Journal of Universal Computer Science, vol. 11, no. 10 (2005), 1695-1717
submitted: 20/6/05, accepted: 23/9/05, appeared: 28/10/05 © *J.UCS*

Modular Verification of a Component-Based Actor Language

Marjan Sirjani
(Department of Electrical and Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
and
School of Computer Science, IPM, Niavaran Sq., Tehran, Iran
msirjani@ut.ac.ir)

Frank S. de Boer
(Department of Software Engineering
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands
f.s.de.boer@cwi.nl)

Ali Movaghar
(School of Computer Science, IPM, Niavaran Sq., Tehran, Iran
and
Department of Computer Engineering
Sharif University of Technology
Azadi Ave., Tehran, Iran
movaghar@ipm.ir)

Abstract: Rebeca is an actor-based language for modeling concurrent and distributed systems as a set of reactive objects which communicate via asynchronous message passing. Rebeca is extended to support synchronous communication, and at the same time components are introduced to encapsulate the tightly coupled reactive objects which may communicate by synchronous messages. This provide us a language for modeling globally asynchronous and locally synchronous systems. Components interact only by asynchronous messages. This feature and also the event-driven nature of the computation are exploited to introduce a modular verification approach in order to overcome the state explosion problem in model checking. In this paper we elaborate on the corresponding theory of the modular verification approach which is based on the formal semantics of components in extended Rebeca.

Keywords: the actor model, reactive systems, Rebeca, component, modular verification.
Category: F.3.1, F.3.2, D.2.4

1 Introduction

Using formal methods and among them formal verification is a promising approach in developing more reliable software systems. 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. The modeling language can be a formal language or shall be provided by formal semantics on which the analysis method is established. The specification language for concurrent and reactive systems is usually based on automata theory or temporal logic [18].

In formal verification approaches, there are two basic methods of analysis: model checking and deductive methods. Typically, model checking is performed by a software tool, performing an exhaustive simulation of the model on all possible inputs and then applying some analysis on it. The main problem with model checking is state explosion. In a deductive method, the problem is formulated as proving a theorem in a mathematical proof system, and the modeler attempts to construct the proof of the theorem, usually using a theorem prover as an aid. Formulating a complex software system as a theorem is not always easy and working with the existing theorem provers usually needs special expertise. Using abstraction and modularity is a general solution for the problem of modeling and also verification of complex systems. By targeting specific domains, one may find more efficient specialized approaches and techniques for abstraction and modularization.

Rebeca ($\underline{Re}active\ Ob\underline{j}\underline{ec}ts\ L\underline{a}nguage$) is an actor-based language with a formal foundation, introduced in [19, 21] 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 [20, 22, 23], also, recently a tool is developed for direct model checking of Rebeca codes [15]. Inherent characteristics of Rebeca are used to introduce compositional verification and abstraction techniques for reducing the state space and make it possible to verify complicated reactive systems.

An extended version of Rebeca is introduced in [24], 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 provides 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, and so, they can be plugged in a model relying on their behavior. 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. A formal operational semantics of the extended Rebeca language is presented and it is shown that how components can be used in a modular verification approach. The modular verification approach, not only provides us with reliable components, but also is useful to overcome the state explosion problem in model checking.

This paper is an extended version of the conference paper [24]. In the process of providing rigorous proofs for the theorems, subtle but important changes are made to the definitions and theorems: the definitions for `initial` state and also internally broadcasting messages are slightly changed to make the theorems valid; formalizing queue abstraction is changed and it is observed that both queue abstraction and component composition can be proved to preserve the specific properties via weak simulation relation. Proofs are provided for the theorems. Also, the syntax definition now include the definition of a reactive class, reactive class body, instantiating rebecs, and a model as a set of components. In order to make the explanations more clear the case study provided in [24] is changed to a running example and the extended Rebeca model used in modular verification is added to the paper (Figure 3) which shows the environment and the components.

