Fundamenta Informaticae 63 (2004) 1–26
IOS Press

Modeling and Verification of Reactive Systems using Rebeca

Marjan Sirjani* **and Ali Movaghar**
Department of Computer Engineering
Sharif University of Technology
Azadi Ave., Tehran, Iran
msirjani@cwi.nl; movaghar@sharif.edu

Amin Shali
Department of Electrical and Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
ashali@ece.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

Abstract. Actor-based modeling has been successfully applied to the representation of concurrent and distributed systems. Besides having an appropriate and efficient way for modeling these systems, one needs a formal verification approach for ensuring their correctness. In this paper, we develop an actor-based model for describing such systems, use temporal logic to specify properties of the model, and apply different abstraction and verification methods for verifying that the model meets its specification. We use a compositional verification approach for verifying safety properties of these models. For that we introduce a notion of component, based on an user-defined decomposition of the model. Components are more abstract than the model itself, and so we can reduce the state space of the model which makes it more amenable to model checking techniques. We prove that our

* Address for correspondence: Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, Iran

abstraction technique preserves a set of behavioral specifications in temporal logic. The soundness of the abstraction is proved by the weak simulation relation between the constructs.

Keywords: actor model, reactive systems, model checking, compositional verification, property preserving abstraction.

1. Introduction

Reactive systems are systems which have ongoing interactions with their environments, accepting requests and producing responses [28]. Such systems are increasingly used in applications where failure is unacceptable: electronic commerce, high-speed communication networks, traffic control systems, avionics, automated manufacturing, etc. Correct and highly dependable construction of such systems is particularly important and challenging. A very promising and increasingly attractive method for achieving this goal is using the approach of formal verification.

A formal verification method consists of three major components: a model for describing the behavior of the system, a specification language to embody correctness requirements, and an analysis method to verify the behavior against the correctness requirements [29, 29, 11].

Object-oriented modeling is widely used for representing reactive systems, which usually exhibit concurrency and are distributed. The actor model [3, 5] is a better candidate than existing object oriented programming languages, because the unit of distribution and concurrency are objects themselves and not threads, like in Java. This provides a simpler and more natural concurrency model. The actor model also promotes independent computing entities to support migration, distribution, dynamic reconfiguration, openness, and efficient parallel execution.

Much work has been done on formal methods with different kinds of models for system behavior and different verification approaches; also, the actor model is used in different ways for modeling open, distributed systems. But to the best of our knowledge, little is done on verifying actor languages (related work is discussed in Section 2).

In this paper we present a formal method for specifying and verifying properties of actor-based models in *Rebeca* [1] [39, 41]. Rebeca is inspired by the actors paradigm, but goes well beyond it by adding the concept of *components* and the ability to analyze a group of active objects as a component. Also, we have *classes* that active objects are instantiated from. Classes serve as templates for state, behavior, and the interface access; adding reusability in both modeling and verification process. Our method is supported by a front-end tool for the translation of Rebeca models into languages of existing model checkers. In order to cope with the problem of the state space explosion we propose a compositional verification approach which exploits the modular design of Rebeca models and the decomposition into components.

More specifically, the key features of our work are:

- using the *actor-based* asynchronous event-driven model for the specification of reactive systems;
- introducing *components* as open (sub-)systems as a basis for compositional verification;

[1] *Reactive Objects Language*

- presenting a *formal semantics* for the model and components, comprising their states, communications, state transitions, and the knowledge of accessible interfaces, which provides a formal basis for proving the correctness of our abstraction and reduction techniques;
- using different *abstraction techniques* based on the computing paradigm of the language which preserve a set of behavioral specifications in temporal logic, and which reduce the state space of a model, making it more suitable for model checking;
- establishing the soundness of these abstraction techniques by proving a *weak simulation* relation between the constructs;
- case studies for the application of our *compositional verification* approach, using the specified abstraction techniques;
- translating Rebeca models into target languages of existing model checkers, enabling *model checking* of actor-based models.

We explain Rebeca and the above features in the following sections.

Outline of the paper. In Section 2 we review the work that is related to our approach of modeling and analyzing systems. Section 3 presents the modeling language Rebeca and its syntax and formal semantics for Rebeca models. Model checking Rebeca models is explained in Section 4. Compositional verification and components as open systems are explained in Section 5. Weak simulation, as an abstraction technique applied on Rebeca components, and the theorems used to formally justify our compositional verification approach are defined in this section. In Section 6, we briefly introduce our tool for automatic translation of Rebeca models and components into popular model-checking languages. A running example is used through the paper to clarify syntax, semantics, model checking and compositional verification of Rebeca models. Section 7 concludes the paper and shows the direction of our future work.

2. Related Work and Motivation

Different languages have been proposed for modeling concurrent and distributed systems at different levels of abstraction. These languages also vary with respect to the formalization of their semantics and corresponding verification techniques, and to what extent these formalizations are supported by tools.

Examples of languages which provide a high-level of abstraction are CSP [20], CCS [33], I/O Automata [26], and RML [8]. A formal semantics of the actor language introduced in [3] is presented in [4, 44]. RML is supported by the model checker Mocha [9]; and FDR is the proof and analysis tool for CSP [36].

On the other hand, verification techniques and corresponding tools have also been developed for existing programming languages. For example, the NASA's Java PathFinder [18] is a translator from a subset of Java to Promela [2]. Its purpose is to establish a framework for verification and debugging of Java programs based on model checking. The Bandera Tool Set [15] is an integrated collection of program analysis, transformation, and visualization components designed to allow experimentation with model-checking properties of Java source code. Bandera takes Java source code and a specification written in Bandera's temporal specification language as input, and it generates a program model and

specification in the input language of one of several existing model-checking tools. These tools in principle can be applied directly to the verification of the actual implementation. However in practice such verification is only possible after an application of certain abstraction techniques to both the data and control [15].

Another approach is to use the language of a model checker itself in modeling concurrent and distributed systems. Some of these tools are successfully used in analyzing real systems, like NuSMV [1] and Spin [2]. The NuSMV system is a tool for checking finite state systems against specification in the temporal logic LTL and CTL. Spin is a widely distributed software package that supports the formal verification of distributed systems. Spin uses a high level language to specify systems descriptions, called Promela and LTL is its specification langauge. However these languages are designed for model checking purposes and their formal semantics are usually not explicitly given. Using these tools also needs certain expertise.

