Local transition for creation).

When the next statement to be executed is a creation statement we have the following transition:

$\langle r, \sigma, x = new(A); S, q\rangle \xrightarrow{r'} \langle r, \sigma', S, q\rangle$ where $\sigma' = \sigma\{r'/x\}$. Here r' is chosen arbitrarily. Freshness of r' is ensured in the context of a component (described in the next section).

Next we describe the semantics of a component which is specified by a set of rebecs.

Definition 9 (Component configuration).

A component is a non-empty, finite set of rebecs and a component configuration is shown as $C = \{l_1, \ldots, l_n\}$ where l_i denotes the local configuration of rebec r_i.

Components interact only by broadcasting anonymous messages. The set of public methods of the rebecs inside a component define its (provided) interface. A message received by a component is broadcasted to all its internal rebecs. We formalize the externally observable behavior of a component by means of a transition relation with labels $!m$ and $?m$ which indicate sending and receiving anonymous asynchronous message m, respectively. Communications between rebecs of a component are hidden.

Definition 10 (Component transition for internal communication).

The following transition describes internal synchronous and asynchronous message passing,

$$\frac{l_i \overset{r_j.m}{\rightarrow} l'_i,\ l_j \overset{r_j.m}{\rightarrow} l'_j,\ i{\neq}j}{\{l_1,...,l_i,...,l_j,...,l_n\} \overset{\tau}{\rightarrow} \{l_1,...,l'_j,...,l'_j,...,l_n\}}$$

Note that this rule describes sending a synchronous or an asynchronous message from r_i to r_j $(i \neq j)$.

Definition 11 (Component transition for send).

The following rule describes broadcast of an anonymous asynchronous message generated by an internal rebec.

$$\frac{l_i \overset{m}{\rightarrow} l'_i}{\{l_1,...,l_i,...,l_n\} \overset{!m}{\rightarrow} \{l_1,...,l'_i,...,l_n\}}$$

Definition 12 (Component transition for receive).

The following rule describes the internal broadcast of a received anonymous (asynchronous) message.

$$\frac{l_i \overset{r_i.m}{\rightarrow} l'_i,\ for\ all\ \ i{\in}\{1,...,n\}}{\{l_1,...,l_i...,l_n\} \overset{?m}{\rightarrow} \{l'_1,...,l'_i...,l'_n\}}$$

Definition 13 (Component transition for creation).

The following rule describes the creation of an internal rebec.

$$\frac{l_i \overset{r}{\rightarrow} l'_i}{\{l_1,...,l_i,...,l_n\} \overset{\tau}{\rightarrow} \{l_1,...,l'_i,...,l_n,l_{n+1}\}}$$

where l_{n+1} denotes the initial local configuration of the newly created rebec r which is required not to exist in $\{l_1, \ldots, l_i, \ldots, l_n\}$, i.e., $r \neq r_i$, $i \in \{1, \ldots, n\}$.

Definition 14 (Component internal transition).

Finally, the following rule describes the internal interleaving execution of rebecs within a component.

$$\frac{l_i \overset{\tau}{\rightarrow} l'_i}{\{l_1,...,l_i,...,l_n\} \overset{\tau}{\rightarrow} \{l_1,...,l'_i,...,l_n\}}$$

A global model simply consists of a set of components.

Definition 15 (Global configuration).

A global configuration is a finite set of component configurations $\{C_1, \ldots, C_n\}$.

Next we define the global transition system which describes the behavior of a set of components as a closed system.

Definition 16 (Global transition for communication).

This transition describes the broadcasting mechanism of asynchronous anonymous messages.

$$\frac{C_i \xrightarrow{!m} C'_i,\ C_j \xrightarrow{?m} C'_j,\ i \neq j}{\{C_1,\ldots,C_i,\ldots,C_j,\ldots,C_n\} \xrightarrow{\tau} \{C'_1,\ldots,C'_i,\ldots,C'_j,\ldots,C'_n\}}$$

Note that an anonymous asynchronous message is broadcasted to all the other components.

Definition 17 (Global internal transition).