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 Sections 3 and 4, 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. 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. A simple example is used as a running example to explain Rebeca syntax and semantics, and also to show the 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. Actors have encapsulated states and behavior; and are capable of changing behavior, creating new actors, and redirecting communication links through the exchange of actor identities. Some interesting work has been done on formalizing the actor model [25, 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 [26]. 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 modeling asynchronous distributed systems are introduced by Lynch and Tuttle in [17]. They showed how to construct modular and hierarchical correctness proofs for their models. Alur and Henzinger proposed RML (Reactive Modules Language) for modeling 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 imperative 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 [21], components are sub-models which are the result of decomposing a closed model in order to apply compositional verification, but here the concept of a component is what we have in component-based modeling which is an independent module with a well-defined interface. 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 techniques for model checking open SDL systems is used in [13].

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 *reactive 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. Regarding the *infinite* behavior of the semantics, communication is assumed to be fair [4]: all the sent messages eventually reach their respective inboxes and will eventually be invoked by the corresponding rebec. Each rebec has an `initial` message server, and in the initial state the queue of the rebec is empty and its statement to be executed is the statement of the `initial` message server.

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., identifiers of the active object in Rebeca.
- The set Var is the set of typed variables with typical elements $x_1, x_2, \ldots, x_n$, including instance variables and also local variables. We denote 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 integers 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, and components which 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 interface includes known objects (rebecs which messages can be sent to), and provided and required message servers. Figure 1 shows the abstract syntax of extended Rebeca.

Definition 2 Syntax of extended Rebeca.

The abstract syntax of Rebeca is defined by the BNF-grammar in Figure 1. The detailed syntax for conditional statements, expressions and actual parameters, and known object bindings are not included. Brackets ([]) are used to show the optional parts.

An *assignment* statement written as $x = e$, assigns the value resulting from the evaluation of the expression e to variable x. A *create* statement $x = newA()$, creates a new rebec as an instance of reactive class A and assigns its unique identity to the variable x. A *reactive class* named by A, is a template that rebecs are instantiated from. The parameters which can be passed to the created rebec are its known rebecs and the parameters which are passed to the *initial* message server. Known rebecs are correspondent to the *knownobjects* part of the interface of each reactive class and determine the rebecs which the messages are sent to.

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 like $m(e_1, ..., e_n)$, 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 one of its rebecs (non-deterministically chosen). Alternatively, we could also broadcast internally the message to all the rebecs of the receiving component; however, this model is not consistent with our queue abstraction theory described later.

```
<model> ::=
    <reactiveclasses>
    <main>
    <components>
<reactiveclasses> ::= {<reactiveclass>}+
<reactiveclass>   ::=
    reactiveclass <reactiveclassName>'('<queueLength>')' '{'
        <knownobjects>
        <provided>
        <required>
        <statevars>
        <body>
    '}'
<knownobjects> ::=
    knownobjects '{'
        {<var>';'}*
    '}'
<provided> ::=
    provided '{'
        {<methodName>';'}*
    '}'
<required> ::=
    required '{'
        {<methodName>';'}*
    '}'
<statevars> ::=
    statevars '{'
        {<var>';'}*
    '}'
<body> ::=
    {<method>';'}+
<method> ::=
    msgsrv <methodName> '(' <parameters> ')'[ '{'
        {<statement>';'}*
    '}']
<parameters> ::= <var> | <var> ',' <parameters>
<var> ::= <typeName> <varName>
<statement> ::=
    <send> | <assignment> | <conditional> | <create> | <receive>
<send> ::=
    [<varname> '.'] <methodName> '(' {<actualParameters>}* ')'
<assignment> ::= <varname> = <expr>
<create> ::=
    <varname> = new <reactiveclassName> '(' <knownobjectsBindings>;
                                                <actualParameters> ')'
<receive> ::= receive '(' <methodNames>')'
<main> ::=
    main '{'
        {<rebec>';'}+
    '}'
<rebec> ::=
    <reactiveclassName> <varname> '(' <knownobjectsBindings>;
                                                <actualParameters> ')'
<components> ::=
    components '{'
        {<varnames>';'}+
    '}'
<varnames> ::=  <varname> | <varname> ',' <varnames>
```

Figure 1: Extended Rebeca syntax

Execution of a *send* statement, say $r.m(e_1, ..., e_n)$, 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 an 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 'handshake' 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, say $receive(m_1, ..., m_n)$, 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 happens 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 incoming messages. According to that choice the corresponding sender get unblocked, passes the data, and continues its execution. Other senders stay in their blocked state.