Apart from the identification of suitable language characteristics which mainly concern modeling issues like the level of abstraction, modularity and usability for practitioners, the two main approaches in formal verification both have their own deficiencies: Model checking in general suffers from the state-space explosion problem and deductive verification techniques require a high expertise and intensive interaction with the underlying theorem prover. In general, compositionality allows one to master both the complexity of the design and verification of software models. Decomposing a model into sub-models, verifying the properties of sub-models, and deducing the overall property is the main idea in compositional verification methods. Compositional verification can be exploited effectively only when the model is naturally decomposable [35].

Compositional verification has been used in different ways in the analysis of models of concurrency. Clarke, Long and McMillan used interface processes to model the environment for a component [12]. They modeled systems as finite transition systems and used CTL [16] to specify their properties. Input-output automata for modeling asynchronous distributed systems [27, 26] are introduced by Lynch and Tuttle. They showed how to construct modular and hierarchical correctness proofs for their models. Kesten and Pnueli mentioned modularization and abstraction as the keys to practical formal verification, using fair Kripke structure as the computational model for reactive systems and temporal logic as a requirement specification language [22]. An extension of bisimulation in a compositional proof of correctness of a protocol is used by Larsen and Milner in [25]. Alur and Henzinger proposed RML 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. Its compositional verification approach is assume-guarantee.

In the design of Rebeca both modeling and verification issues played a dominant role. Object-oriented modeling can be considered as the most successful approach in modeling in the software engineering community. The main motivation in designing Rebeca is to provide an object-based language with clearly defined encapsulated units of concurrency which can be easily used by software engineers. Furthermore, Rebeca provides a natural modular design approach with loosely coupled modules which makes the model suitable for applying compositional verification techniques.

Object-oriented models for concurrent systems have widely been proposed since the 1980s [3, 10, 14]. The *actor* model was originally introduced by Hewitt [19] as an agent-based language. It was later developed by Agha [3, 4, 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. Valuable work has been done on formalizing the actor model [5, 30, 44, 45, 17]. The actor model was first explained as a simple functional model [3, 4, 5], but several imperative languages have also been developed based on it [34, 48, 47]. Besides its theoretical basis, the actor model and languages provide a very useful framework for understanding and developing open distributed systems.

As far as we know, there is hardly any work on the tool-supported formal verification of actors [41, 37]. In order to integrate the practice of software engineering and formal verification Rebeca provides a rigorous semantic basis of an imperative view of actors. It is designed based on a powerful yet simple paradigm; providing the basic necessary constructs in a Java-like syntax which is easy to use for practitioners. In this paper we show how to exploit the event-driven computation model of Rebeca in automated abstraction and compositional verification techniques which preserve LTL-X and ACTL properties.

A tool for translating Rebeca to SMV and Promela enables us to model check Rebeca codes both in closed and open forms. We use our tool to show that our compositional verification approach reduces the state space in many practical cases [40]. A similar approach in using abstraction technique for model checking SDL systems is discussed in [21].

3. Rebeca: Syntax and Semantics

The model proposed here [39] is similar to the actor model in that it has independent active objects, asynchronous message passing, unbounded buffers for messages, dynamically changing topology, and dynamic creation of active objects. We add class declarations to the syntax; classes act like templates for states, behavior, and interfaces of active objects. Also, we have the notion of a component as a set of concurrently executing active objects, and the role of internal and external active objects differs from the one in the original actor model [3]. Our components are sub-models which are the result of decomposing a closed model in order to apply compositional verification, and should not be confused with the concept of components in component-based modeling which are independent modules with well-defined interfaces.

Our objects are reactive and self-contained. We call each of them a *rebec*, for *reactive object*. Computation takes place by message passing and execution of the corresponding methods (message server) of messages. Each message specifies a unique message server to be invoked when the message is serviced. Each rebec has an unbounded buffer, called a queue (or inbox), for arriving messages. When a message at the head of a queue of a rebec is serviced, its message server is invoked and the message is deleted from the queue. We may refer to the messages as 'method invocation requests'.

Each rebec is instantiated from a *class* and has a single thread of execution. We define a *model*, representing a set of rebecs, as a closed system. It is composed of rebecs, which are concurrently executed, and are interacting with each other. We can introduce *components* as open systems, consisting of subsets of rebecs in a model.

The execution of a message server is triggered by removing its message from the top of the queue and results in an atomic execution of its body which cannot be interleaved by any other method execution. Note that this coarse-grained granularity of the interleaving of methods is compatible with the asynchronous nature of the communication of Rebeca, which does not contain suspending communication primitives (e.g. a possibly suspending receive state). It also reduces the state space and makes the model simpler.

3.1. Syntax

The syntax for reactive classes (reactive-object templates), rebecs (reactive class instantiations), and models (parallel composition of rebecs) is presented in Figure 1. The syntax of a `<reactive class>` definition is similar to Java, except for the definition of `<knownobjects>`. The rebecs included in the `<knownobjects>` part of a reactive class definition, are those rebecs which their message servers may be called by instances of this reactive class.

After declaring the known rebecs, a list of reactive class fields are declared in `<statevars>` part. Then the methods, which may themselves contain local variables, are defined as message servers. Variables are typed, and method declarations follow a standard syntax. Unlike Java, methods have no return mechanism and therefore no return type. The core language for statements (`<statement>`) allows the remote method invocation requests (`<mir>`), assignments (`<assignment>`), if-statements (`<conditional>`), object creation (`<create>`), and sequential composition.

In `<mir>`, after specifying the callee (receiver) id, the method name and actual parameters are included. This can be viewed as a message consists of the callee id, message id and the parameters passed to the callee. Although not mentioned explicitly in the message, the caller (sender) passes its rebec identity (self) to the callee (receiver). Caller and callee may be the same rebec, modeling local calls (sends to self).