All the other transitions of components are as internal computation steps in the global configuration.

$$\frac{C_i \xrightarrow{\tau} C'_i}{\{C_1,\ldots,C_i,\ldots,C_n\} \xrightarrow{\tau} \{C_1,\ldots,C'_i,\ldots,C_n\}}$$

4.1 Example: Bridge Controller

Here, we explain a simple example to show our modeling approach. Consider a bridge with a one-way track where only one train can pass at a time. This example can be easily extended to multiple tracks. Trains enter the bridge from its left side, pass it, and exit from the right side. Rebeca code for this example is shown in Figure 1. We model the two ends of the bridge by two objects controlling these ends. These objects are described by the classes *leftController* and *rightController*. The rebecs *theLeftCtrl* and *theRightCtrl* are instantiated from these two classes and together form a component. Trains are modeled by the *Train* class. Many trains can be instantiated from this class, but in this example we only have two trains instantiated. Each single train instance is modeled as a component. Trains announce their arrival by broadcasting the anonymous message *Arriva(MyTrainNr)* to the Controller component. To this message only the *leftController* will react by broadcasting the *YouMayPass(MyTrainNr)* message after which the *leftController* waits for the synchronous message passed. The message *YouMayPass(MyTrainNr)* will be received by both trains, however only the train identified by *MyTrainNr* will enter the bridge (after the test the other train will remove the message from its queue and wait for the next message). Passing the bridge is modeled by broadcasting the message *Leave* to the Controller component. To this message only the *rightController* will react by sending the synchronous message passed to the *left-Cntroller* which enables the *leftController* to receive a new *Arrive* messages. Note that thus no trains are allowed to enter the bridge (by executing *GoOnTheBridge*) while the

```
activeclass leftController() {
    knownobjects { rightController right; }
    provided { Arrive; }
    required { YouMayPass; }
    statevars { int trainsin; }
    msgsrv initial() {
        trainsin = 0;
    }
    msgsrv Arrive (int TrainNr) {
        YouMayPass(TrainNr);
        trainsin = trainsin + 1;
        receive(passed);
    }
}
```

```
activeclass rightController() {
    knownobjects { leftController left; }
    provide { Leave; }
    request {}
    statevars { int trainsout; }
    msgsrv initial() {
        trainsout = 0;
    }
    msgsrv Leave() {
        trainsout = trainsout + 1;
        left.passed();
    }
}
```

```
activeclass Train() {
     knownobjects {}
      provided { YouMayPass; }
      required { Arrive; Leave; }
     statevars { boolean OnTheBridge; }
     msgsrv initial(int MyTrainNr) {
         self.ReachBridge();
         OnTheBridge = false;
  }
     msgsrv YouMayPass(int TrainNr) {
         if (TrainNr == MyTrainNr) {
             self.GoOnTheBridge();
             OnTheBridge = true;
         }
     }
     msgsrv GoOnTheBridge() {
         Leave();
         OnTheBridge = false;
         self.ReachBridge();
    }
     msgsrv ReachBridge() {
         Arrive(MyTrainNr);
     }
 }

 main {
     Train train1(1);
     Train train2(2);
     leftController theLeftCtrl(theRightCtrl);
     rightController theRightCtrl(theLeftCtrl);
 Components:
     {train1}; {train2};
     {theLeftCtrl, theRightCtrl};
 }
```

Fig. 1. Bridge controller example, modeled in extended Rebeca

leftController is suspended. Two variables, *trainsin* and *trainsout*, are added to the code for verification purposes, explained in Section 5.1.

In Figure 1, encapsulation of rebecs in a component and also three types of message passing can be seen. The two left and right controllers of the bridge are tightly coupled and are encapsulated in a component. It allows the synchronous message passing between them. Trains are independent objects and can communicate by broadcasting asynchronous messages. It is also shown that the broadcasted messages are only serviced by the provider rebecs.

5 Formal Verification

Formal verification of properties for components, is a problem of *model checking of open systems*. By an *open system*, we mean a system that interacts with its environment and whose behavior depends on this interaction; unlike a *closed system*, whose behavior is completely determined by the state of the system. The crucial point in model checking an open system, which is usually referred to as *module checking*, is modeling the environment. To model the nondeterminism, an environment can be modeled as a general process with arbitrary behavior [13, 6].