The body of each method is a sequence of statements. It can be denoted by S as a sequential statement composed of the basic actions. A *method definition*, $method$, can be denoted as $msgsrv\ m(u_1 : t_1, \ldots, u_n : t_n)[: S]$, which denotes the method that is invoked by 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 $m(u_1 : t_1, \ldots, u_n : t_n) : S$ corresponds to an asynchronous message, and $m(u_1 : t_1, \ldots, u_n : t_n)$ corresponds to a synchronous message. Note that synchronous messages do not invoke a method and as such they model synchronous data transmission.

After defining the reactive classes we have two parts as *main* part and *components* part. In the *main* part the set of rebecs are instantiated and in the *components* part the grouping of rebecs into components are determined.

The Bridge Controller: syntax. Here, we explain a simple example to show the syntax of extended Rebeca. 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 2. We model the two ends of the bridge by two reactive objects controlling these ends. The template of these reactive 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 makes a component. All the reactive classes have their corresponding *initial* message server.

In Figure 2, 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 one of the provider rebecs.

Three kinds of message passing used in this example are now further explained. The *ReachBridge* and *GoOnTheBridge* messages are asynchronous messages which are sent to internal rebecs of a component, here, they are both sent to *self*. The messages *Leave* and *Arrive* are broadcasted to anonymous receivers, which only one of the rebecs providing these messages react to. The message server of *Arrive* is provided by rebecs instantiated from *leftController* reactive class, and the message server of *Leave* is provided by rebecs instantiated from *rightController* reactive class. There is a synchronous message, *passed*, which intends to synchronize the *theLeftCtrl* and *theRightCtrl* rebecs, which are rebecs included in a component.

The variable *OnTheBridge* of the reactive class *Train* is used in Section 5 for verification purposes and two variables *trainsin* and *trainsout* are added to the code of *leftController* and *rightController* for modular verification (also explained in Section 5).

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 describes 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: $A_1 = B_1, \ldots, A_n = B_n$, 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

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

reactiveclass rightController() {
  knownobjects { leftController left; }
  provided { Leave; }
  required {}
  statevars { int trainsout; }
  msgsrv initial() {
    trainsout = 0;
  }
  msgsrv Leave() {
    trainsout = trainsout + 1;
    left.passed();