It is required that every reactive class definition has at least one method named *initial*. In the initial state of the system, each rebec has an *initial* message in its message queue, so *initial* is the first method executed by each rebec. After defining the reactive classes, there is a keyword `<main>` followed by the definition of the Rebeca model which is defined as a finite collection of rebecs that are (created and then) run in parallel. In declaring a rebec, the bindings to its known rebecs is specified in its parameter list.Variables are typed and the variables denoting a *known object*, a *receiver* of a message, and a *created* object have to be of type *rebec identifier*. Rebec identifiers can be passed as parameters, but cannot be referenced in an *assignment* statement.

We use producer-consumer as a simple running example through the paper to show the syntax and semantics of Rebeca and also other main ideas of the paper. We start with a simple version and discuss different features of Rebeca by extending this example.

Example 3.1. (Producer-consumer: a Rebeca model)
There is a buffer which a producer puts its products in it and a consumer takes the products from it. The producer cannot put a product in a full buffer and a consumer cannot take a product from an empty buffer. Also, the buffer is a critical section that both the producer and the consumer cannot put and take the products in/of the buffer at the same time.

The system consists of reactive classes: *Buffer*, *Producer*, and *Consumer*, that are templates for defining a buffer, a producer, and a consumer (see Figure 2). The known rebecs of the Buffer are the Producer and the Consumer, and the known rebec of the Producer and the Consumer is only the Buffer. The Producer and the Consumer do not send messages to each other directly.

State variables of each rebec are declared after the known objects. The rebec buffer has variables to show when the buffer is empty or full, whether the producer or the consumer are waiting, the length of the buffer which is the number of elements in the buffer, and pointers to the next empty and next full elements which the producer puts the next product in it and the consumer takes the next product from it. The producer and the consumer have no state variables.

```
<model> ::=
    <reactiveclasses>
    <main>
<reactiveclasses> ::= {<reactiveclass>}+
<reactiveclass> ::=
    reactiveclass <reactiveclassName>'('<queueLength>')' '{'
        <knownobjects>
        <statevars>
        <body>
    '}'
<knownobjects> ::=
    knownobjects '{'
        {<var>;}*
    '}'
<statevars> ::=
    statevars '{'
        {<var>;}*
    '}'
<body> ::=
    {<method>}+
<method> ::=
    msgsrv <methodName> '(' {<parameters>} ')' '{'
        {<statement>;}*
    '}'
<parameters> ::=
    <var> | <var> ',' <parameters>
<var> ::=
    <typeName> <varName>
<statement> ::=
    <mir> | <assignment> | <conditional> | <create>
<mir> ::=
    <varname> '.' <methodName> '(' {<varname>}* ')' ';'
<create> ::=
    <varname> = new <reactiveclassName> '(' <knownobjectsBinding> ')'
<model> ::=
    main '{'
        {<rebec>;}+
    '}'
<rebec> ::=
    <reactiveclassName> <varname> '(' <knownobjectsBinding> ')'
```

Figure 1. Reactive class, rebec and model definition syntax

```
activeclass BufferManager(5) {
  knownobjects {
    Producer producer;
    Consumer consumer;
  }
  statevars {
    boolean empty;
    boolean full;
    boolean producerWaiting;
    boolean consumerWaiting;
    int bufferCount;
    int nextProduce;
    int nextConsume;
  }
  msgsrv initial() {
    bufferCount = 5;
    empty = true;
    full = false;
    producerWaiting = false;
    consumerWaiting = false;
    nextProduce = 0;
    nextConsume = 0;
  }
  msgsrv giveMeNextProduce() {
    if (empty || !(nextProduce ==
nextConsume)) {
      producer.produce(nextProduce);
    }
    producerWaiting = true;
  }
  msgsrv giveMeNextConsume() {
    if (!empty) {
      consumer.consume(nextConsume);
    }
    consumerWaiting = true;
  }
  msgsrv ackProduce() {
    nextProduce = (nextProduce + 1) %
bufferCount;
    if (nextProduce == nextConsume) {
      full = true;
    }
    empty = false;
    if (consumerWaiting) {
      consumer.consume(nextConsume);
      consumerWaiting = false;
    }
  }
  msgsrv ackConsume() {
    nextConsume = (nextConsume + 1) %
bufferCount;
    if (nextConsume == nextProduce) {
      empty = true;
    }
    full = false;
    if (producerWaiting) {
      producer.produce(nextProduce);
      producerWaiting = false;
    }
  }
}
activeclass Producer(5) {
  knownobjects {
    BufferManager bufferManager;
  }
  statevars {
  }
  msgsrv initial() {
    self.beginProduce();
  }
  msgsrv produce(int bufNum) {
    bufferManager.ackProduce();
    self.beginProduce();
  }
  msgsrv beginProduce() {
    bufferManager.giveMeNextProduce();
  }
}
activeclass Consumer(5) {
  knownobjects {
    BufferManager bufferManager;
  }
  statevars {
  }
  msgsrv initial() {
    self.beginConsume();
  }
  msgsrv consume(int bufNum) {
    bufferManager.ackConsume();
    self.beginConsume();
  }
  msgsrv beginConsume() {
    bufferManager.giveMeNextConsume();
  }
}
main {
  BufferManager bufferManager(producer,
consumer):();
  Producer producer(bufferManager):();
  Consumer consumer(bufferManager):();
}
```

Figure 2. Producer Consumer example

State variables are followed by message servers. Each reactive class includes an initial method as explained earlier. The buffer has two message servers provided to get the requests of the producer and consumer. Two other message servers get the acknowledgements of the producer and consumer and make the pointers and full/empty indicators up to date.

The producer have two message servers, the method *beginProduce* requires an empty space in the buffer by sending *giveMeNextProduce* message to the buffer, and the method *Produce* is called by the buffer to provide the index of the empty element available for the producer to put its product. By executing the method *Produce* an acknowledgement is sent to the buffer and a *beginProduce* message is sent to self to repeat the cycle of production. The body and behavior of the consumer is similar to the producer with the symmetric message servers.

3.2. Semantics

The operational semantics of a reactive system can be defined as a labeled transition system. Labeled transition system is a quadruple of a set of states (S), a set of labels (L), a transition relation on states (T), and a set of initial states of the system (S_0).