For module checking components in extended Rebeca, we define a general environment. A component interacts with its environment by means of sending and receiving asynchronous anonymous messages. Because of the asynchronous nature of the communication mechanism, we only need to model the messages generated by the environment. Each message generated by the environment is broadcasted to the internal rebecs of the component. If the required service is provided by a rebec, the message is put in the rebec's queue.

To model an environment which simulates all the possible behaviors of a real environment, we need to consider an environment nondeterministically sending unbounded number of messages. It is clear that model checking will be impossible in this case. To overcome this problem, we use an abstraction technique. Instead of putting incoming messages in the queues of rebecs, they may be assumed as a constant (although unbounded) set of requests to be processed at any time, in a fair interleaving with the processing of the requests in the queue. This way of modeling the environment, generates a closed model which is bisimilar to the model resulting from a general environment which nondeterministically sends unbounded number of messages.

We will proceed by a formal definition of a general environment for Rebeca components. Then we show that the component's behavior in this general environment, weakly simulates the behavior of the component being concurrently executed with any arbitrary component. So, we can use model checking to prove certain properties for a component interacting with a general environment, and then deduce that these properties preserve for that component in any environment. Before showing the weak simulation, we use our abstraction technique to overcome the unboundedness problem of queues in a general environment, and make model checking feasible. Before that, we also use common data abstraction techniques on parameters of incoming messages, to make the number of messages bounded.

Definition 18 (Environment of a component).

For each component C, we define a component E_C as a general environment for C, where E_C nondeterministically broadcasts all the provided messages of C.

The global configuration made by C and E_C is a closed model which we denote it as M, i.e. $M = \{C, E_C\}$. The interface and body of component E_C can automatically be derived from the interface of C. The required messages of E_C are all the provided messages of C, E_C has no provided message and no instance variable. For each provided messages m_C of C, there is a rebec in E_C, which has one method named *active* in its body. This method sends two messages: first m_C to C, and second an *active* message to itself. Sending the active message to itself makes an infinite loop for sending the m_C to C. According to the broadcast mechanism, the environment component E_C also receives all the messages from component C. As there are no provided messages in E_C, they are all purged.

In modeling environment as a component, we use the existing data abstraction techniques for the parameters of incoming messages to reduce the number of messages to a finite set, but still the number of sent messages can be unbounded. Given this assumption, we proceed to next definition.

Definition 19 (Queue abstraction).

In the model $M = \{C, E_C\}$, instead of putting all the messages coming from E_C in the message queues of rebecs in C, we model each external message by a transition of C. More specifically, for each external message m we introduce the following local transition:

$$\langle r, \sigma, nil, q\rangle \xrightarrow{r.m} \langle r, \sigma, S, q\rangle,$$

where S is the handler of m. In this way, the queues of the closed system C only contain internal messages and we obtain a finite model of M in case the transition system of the closed system C is finite. This abstraction of M we denote by C^a.

Theorem 1 (Correctness of the queue abstraction).

The model $M = \{C, E_C\}$, is bisimilar to model C^a (with the conventional definition for bisimulation). The proof is based on the fair interleaving manner of processing the messages in the queue and the set of provided messages.

Now, we will proceed by defining the weak simulation relation between two models in Rebeca, first one consists of a component and the general environment (namely $\{C, E_C\}$), and the second one consists of that component composed by any arbitrary component (namely $\{C, C'\}$). Our final goal is to prove certain properties for $\{C, E_C\}$, and conclude that they preserve for $\{C, C'\}$. Here, according to Theorem 1, we can develop the weak simulation relation upon the abstracted model (C^a instead of $\{C, E_C\}$).

Next, we define a general definition for weak simulation, and then continue by applying the definition on our specific models.

Definition 20 (Weak Simulation).