  }
}
```

```
reactiveclass 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();
  rightController theRightCtrl
                  (theLeftCtrl;);
components:
  {train1};{train2};
  {theLeftCtrl, theRightCtrl};
}
```

Figure 2: Bridge controller example, modeled in extended Rebeca

- 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 denoted 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 :

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

3. $\langle r, \sigma, x.m(e_1, \ldots, e_n); S, q\rangle \xrightarrow{\tau} \langle r, \sigma, S, q.m(v_1, \ldots, v_n)\rangle$
   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:

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

– We have the following transition which describes 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 $m(u_1, \ldots, u_n)$, $m \in \{m_1, \ldots, m_n\}$, 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 the configuration of which is given by $C = \{l_1, \ldots, l_n\}$ 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 forwarded to one of its internal rebecs (non-deterministically chosen). 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,

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

Note that this rule describes sending a synchronous or an asynchronous message from r_i to r_j ($i \neq j$) (look at Definition 6.2)

Definition 11 Component transition for send.

The following rule describes broadcast of an anonymous asynchronous message generated by an internal rebec (look at Definition 6.1).

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

Definition 12 Component transition for receive.

The following rule describes the reception of an anonymous (asynchronous) message.

$$\frac{l_i \overset{r_i.m}{\rightarrow} l_i',\ for\ some\ \ i{\in}\{1,...,n\}}{\{l_1,...,l_i...,l_n\} \overset{?m}{\rightarrow} \{l_1',...,l_i'...,l_n'\}}$$

Note that only one nondeterministically chosen rebec which provide a message server receives the corresponding message, i.e., the message is added to its message queue. For the other rebecs the message will simply be purged.

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.

$$\frac{l_i \xrightarrow{\tau} l'_i}{\{l_1,\ldots,l_i,\ldots,l_n\} \xrightarrow{\tau} \{l_1,\ldots,l'_i,\ldots,l_n\}}$$

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. This rule shows that when the message is sent it is put in the message queue of the receiver.

Definition 17 Global internal transition .

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

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

The Bridge Controller: semantics. Here, we explain more about the Bridge Controller example to show our modeling approach. The model begins by the execution of the *initial* message service. By executing the *initial* message server of each train, a *ReachBridge* message is sent to *self*, which in turn causes an *Arrive* message to be sent to the train. Trains announce their arrival by broadcasting the anonymous message *Arrive(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 (according to the conditional statement in the *YouMayPass* message server, the other train will do nothing). 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 *leftController* which enables the *leftController* to receive new *Arrive* messages. Note that thus no trains are allowed to enter the bridge (by executing *GoOnTheBridge*) while the *leftController* is suspended.

5 Formal Verification

For formal verification of systems we need a behavioral model to explain the behavior of the system, a property specification language to specify the required properties, and an analysis method to check the properties against the behavior. Rebeca is our behavioral model for modeling the system. Our specification language is temporal logic based on the state variables of rebecs in the Rebeca code. For analyzing the model, we need to model check an open system and we present our method for model checking such systems. We integrate model checking and deduction in our approach.

Property specification language. We use temporal logic as our property specification language. A *temporal formula* is constructed out of *state formulas* (assertions) to which we apply boolean connectives and temporal operators. State formulas are propositions defined over standard operations and relations over Var, the set of state variables. We naturally do not consider the message queue contents in our state formulas. So, the properties are based on state variables of each rebec in the model.

Model checking open systems. 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 [16, 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 put in the queue of an internal rebec which the required service is provided by. If there is more than one rebec providing that service, one of them is nondeterministically chosen.

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 (Definition 18). 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 (Definition 19). This way of modeling the environment, generates a closed model which simulates the model resulting from a general environment which nondeterministically sends unbounded number of messages (Theorem 24-1).

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 (Theorem 24-2). So, we can use model checking to prove certain properties for a component interacting with a general environment, and then deduce that these properties are preserved for that component in any environment (Theorem 23). 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. 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, containing only rebecs in their initial states (note that this implies that the queues are empty), 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.

The Bridge Controller: The controller as a component, and its environment. In Figure 2, we have two rebecs *theLeftCtrl* and *theRightCtrl* as a component. For module checking this component we abstract the model from other rebecs, and model an arbitrary environment for the controller component. This environment is constructed according to the provided messages of the component. The Rebeca code for this component and its environment is shown in Figure 3. The provided messages of the two rebecs, *theLeftCtrl* and *theRightCtrl*, are the messages *Arrive* and *Leave*. The reactive classes which build the environment are made based on these messages. These reactive

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



reactiveclass rightController() {
  knownobjects { leftController left; }
  provided { Leave; }
  required {}
  statevars { int trainsout; }
  msgsrv initial() {
    trainsout = 0;
  }

  msgsrv Leave() {
    trainsout = trainsout + 1;
    left.passed();

  }


}
```

```
reactiveclass ctrlEnvArrive() {
  knownobjects {}
  provided {}
  required { Arrive }
  statevars {}
  msgsrv initial(int MyTrainNr) {
    self.active();
  }

  msgsrv active() {
    Arrive(1);
    Arrive(2);
    self.active();
  }
}
reactiveclass ctrlEnvLeave() {
  knownobjects {}
  provided {}
  required { Leave }
  statevars {}
  msgsrv initial(int MyTrainNr) {
    self.active();
  }
  msgsrv active() {
    Leave();
    self.active();
  }
}
main{
  ctrlEnvArrive EnvArrive;
  ctrlEnvLeave  EnvLeave;
  leftController  theLeftCtrl();
  rightController theRightCtrl
                     (theLeftCtrl;);
components:
 {EnvArrive,EnvLeave};
 {theLeftCtrl, theRightCtrl};
}
```

Figure 3: The controller and its environment

classes have no known objects, no state variables, and no provided messages; each of them has only one required message, *Arrive* and *Leave*. The *Arrive* message has a parameter which has to be considered. An equivalent way is to have one reactive class for each different value of the parameter.