To define operational semantics of Rebeca, we first formalize the definitions of a rebec, a model, and their constituents (Figure 3). A rebec, r_i, with a unique identifier i, is defined as a triple $< V_i, M_i, K_i >$, where V_i is the set of its state variables, M_i is the set of its methods identifiers, and K_i is the set of all known rebecs of r_i. For a Rebeca model, there is a universal set $\mathcal{I}$ of all *rebec identifiers* that are involved in the model, and a universal set $\mathcal{K}$ of all *known rebecs* of all members of $\mathcal{I}$.

A message msg is defined as: $msg =< sendid, i, mtdid >$, where $sendid$ is the identifier of the sender, i is the identifier of the receiver, and $mtdid$ denotes the method of receiver r_i which is called when the message is received. For the sake of simplicity, we ignore the message parameters in our semantics definition. $\mathcal{U}$ is the set of all possible values for all types of variables that can be defined in a rebec, $\mathcal{V}_i = \{v | v : V_i \rightarrow \mathcal{U}\}$ is the set of possible values for variables of rebec i, and $\mathcal{V}_M = \bigcup_{i \in I_C} \mathcal{V}_i$.

Each rebec has a queue which can be defined as a finite sequence of messages. We denote the set of all finite sequences on a given set A as $seq(A)$. The mailbox of a component is like a multi-queue consisting of all the queues of its rebecs and including all the messages that have been sent from internal rebecs and have not yet been received. Operational semantics of a Rebeca model is defined as a labeled transition system $M = (S, L, T, s_0)$, and is shown in Figure 4.

The state space of the model is

$$\prod_{i=1}^{n} (S_i \times q_i), \tag{1}$$

where each S_i is a model of the local state of rebec r_i consisting of a valuation that maps each local field variable to a value of the appropriate type; and the inbox q_i, an *unbounded* buffer that stores all incoming messages (`<mir>`) for rebec r_i in a FIFO manner.

The set of action labels L is the set of all `<mir>` calls in the given `<model>`; such calls record the processing of those messages that are part of the target rebec provided message servers;

A triple $(s, l, s') \in S \times L \times S$ is an element of the transition relation T iff

- in state s there is some i ($1 \leq i \leq n$) such that l is the first message in the inbox q_i, l is of the form $< sendid, i, mtdid(vars) >$, and $sendid$ is the rebec identifier of the requester (sender rebec,

- r_i is a rebec with the unique identifier i, defined as $< V_i, M_i, K_i >$.
- V_i is the set of state variables of the rebec r_i.
- M_i is the set of methods identifiers of the rebec r_i.
- K_i is the set of all known rebecs of r_i.
- $\mathcal{I}$ is the set of all rebecs identifiers.
- $\mathcal{K}= \bigcup_{i \in \mathcal{I}} K_i$ is the set of known rebecs of all rebecs.
- $\mathcal{M}= ||_{i \in \mathcal{I}} r_i$ is the set of rebecs $\{r_i | i \in \mathcal{I}\}$ concurrently executing, making the closed model $\mathcal{M}$, and we have $V_{\mathcal{M}} = \bigcup_{i \in \mathcal{I}} V_i$, $M_{\mathcal{M}} = \bigcup_{i \in \mathcal{I}} M_i$, $K_{\mathcal{M}} = \bigcup_{i \in \mathcal{I}} K_i$.
- $msg =< sendid, i, mtdid >$ is a message sent by the rebec $sendid$ to call the method $mtdid$ of rebec i.
- $\mathcal{U}$ is the set of all possible values for all types of variables that can be defined in a rebec.
- $\mathcal{V}_i = \{v | v : V_i \rightarrow \mathcal{U}\}$ is the set of possible valuations for variables of rebec r_i.
- $\mathcal{V}_{\mathcal{M}} = \{v | v : V_{\mathcal{M}} \rightarrow \mathcal{U}\}$ is the set of possible valuations for variables of model $\mathcal{M}$.
- $Q_{\mathcal{M}} = \prod_{i \in I_{\mathcal{M}}} seq(I_{\mathcal{M}} \times M_i)$ is the set of possible states for the inbox of model $\mathcal{M}$, defined as a multi-queue. Each queue is defined as a finite sequence of messages corresponding to an internal rebec as the receiver.

Figure 3. Summery of Definitions.

implicitly known by the receiver), i is the rebec identifier of r_i (receiver rebec), and $mtdid$ is the name of the method m of r_i which is invoked, together with its parameters $vars$;

- state s' results from state s through the atomic execution of two activities: first, rebec r_i deletes the first message l from its inbox q_i, second, method m is executed in state s. The latter may add requests to rebecs' inboxes , change the local state, and create new rebecs;
- if new rebecs are created in the invocation of m, then the state space S *expands dynamically* from the one in (1) to

$$\Big(\prod_{i_{\text{new}}} (S_{i_{\text{new}}} \times q_{i_{\text{new}}})\Big) \times \prod_{i=1}^{n} (S_i \times q_i), \tag{2}$$

where i_{new} ranges over the new rebecs created within that method invocation and s' is an element of (2);

Clearly, the execution of the above methods relies implicitly on a standard semantic for the imperative code in the body of method m. Within such code, `<mir>` requests may be issued and rebecs may be created. In our semantics, messages (method invocation requests) (`<mir>`) are the sole mechanism for communication between these rebecs. Regarding the *infinite* behavior of our semantics, communication is assumed to be fair [3]: all `<mir>` requests eventually reach their respective inboxes and will eventually be invoked by the corresponding rebec. The initial state s_0 is the one where each rebec has its `initial` message as the sole element in its inbox.