Given two transition systems $\Sigma_1 = (S_1, T_1, I_1)$ and $\Sigma_2 = (S_2, T_2, I_2)$ where S_i is the set of states for Σ_i, $T_i \subseteq S_i \times S_i$ is the transition relation, I_i is the initial state for Σ_i:

1. A relation H is a R-simulation between Σ_1 and Σ_2, where $H, R \subseteq S_1 \times S_2$, if and only if for all s_1 and s_2, if $H(s_1, s_2)$ then the following conditions holds:
   (a) $R(s_1, s_2)$.
   (b) For every state s_1' such that $(s_1, s_1') \in T_1$ we have $(s_1', s_2) \in H$ (stuttering), or else there is a state s_2' with the property that $(s_2, s_2') \in T_2$ and $(s_1', s_2') \in H$.
2. We say that Σ_2 R-simulates Σ_1 (denoted by $\Sigma_1 \leq \Sigma_2$) if there exists a R-simulation H between Σ_1 and Σ_2 such that $H(I_1, I_2)$.

The above general definition for the specific case of models in extended Rebeca, shall be instantiated by defining relation H between the states of two systems. For that we need a projection definition: $s_i \downarrow C$ means the projection of state s_i of the model M_i, over variables and queues of rebecs in component C, and for the queues, only considering messages coming from internal rebecs. It means ignoring the variables and contents of message queues of other components in M and also ignoring the messages sent from other components in the queues of rebecs in C. So, $s_1 \downarrow C = s_2 \downarrow C$ means variables of rebecs in component C have the same value in states s_1 and s_2, and also the message queues of rebecs in C have the same content in states s_1 and s_2, considering only the messages coming from internal rebecs of C.
Based on Definitions 18 , 19 and 20, we have the following theorem.

Theorem 2 (Weak simulation between models).

Given a component C and an arbitrary component C', the transition system $\Sigma_1 = (S_1, T_1, I_1)$ of the (abstracted) model C^a, R-simulates the transition system $\Sigma_2 = (S_2, T_2, I_2)$ of the model $M_2 = \{C, C'\}$, where $R(s_1, s_2)$ iff $s_1 \downarrow C = s_2 \downarrow C$.

The formal proof is not included here, but the extension to Rebeca is made carefully to keep the proof similar to one in [15]. Intuitively, it is based on the fact that in each state of the transition system Σ_2, the enabled transitions are a subset of enabled transitions in the correspondent state of the transition system Σ_1. By correspondent states, we mean the states in Σ_1 and Σ_2 which satisfy the simulation relation, starting from the initial state. This is because of the definition of general environment which covers all the possible messages, and hence transitions.

Definition 21 (Satisfaction relation).

1. A computation of a component C is a maximal execution path beginning at the initial state. Given an LTL formula ϕ, we say that $C \models \phi$ iff ϕ holds for all computations of C.
2. Given a CTL formula ϕ, we say that $C \models \phi$ iff ϕ holds in the initial state of C.

We have the following theorem from [9].

Theorem 3 (Property preservation).

If M_1 weakly simulates M_2, then for every ACTL* or LTL formula ϕ without the next operator (with atomic propositions on variables in M_1), $M_2 \models \phi$ implies $M_1 \models \phi$.

In the module checking approach we used Definition 18, by modeling a general environment as a component. Then, we used Definition 19 and Theorem 1 to abstract from unbounded queues resulting from external messages and as such obtain a reduction of the state-space.

Next, we shall explain how to model check the obtained closed model. In model checking the asynchronous kernel of Rebeca, we gained a significant state reduction due to the asynchronous nature of communication and computation which allows to model the execution of a method as an atomic operation. In the presence of the synchronous communication mechanism this is no longer possible because of the additional synchronization between sender and receiver which requires the introduction of new states. However, this extension is bounded by the number of synchronous messages and rebecs, and as an internal behavior of a component, it is resolved by model checking, without any effects on the theorem.