In modeling the 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 the 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 assume each external message to be always present, and model it by a transition of C. More specifically, for each external message m we introduce the following local transition:

$$\langle r, \sigma, nil, q\rangle \xrightarrow{\tau} \langle r, \sigma, S, q\rangle \tag{1}$$

where S is the handler of m. Consequently, when the statement to be executed is nil, we can take a message from top of the queue, like in Definition 4, or execute the message server of an external message, which is caused by the above transition. In this way, the queues of the component C only contain internal messages and we obtain a finite model in case C only generates a finite number of internal messages. We denote this behavioral abstraction of C by the transition relation $\rightarrow^a$: the local behavior of the rebecs is described by the local transitions with the above local transition for receiving external messages; furthermore at the component level we restrict to the component transitions for sending anonymous (asynchronous) messages and the transitions describing internal computation steps. In other words, we do not have component transitions for receiving anonymous (asynchronous) messages since they are modeled by the above local transition. Furthermore, the component transition for sending anonymous (asynchronous) messages also generates only the silent τ-action.

Let $\Sigma(M)$, for $M = \{C, E_C\}$, be the transition system of $M = \{C, E_C\}$ generated by the transition relation $\xrightarrow{\tau}$ from the initial set of components $\{C, E_C\}$. By $\Sigma(C)$ we denote the transition system generated by the transition relation $\rightarrow^a$ from the initial state $\{C\}$.

Next we will describe a proof method for establishing LTL properties without the next operator (LTL-X) and CTL properties without the next operator and existential path quantifiers (ACTL-X).

Definition 20 Satisfaction relation.

1. A computation of a transition system Σ is a maximal execution path, beginning at an initial state. Given an LTL formula ϕ, we say that $\Sigma \models \phi$ iff ϕ holds for all the computations of Σ ([18]).

2. Given a CTL formula ϕ, we say that $\Sigma \models \phi$ iff ϕ holds in the initial state of the transition system Σ ([9]).

Definition 21 Proof method.

Here, we want to prove that if a component C, satisfies a property ϕ, then ϕ is satisfied by *any* model containing the component C. Then, we are able to model check C to see whether $\Sigma(C) \models \phi$, and if this is true we conclude that C can be plugged in any model and still satisfies ϕ (for certain properties ϕ).

The correctness of this proof method follows from the following implication: $\Sigma(C) \models \phi$ implies $\Sigma(M') \models \phi$, for any model M' containing C (where $\Sigma(M')$ denotes the transition system of M').
This implication in turn follows from the following two implications:

1. $\Sigma(C) \models \phi \Rightarrow \Sigma(M) \models \phi$, for $M = \{C, E_C\}$, and
2. $\Sigma(M) \models \phi \Rightarrow \Sigma(M') \models \phi$, for an arbitrary model M' containing C.

Both implications follow from following definition of weak simulation and the corresponding property preservation theorem and is proved to be valid by Theorem 24.

Definition 22 Weak Simulation.

Consider 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, and I_i is the set of initial states for Σ_i.

We define $\Sigma_1 \sqsubseteq_R \Sigma_2$ (Σ_1 weakly R-simulates Σ_2) if $R \subseteq S_1 \times S_2$ is a relation between Σ_1 and Σ_2 such that for all $s_1 \in S_1$ and $s_2 \in S_2$, if $R(s_1, s_2)$ then for every transition $T_2(s_2, s'_2)$ either $R(s_1, s'_2)$ (stuttering) or there exists a state $s'_1 \in S_1$ such that $R(s'_1, s'_2)$ and $T_1(s_1, s'_1)$.

The transition system Σ_1 weakly simulates Σ_2 (denoted by $\Sigma_1 \sqsubseteq \Sigma_2$) if $\Sigma_1 \sqsubseteq_R \Sigma_2$, for some relation R, and for every initial state $s_2 \in I_2$ there exists an initial state $s_1 \in I_1$ with $R(s_1, s_2)$.