Example 3.2. (Producer-consumer: Initial state)
In Example 3.1, in the initial state there are a buffer, a producer and a consumer with their *initial* methods in their inboxes. So, we have three enabled transitions. Execution of the *initial* methods may cause

Operational semantics of a model $\mathcal{M} = ||_{i \in \mathcal{M}} r_i$ is defined as a labeled transition system (S, L, T, s_0)

- $S = \prod_{i=1}^{n} S_i \times q_i$ is the set of states where $S_i = \mathcal{V}_i \times K_i$. $\mathcal{V}_i$ is the set of possible values for all the variables of rebec i, K_i is the set of known rebecs and known methods of r_i; and q_i is the set of possible states for the message queue.
- $L = \bigcup_{i \in \mathcal{I}} \mathcal{I} \times M_i$ is the set of labels, that are all possible messages that can be passed around in $\mathcal{M}$, where $\forall (x, y, m) \in L$ we have $m \in K_x$.
- $T \subseteq S \times L \times S$ is the set of transition relations on states, where
  $s_1 \rightarrow^{l} s_2 \in T$, iff $s_1, s_2 \in S$, and $l = (x, y, m) \in L$ is an enabled transition, which means
  $\exists i \in \mathcal{I} \wedge q \in Q \mid l = head(s_1.q.i)$,i.e., l is a message on top of the queue and
  s_2 results from s_1 and l as follows:
  - The message is popped from $s_1.q$, i.e., $s_2.q.y := tail(s_1.q.y)$.
  - Transition that is fired by message $l = (x, y, m)$ causes the method m of the rebec y to be executed as an atomic operation, in which:
    - Execution of ordinary statements in m may change the value of some variables of y ($s_1.v$), and
    - execution of each $send$ statement in m, changes the message queue($s_1.q$).
    - execution of each $create$ statement in m expands the state space S dynamically from $\prod_{i=1}^{n} S_i \times q_i$ to $\left(\prod_{i_e} S_{i_e} \times q_{i_e}\right) \times \prod_{i=1}^{n} S_i \times q_i$ where i_e is the created rebec.
- $s_0 = \mathcal{V} \times Q_0$ is the initial state of the model. Variables are initialized to their default values according to their types, and Q_0 is defined such that the queue of each rebec with identifier i includes only the message $(i, i, init_i)$. It is obvious that $s_0 \in S$.

Figure 4. Operational Semantics of a Rebeca Model.

sending messages to others or to *self*, and/or setting field variables. In the initial method of the rebec buffer instance variables are initialized; and in the initial methods of the rebecs producer and consumer messages *beginProduce* and *beginConsume* are sent to self.

Example 3.3. (Producer-consumer: state transitions)
Here we mention some of state transitions of the system.

- After execution of the $initial$ method of the producer, we have a *beginProduce* message in its inbox. When the *beginProduce* message in the inbox of producer is selected to be served, it is popped from the inbox and its code is executed by sending the messages *giveMeNextProduce* to the buffer. This *message* is added to the inbox of the buffer.
- Execution of *giveMeNextProduce* method by the buffer depends on the state variable $full$. If the $full$ variable is false, it causes sending a *Produce* message to the producer, if the $full$ variable is true the state variable $producerWaiting$ is changed to true to show that the producer is waiting for an empty place to put its product in.

Example 3.4. (Producer-consumer: Dynamic creation and topology)
Another version of producer-consumer example is shown in Figure 5. In this example we show dynamic creation of rebec buffer, and dynamic changing topology. Unlike in Example 3.1, where there is one instance of rebec buffer, and a constant number of buffer elements available, here we allow dynamic creation of instances of rebec buffer. Each buffer has one buffer element and a pointer to the next buffer.

```
reactiveclass Buffer(4) {
  knownobjects {
    Producer producer;
    Consumer consumer;
  }

  statevars {
    boolean empty;
    boolean consumerWaiting;
    Buffer nextBuffer;
  }

  msgsrv initial() {
    empty = true;
    consumerWaiting = false;
    nextBuffer = null;
  }

  msgsrv readyToProduce() {
    if (!full) {
      producer.produce();
    }
    else {
      nextBuffer = new Buffer(producer,
      consumer):();
      producer.setBuffer(nextBuffer);
    }
  }

  msgsrv readyToConsume() {
    if (!empty) {
      consumer.consume();
    }
    else if (nextBuffer != null) {
      consumer.setBuffer(nextBuffer);
    }
    else {
      consumerWaiting = true;
    }
  }

  msgsrv ackProduce() {
    empty = false;
    if (consumerWaiting) {
      consumer.consume();
      consumerWaiting = false;
    }
  }

  msgsrv ackConsume() {
    empty = true;
  }
}
```

```
reactiveclass Producer(2) {
  knownobjects {
    Buffer buffer;
  }
  statevars {
  }
  msgsrv initial() {
    self.beginProduce();
  }

  msgsrv produce() {
    buffer.ackProduce();
    self.beginProduce();
  }

  msgsrv beginProduce() {
    buffer.readyToProduce();
  }
  msgsrv setBuffer(Buffer b) {
    buffer = b;
    self.beginProduce();
  }
}

reactiveclass Consumer(2) {

  knownobjects {
    Buffer buffer;
  }
  statevars {
  }

  msgsrv initial() {
    self.beginConsume();
  }

  msgsrv consume() {
    buffer.ackConsume();
    self.beginConsume();
  }

  msgsrv beginConsume() {
    buffer.readyToConsume();
  }
  msgsrv setBuffer(Buffer b) {
    buffer = b;
    self.beginProduce();
  }
}

main {
  Buffer buffer(producer, consumer):();
  Producer producer(buffer):();
  Consumer consumer(buffer):();
}
```

Figure 5. Producer-Consumer Example (with Dynamic Behavior)

At the initial state one buffer is created, after one production the producer makes the buffer element of this buffer full. Then, for the next product another buffer is created. Buffer rebecs are like the nodes of a link list which are created on demand. The rebec consumer starts to consume from the first node and moves forward in this link list. For the sake of simplicity, we do not model releasing of the consumed nodes.

As shown in Figure 5, if a buffer is full and a producer asks for a space by sending *readyToProduce*, then a new buffer is created and its known objects are set to be the producer and consumer. Then, the rebec id of this new rebec is sent to the producer as a parameter of the message *setBuffer*. Execution of the message server of *setBuffer* will change the known object of the producer to the buffer which is newly created. This is an example of dynamic changing topology.

4. Model Checking Rebeca Models

In formal verification we try to prove or disprove that a model satisfies some specifications. There are two basic approaches to analysis: model checking and deductive methods. Typically, model checking is performed by an exhaustive simulation of the model on all possible inputs. In this case, a software tool performs the analysis. 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) [7].