5.1 Verifying Properties of the Bridge Controller

Consider the Bridge controller model in Figure 1, which is explained in Section 4.1. Safety property of the model is verified using our tool, Rebeca Verifier [18]. Rebeca Verifier enables us to enter our model as Rebeca code, and enter the properties as LTL formulas based on variables in the Rebeca code. The model and the properties are then translated to the modeling and specification languages of the back-end model checker, Spin [3]. In this example we explain how we can check the mutual exclusion property, which is at any moment only one train should be on the bridge. This property can be specified using the state variable *OnTheBridge* of the trains. The LTL formula for checking this property is the followings ($\Box$ denotes *always*) :

– Mutual exclusion: $\Box!(train1.OnTheBridge \ \&\& \ train2.OnTheBridge)$

To show our module checking approach and the abstraction techniques discussed, we consider the controller as an open component C. Our purpose is to check its properties in all the possible conditions, i.e., in a general environment. A general environment can be considered as an environment sending to controller component, all of its provided messages in a nondeterministic way, what we called E_C. The provided messages are *Arrive* serviced by *leftController* and *Leave* serviced by *rightController*. Our tool supports module checking components by modeling the abstracted environment C^a instead of E_C. Here, it means that in those states where the statement to be executed is nil, we can take a message from top of the queue to execute it, or execute one of the two provided message servers, *Arrive* and *Leave*, which are not put in the queue but are placed in a constant set and is considered to be always enabled.

In Module checking the controller component, we remove all other rebecs including their state variables and queues. So, we cannot reach *OnTheBridge* variables to check the properties. In this case, state variables *trainsin* of *theLeftCtrl* and *trainsout* of the *theRightCtrl* are used to check the mutual exclusion property which is restated as: $\Box\ (theLeftCtrl.trainsin - theRightCtrl.trainsout \leq 1)$. Model checking proves that this property holds, and based on our theory established in Section 5, we can conclude that this property holds for any model consisting of the controller component and any arbitrary component.

6 Conclusion and Future Work

In this paper, we have enriched the modeling power of the basic message-driven, asynchronous computational model of the actor-based language Rebeca by introducing a formal concept of components for structuring a model in Rebeca and to integrate asynchrony by synchronous message passing. We exploited the additional structuring mechanisms, provided by components, in a compositional verification approach based on model-checking. Formal semantics of Rebeca is used to establish the verification theory corresponding this approach. We use the Rebeca Verifier tool as an integrated tool , which uses NuSMV and Spin as back-end model checkers to verify properties of Rebeca models and also supports modularization and abstraction techniques provided.

Our research group in Tehran and Sharif universities is working on the Rebeca Verifier tool. The next step will involve the extension to components and providing the fully automated module checking technique for extended Rebeca. For more details and further information refer to our home page [2].

Furthermore, we used Rebeca for modeling security protocols, using dynamic data structures to describe the behavior of intruders.For model checking these applications, we therefore need appropriate abstraction techniques.

Acknowledgement

Part of the work of the second author is funded by the European IST project Omega (2001-33522).

References

1. NuSMV user manual. availabe through http://nusmv.irst.itc.it/NuSMV/ userman/index-v2.html.
2. Rebeca. http://khorshid.ut.ac.ir/∼rebeca.
3. Spin user manual. available through http://netlib.bell-labs.com/netlib/spin/whatisspin.html.
4. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1990.
5. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
6. R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automating modular verification. In *CONCUR: 10th International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 82–97. Springer-Verlag, Berlin, Germany, 1999.

7. R. Alur, T. A. Henzinger, F. Y. C. Mang, and S. Qadeer. MOCHA: Modularity in model checking. In *Proceedings of CAV'98*, volume 1427, pages 521–525. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998.
8. R. Alur and T.A. Henzinger. Reactive Modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
10. M. Gaspari and G. Zavattaro. An actor algebra for specifying distributed systems: The hurried philosophers case study. *Lecture Notes in Computer Science*, 2001:216–246, 2001.
11. C. Hewitt. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, April 1972.
12. N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DTSpin. In *FME'2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 531–548. Springer-Verlag, Berlin, Germany, 2002.
13. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
14. N.A Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.
15. M. Sirjani and A. Movaghar. An actor-based model for formal modelling of reactive systems: Rebeca. Technical Report CS-TR-80-01, Tehran, Iran, 2001.
16. M. Sirjani, A. Movaghar, H. Iravanchi, M. Jaghoori, and A. Shali. Model checking Rebeca by SMV. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'03)*, pages 233–236, Southampton, UK, April 2003.
17. M. Sirjani, A. Movaghar, A. Shali, and F.S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, Dec. 2004.
18. 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 2004*, pages 145–148. IEEE Computer Society, June 2004.
19. C. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, August 2002.
20. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.