Theorem 23 Property preservation.

If the transition system Σ_1 weakly simulates Σ_2, then for every ACTL or LTL formula ϕ without the next operator (with atomic propositions on variables in M_1), $\Sigma_1 \models \phi$ implies $\Sigma_2 \models \phi$ ([9]).

We have the following weak simulation relations.

Theorem 24 Capturing queue abstraction and component composition.

Queue abstraction and component composition are captured by weak simulation. Let $M = \{C, E_C\}$ and M' be an arbitrary model containing C.

1. Queue abstraction: $\Sigma(C) \sqsubseteq \Sigma(M)$.
2. Component composition: $\Sigma(M) \sqsubseteq \Sigma(M')$.

Proof.

1. We can prove that $\Sigma(C)$ weakly R-simulates $\Sigma(M)$, $M = \{C, E_C\}$, by defining $R(s_C, s_M)$ if the set of local configurations of the rebecs in s_C consists of the local configurations of those rebecs in s_M which belong to the component C with their queues projected unto the internal messages only.

   We need to show that for all $s_C \in S_C$ and $s_M \in S_M$, if $R(s_C, s_M)$ then for every transition $T_M(s_M, s'_M)$ either $R(s_C, s'_M)$ or there exists a state $s'_C \in S_C$ such that $R(s'_C, s'_M)$ and $T_C(s_C, s'_C)$. All the transitions in T_M can be of two types according to the Definitions 16 and 17. For all the communication transitions in T_M (Definition 16), which are caused by sending (external) messages from E_C to C, there is a silent transition in T_C described in the queue abstraction definition (local transition (1) in Definition 19). For every transition in T_M which is corresponding to the message server that is nondeterministically chosen to be executed, there is a corresponding transition in T_C. So, we will have $T_C(s_C, s'_C)$ and $R(s'_C, s'_M)$. It can be seen that although the queue abstraction technique seems to be straight forward, it does not simply work in any kind of setting. Note that in our setting for a message broadcasted to a component, an asynchronous message is sent to one of its rebecs (non-deterministically chosen) which provides the service. For example, the design decision of internally broadcasting the messages coming from other components breaks the theorem. In that setting, all the internal rebecs which provide the message server have to reply to an external message. In M the message is put in the queues of all the rebecs providing the message and can be taken and served in any time afterwards, but in C all the rebecs have to execute the corresponding message server before going ahead. Hence, the behavior of C is no more an over-approximation of the behavior of M and the weak simulation relation does not hold.

   Another design decision is regarding to the initial state. The queue abstraction technique does not hold if we assume the initial state as the state with all the initial messages put in the queues. That is why we assume the queues to be empty and for each rebec the statement to be executed is the first statement of the initial message server.

   The messages which are sent out by C are handled in the same way in both models (they are purged) and cause similar transitions in both transition systems. For all the internal transitions in T_M (Definition 17) the condition holds because of the similar local configuration and internal messages in the queues according to the definition of R.

2. Furthermore, we can prove that $\Sigma(M)$ weakly R'-simulates $\Sigma(M')$, where M' is any model containing C, by simply defining $R'(s_M, s_{M'})$ if the set of local configurations of the rebecs belonging to the component C in s_M and $s_{M'}$ coincide.

   Here, the set of external messages which are sent to C in M' is a subset of external messages which are sent to it in M. So, for all the transitions $T'_M(s_{M'}, s'_{M'})$

caused by sending an external message to C there is a transition $T_M(s_M, s'_M)$ where $R(s'_M, s'_{M'})$. Also, the other transitions corresponding to C are the same, as the set of local configurations of the rebecs belonging to the component C in s and s' coincide. And, all the transitions which cause changes in components other than C in M', are the stuttering steps in our weak simulation relation, as $R'(s, s')$ is simply defined by the correspondence of the set of local configurations of the rebecs belonging to the component C in s and s'. ■

Next, we shall explain how to model check the transition system $\Sigma(C)$. 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 Theorem 24.