For verifying the behavior of a model, we need a language to specify its properties. Temporal logic and automata are alternatively used for this purpose. Here we choose temporal logic as our property specification language. Model checking can only be applied on finite systems, so we use abstraction techniques to make our model finite. Unbounded message queues, unbounded data types, and unbounded creation of rebecs are not allowed. Another method for reducing the state space is the coarse grained granularity in the interleaving that models the concurrency of the system. Each method is executed as an atomic operation. Below, we describe these features in model checking Rebeca models in more detail.

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 V, where $V = \bigcup_{i \in \mathcal{I}} V_i$. 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. For model checking we abstract from the dynamic rebec creation and dynamic changing topology and consider it as the future work.

Bounded queues Finite-state model checkers are not able to deal with infinite state space, which is present in Rebeca due to the unboundedness of the queues' capacity. Thus, we need to impose an abstraction mechanism on our models: each rebec has a user-specified, finite upper bound on the size of its queue. The computation of the successor state s' of a transition (s, l, s') is as before, except that s' equals s (stuttering step) if the request l did not reach the filled-up queue of the target.

In general, unbounded queues still provide a major challenge in automatic verification, and it is common practice to introduce upper bounds for the queues. The presence of bounded queues may complicate the interpretation of the analysis results, but the possible counter-examples found by model

checkers can be used in debugging. There are also many cases that the number of sent messages does not increase unboundedly and so the queues stay bounded. In these cases, for keeping the semantics unchanged it is sufficient to choose a proper size for the queue. In some cases like in modeling security protocols we want the queues to be bounded to be able to model the overflow condition of the queues. Our contribution here is in the compositional verification approach, where the queues are abstracted from the external messages, and thus external messages do not give rise to overflow (discussed in Section 5).

Atomic execution of each method As we do not have any explicit receive statement in Rebeca, and as we do not have any shared variable among rebecs, we can execute without loss of generality a method atomically. More specifically, all generated messages can be sent at the end of each method execution preserving the order. In general for model-checking purposes we have to assume for each possible loop in a method a static given upper bound of its iterations. Consequently a program with such loops can be compiled into an equivalent program without loops.

A front-end tool for model checking For model checking Rebeca codes we developed a tool which is explained in Section 6. Using this tool we can translate Rebeca codes to SMV [1] or Promela [2] and model check it by existing model checkers. In these tools, we have bounded data types, bounded message queues and in the future version a bounded number of rebec creation. Property specification language is based on the specification languages of back-end model checkers: LTL and CTL. The execution of each method is accomplished as an atomic operation. Message blocks are not implemented yet. A queue length, which can be different for each rebec, is provided by the tool and is defined by the modeler. The queue overflow can be checked as a property by the tool, and the queue length can be increased if necessary. In the models which the number of sent messages does not increase unboundedly, and thus the queue is bounded itself, this facility made it possible to prevent the queue overflow and to keep the semantics of the original model unchanged. An ongoing project is developing another tool for generating the state space from Rebeca codes and then model check it directly.

Example 4.1. (Producer-consumer: Model checking the code)
The producer-consumer Rebeca model in Figure 2 (with static behavior) is translated to SMV using our tool and then it is model checked using NuSMV. The total state space generated by NuSMV includes $2.14e14$ states and the reachable state are 374 states. The safety properties:
$\Box!(buffer.full \wedge buffer.empty)$ and
$\Box(!buffer.empty \wedge !buffer.full) \rightarrow$
$!(buffer.nextProduce = buffer.nextConsume)$
are checked and are both true.

5. Compositional Verification

One of the most important problems in model checking is the state-explosion problem. Compositional verification is a way to tackle this problem. In compositional verification the goal is to check properties of the components of a system and deduce global properties from these local properties. The main difficulty with this approach is that local properties are often not preserved at the global level [12].

In compositional verification, the specification of a system is decomposed into the properties of its components which are then verified separately. If we deduce that the system satisfies each local

property, and show that the conjunction of the local properties implies the overall specification, then we can conclude that the system satisfies this specification too [24, 11, 32]. There has been a strong trend to use compositional approaches in formal verification of systems [23, 43, 46, 41]. The closest approach to our work is [38].

An overview In its simplest form assume a system consists of two modules P and Q which communicate with each other and also with their environment. We show this system as $P||Q$. If φ_P is the specification of P ($P \models \varphi_P$) and φ_Q is the specification of Q ($Q \models \varphi_Q$), we would like to reason according to the following rule [31]:

$$\frac{\begin{array}{l} P \models \varphi_P \\ Q \models \varphi_Q \\ \varphi_P \wedge \varphi_Q \Rightarrow \varphi \end{array}}{P||Q \models \varphi}$$

As mentioned above, the local property φ_P does not necessarily hold after P is composed with Q. To use the above rule, the composed system should maintain inherent properties of its components. In other words composition of P and Q should not alter φ_P and φ_Q in the whole system.

In addition, a component of a system is typically designed to work only in the environment of that system. Thus, the module P does not necessarily satisfy the useful property that we need in an arbitrary environment. The reachable state space of P in any possible environment may in fact be much larger than the state space of P composed with Q. This is called *environment problem*. Two possible solutions for this problem are *compositional minimization* and *assume/guarantee reasoning*. In compositional minimization a reduced version of Q, say Q', is derived that characterizes just the behavior of Q that is visible to P. Q' is called an *interface process* and models a reduced environment. The property of $P||Q'$ can also be stated for $P||Q$. In assume/guarantee reasoning, φ_P is specified and guaranteed regarding some assumptions about the environment which have to be satisfied by Q. In our compositional verification approach we model a reduced environment which can be considered as compositional minimization. We also apply some further abstractions on the reduced environment which is explained in the following section. We may also need some assumptions about the environment to prove certain properties, but in general we do not apply assume/guarantee approach.

5.1. Compositional Verification in Rebeca: Components

In general, compositional verification may be exploited more effectively when the model is naturally decomposable [35]. In particular, a model consisting of inherently independent modules is suitable for compositional verification. Our actor-based model provides such independent modules because of the asynchronous communication mechanism which involve only an explicit non-blocking send operation.

Decomposition into components In Rebeca, for verification purposes we may *decompose* a closed model and think of one part as a component which is an open system and the remainder as the environment that makes the overall system closed. This de-composition, determines which rebecs in the model have to be modeled with state and behavior (the component) and which rebecs may be abstracted such that they only *send* messages (the environment).

Modeling environment Since environment rebecs never execute their own methods, there is no need to model their inboxes, state, or behaviors. In a Rebeca component model, we call environment rebecs *external* and all other rebecs *internal*.

Abstracting environment This de-composition process abstracts the model considerably: only internal rebecs are fully modeled; external rebecs are only modeled in their capacity to request remote method invocations (sending messages). So, they are only modeled as the set of external messages that can be sent by them. This set of external messages represent the environment for the component.

Abstracting the queues from external messages Instead of putting external messages in an internal's inbox, they may be processed at any time, up to fair interleaving with the processing of requests in the inbox. This makes the model checking more efficient. Formally the behavior of the environment of a component is modeled by additional transitions which describe its messages sent to the component. In other words with respect to the external environment, a component behaves like an I/O automata [26], where inputs from the environment are always enabled.

External messages attached to components External messages coming into the component are present in all the states and we can imagine that they are like the members of a set that is constantly attached to all the states in the corresponding labeled transition system. In this way we abstract from buffering the external messages, and we do not need to have a special rebec or component modeling an environment. The environment of each component is modeled as extra transitions, added to operational semantics of a component. It is shown in the definition of the set of transition relations in the labeled transition system of Figure 7.

Abstracting from parameters and dynamic topology For the sake of simplicity, we abstracted the method definitions from their parameters. Methods with variables which range over finite data types as parameters, can be modeled as multiple methods with no parameters. Consequently, assuming a statically given a priori upper bound to the number of created objects we can model a restricted form of dynamic topology.

Dynamic creation of rebecs In the compositional verification approach, the behavior of internal rebecs of a component is fully modeled without any abstraction. So, dynamic creation of internal rebecs can be also modeled naturally. By abstracting the environment we model it with a constant set of external messages. If we assume an environment which is dynamically changed by creation of new rebecs, then the set of external messages can be considered as a constant set only if the behavior of internal rebecs does not depend on the sender of a message. As the set of active classes is a constant, and new rebecs are created from this constant set of active classes we can still model the environment as a constant set of external messages.

If in the code of a rebec, there is an explicit reference to the sender of a message then the behavior of the receiver depends on the sender of the message and our abstraction no more preserves the original behavior. For the sake of simplicity, in the definition of operational semantics of a component (Figure 7), we do not consider dynamic creation of rebecs (nor internal and external).

Deciding on how to decompose Internal rebecs constitute the "focus" of a particular analysis. Determination of such a focus may often be the result of intuition and experience with similar patterns of open systems and depends on the properties which have to be proved. It is the responsibility of the modeler and cannot be fully automated, although some work has been done in automating this process and eliminating user guidance [6]. There is no general approach in decomposing the system in components, components have to be selected carefully to lead to a smaller state space [24]. In many cases, specially when there is symmetry in the model, we can reduce the state space significantly.

Composing two components With the decomposition technique the universe of rebecs is always known. The active classes in the closed system designates this set. Given a model as the universe of rebecs, any (finite) subset thereof can be the set of internal rebecs of some Rebeca component. Given two such components, we are able to compose them into another component. The resulting component is the union of internal rebecs of the constituents. Internal and external messages can be obtained knowing the universe of rebecs and internal rebecs. Note that decomposing a given close model is different from composing open components which are defined in an unknown environment.

- $C = ||_{i \in I_C} r_i$ is a set of rebecs $\{r_i | i \in I_C\}$ concurrently executing, and we have $V_C = \bigcup_{i \in I_C} V_i$, $M_C = \bigcup_{i \in I_C} M_i$, $K_C = \bigcup_{i \in I_C} K_i$.
- $I_C \subseteq \mathcal{I}$ is the set of identifiers of internal rebecs of C.
- $C = ||_{i=1}^{n} C_i$ is the parallel composition of n components $C_i, i = 1, ..., n$, and we have $I_C = \bigcup_{i=1}^{n} I_{C_i}$, $V_C = \bigcup_{i=1}^{n} V_{C_i}$, $M_C = \bigcup_{i=1}^{n} M_{C_i}$, $K_C = \bigcup_{i=1}^{n} K_{C_i}$.
- $\mathcal{V}_C = \{v | v : V_C \rightarrow \mathcal{U}\}$ is the set of possible valuations for variables of component C.
- $Q_C = \prod_{i \in I_C} seq(I_C \times M_i)$ is the set of possible states for the inbox of component C, defined as a multi-queue. Each queue is defined as a finite sequence of messages corresponding to an internal rebec as the receiver.

Figure 6. Summary of definitions (cont.).

The definitions for components are formalized in Figure 6 and operational semantics of a component C is summarized in Figure 7 [39].

Example 5.1. (A component in a Rebeca model)
In our producer-consumer example with no dynamic behavior (Figure 2), we can take rebecs $buffer$ and $producer$ as an open component, and the consumer as environment. This component can be denoted by $buffer||producer$. The external messages coming to the component are *ackConsume* and *giveMeNextConsume* messages from the consumer to the buffer. We assume these messages are always enabled.

Example 5.2. (Composition of components in a Rebeca model)
If we compose two components $buffer||producer$ and $buffer||consumer$, we will have $buffer||producer||consumer$. It is the union of internal rebecs which made a closed system here. Internal and external messages can be obtained knowing the universe of rebecs and internal rebecs.