The Bridge Controller: verifying the properties. Consider our running example, the Bridge Controller of Figure 2. A safety property of the model is verified using our tool, Rebeca Verifier [22]. 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 (□ denotes *always*):

Mutual exclusion:

$\Box!(train1.OnTheBridge \mathbin{\&\&} train2.OnTheBridge)$

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 the 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*. Rebeca Verifier supports module checking by automatically composing the open component with its environment and reduce the problem to a model checking problem. The tool also apply the abstraction automatically and model checks the abstracted model C^a instead of $\{C, E_C\}$. 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 (see Figure 3). So, we cannot reach *OnTheBridge* variables of trains 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 proved that this property holds, and based on our theory established in Section 5, we conclude that this property holds for any model consisting of the controller component and any arbitrary component.

6 Conclusion and Future Work

It is shown that using Rebeca, as an actor-based language, is natural and efficient in modeling concurrent and distributed systems which their communication paradigm is only asynchronous, but, as expected, modeling synchrony introduces complexity in the model. This paper is an extension of [24], in which we have proposed extended Rebeca. In extended Rebeca the modeling power of the asynchronous and message-driven computational model of Rebeca is enriched by introducing a rendezvous-like synchronous message passing.

One of the most important motivations in using Rebeca is the support for formal analysis, including abstraction and compositional verification approaches [21, 22] and also optimization techniques in model checking Rebeca codes [14, 15], which are all based on the computational model of Rebeca. Components are added to extended Rebeca to encapsulate the reactive objects which communicate by synchronous messages. Hence, in a higher level of abstraction, where instead of reactive objects we take components as modules, the compositional verification and abstraction techniques can still be applied. Extended Rebeca provides us a language capable of modeling globally asynchronous and locally synchronous systems and supported by a modular verification approach. Application of partial order reduction and symmetry techniques in model checking extended Rebeca is not yet studied.

Our research group in Tehran and Sharif universities is working on the real world case studies and Rebeca Verifier tool. Extended Rebeca and the compositional verification approach are used in verifying IEEE 802.1D [12]. A project in using extended Rebeca for hardware/software co-verification is in its early stages and small case studies are modeled and model checked. The next step in our tool development project 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].

Acknowledgement

The research of the first and third authors is partly supported by a grant from IPM (No. CS1384-3-02).

References

1. NuSMV. http://nusmv.irst.itc.it/NuSMV.
2. Rebeca. http://khorshid.ut.ac.ir/∼rebeca.
3. Spin. http://netlib.bell-labs.com/netlib/spin.
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*, pages 82–97. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 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. H. Hojjat, H. Nakhost, and M. Sirjani. Formal verification of the IEEE 802.1D spanning tree protocol using extended Rebeca. In *Pre-proceedings of FSEN'05*, pages 201–216, Tehran, Iran, 2005. (To appear as Elsevier ENTCS).
13. 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.
14. M. M. Jaghoori, M. Sirjani, M. R. Mousavi, and A. Movaghar. Symmetry Reduction in the Formal Verification of Rebeca Models. In *Proceedings of ICDCIT'05*, India, 2005, to appear as Springer LNCS.
15. M. M. Jaghoori, A. Movaghar, and M. Sirjani. Modere: The Model-checking Engine of Rebeca. In *Proceedings of ACM SAC'06*, Symposium on Applied Computing - Software Verification track, France, 2006, to appear.
16. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
17. N.A Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.
18. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems (Safety)*. Springer-Verlag, Berlin, Germany, 1995.
19. M. Sirjani and A. Movaghar. An actor-based model for formal modeling of reactive systems: Rebeca. Technical Report CS-TR-80-01, Tehran, Iran, 2001.
20. 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.
21. M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundameta Informatica*, 63(4):183–235, December 2004.

22. 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 Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 145–148. IEEE Computer Society, 2004.
23. M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Model checking, automated abstraction, and compositional verification of Rebeca models. *Journal of Universal Computer Science*, 11(6):1054–1082, 2005.
24. M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Extended Rebeca: A Component-Based Actor Language with Synchronous Message Passing. In *Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 212–221. IEEE Computer Society, 2005.
25. C. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, August 2002.
26. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.