5.2. Formal Justifications

The state explosion problem may be avoided by using techniques that replace a large component by a smaller component which satisfies the same properties. We need a notion of equivalence or preorder

Operational semantics of a component $C = ||_{i \in I_C} r_i$ is defined as a labeled transition system (S_C, L_C, T_C, s_{C_0})

- $S_C = \prod_{i=1}^{n} S_i \times q_i$ is the set of component states where $S_i = \mathcal{V}_i \times K_i$. $\mathcal{V}_i$ is the set of possible values for all the variables of rebec i, K_i is the set of known rebecs and known methods of r_i; and q_i is the set of possible states for the message queue.
- $L_C = \bigcup_{i \in I_C} \mathcal{I} \times M_i$ is the set of labels, that are all possible messages to C, where $\forall (x, y, m) \in L_C$ we have $m \in K_x$.
- $T_C \subseteq S_C \times L_C \times S_C$ is the set of transition relations on states, where
  $s_1 \rightarrow^{l} s_2 \in T_C$, iff $s_1, s_2 \in S_C$, and $l = (x, y, m) \in L_C$ is an enabled transition, which means
  $\exists i \in I_C \wedge q_C \in Q_C \mid l = head(s_1.q_C.i)$,i.e., l is an internal message on top of the queue or $l.x \notin I_C$, i.e., l is an external message; and
  s_2 results from s_1 and l as follows:
  - If $x \in I_C$ then the message is popped from $s_1.q_C$, i.e., $s_2.q_C.y := tail(s_1.q_C.y)$, otherwise $s_1.q_C$ does not change.
  - Transition that is fired by message $l = (x, y, m)$ causes the method m of the rebec y to be executed as an atomic operation, in which:
    - Execution of ordinary statements in m may change the value of some variables of y ($s_1.v_c$), and
    - execution of each $send$ statement in m,
      if it is a send to an internal rebec, changes the message queue($s_1.q_C$),
      otherwise it has no effect on the state.
- $s_{C_0} = \mathcal{V}_C \times Q_{C_0}$ is the initial state of the component. Variables are initialized to their default values according to their types, and Q_{C_0} is defined such that the queue of each rebec with identifier i includes only the message $(i, i, init_i)$. It is obvious that $s_{C_0} \in S_C$.

Figure 7. Operational Semantics of a Component.

among structures guaranteeing that two components satisfy the same set of formulas in a given logic or that certain properties are preserved.

A simulation relates a component to an abstraction of that component. Because the abstraction can hide some of the details of the original structure, it might have a smaller set of state variables. The simulation guarantees that every observable behavior of a component is also a behavior of its abstraction. However, the abstraction might have behaviors that are not possible in the original component.

Weak simulation and property preservation Now, we explain the weak simulation relation among components in our model. Here, the model is simplified by ignoring dynamic creation and dynamic topology. Therefore, referring to the operational semantics of models in Section 3.2, the state space S won't expand dynamically from formula (1) to formula (2).

As explained before, external messages coming into the component are present in all the states and we can imagine that they are like the members of a set that is constantly attached to all the states in the corresponding labeled transition system. So, in each state, we have a set of variables, a message (multi-)queue, and also a set of external messages. Because the set of external messages is constant in all states, we do not need to consider it in each state.

To define the weak simulation relation between two components, we use the operational semantics definition in Section 3.2 and the component definition in this section, and the following notation. A component C is a set of rebecs, the set of identifiers of internal rebecs of C is denoted by I_C and its

state by s_C. The set of valuations for variables of component C in state s_C is denoted by $s_C.\mathcal{V}_C$. The inbox of component C is defined as a multi-queue, each queue is defined as a finite sequence of messages corresponding to an internal rebec as the receiver. The multi-queue of component C in state s_C is denoted by $s_C.q_C$. As explained in Section 3.2, a label is a message of the form $< sendid, i, mtdid >$, where $sendid$ is the identifier of the sender rebec, i is the identifier of the receiver rebec, and $mtdid$ designates the method of i to be executed.

We also define a projection relation between two states. State $s_{C'}$ is a projection of state s_C (denoted by $s_{C'} \uparrow s_C$), if (1) $I_{C'} \subseteq I_C$; (2) the variables of their common rebecs have the same values, i.e., $s_{C'}.\mathcal{V}_{C'} \subseteq s_C.\mathcal{V}_C$; and (3) the multi-queue $s_C.q_{C'}$ is a projection of $s_C.q_C$.

The multi-queue $q_{C'}$ is a projection of the multi-queue q_C (denoted by $q_{C'} \uparrow q_C$), if $I_{C'} \subseteq I_C$ and for each $i \in I_{C'}$ the sequence of messages $< sendid, i, mtdid >$ in q_C, ignoring messages with $sendid \in I_C - I_{C'}$, is the same as the sequence of messages in $q_{C'}$.

With this terminology, we now define the weak simulation relation.

Definition 1. ***(Weak simulation)*** Given two components C and C' of a given model, represented by labeled transition systems (S_C, T_C, s_{0_C}) with signature of action labels L_C and $(S_{C'}, T_{C'}, s_{0_{C'}})$ with signature of action labels $L_{C'}$, such that $I_{C'} \subseteq I_C$:

1. A relation $H \subseteq S_C \times S_{C'}$ is a weak simulation relation between C and C' if and only if for all $s_C \in S_C, s_{C'} \in S_{C'}$, if $H(s_C, s_{C'})$, then the following conditions hold:
   (a) $s_{C'} \uparrow s_C$.
   (b) for every state s_{C_1} and label $l \in L_C$ such that $(s_C, l, s_{C_1}) \in T_C$, there is a state $s_{C'_1}$ with the property that $s_{C'_1} = s_{C'}$ (if $l \notin L_{C'}$) or $(s_{C'}, l, s_{C'_1}) \in T_{C'}$ (if $l \in L_{C'}$) and $H(s_{C_1}, s_{C'_1})$.
2. We say that C' weakly simulates C (denoted by $C \leq C'$) if there exists a weak simulation relation H between C and C' such that $H(s_{C_0}, s_{C'_0})$.

Next we introduce a theory which provides a formal justification of our compositional verification technique of a component-based model. This theory consists of two theorems, one theorem which semantically characterizes the behavior of a component in the context of a given closed model in terms of the above weak simulation relation, and a general theorem which provides the semantic characterization of the logic in terms of the weak simulation relation.

Theorem 1. ***(Weak simulation relation between a component and its composition with another arbitrary component)*** For any two components C' and X of a model (defined on the same universal set of rebecs), C' weakly simulates $C = C'||X$.

proof. Consider $H = \{(s_C, s_{C'}) \in S_C \times S_{C'} \mid s_{C'} \uparrow s_C\}$. We have to show (1) that H is a weak simulation and (2) $H(s_{C_0}, s_{C'_0})$.

1. To show that H is a weak simulation:
   (a) $s_{C'} \uparrow s_C$ is in the definition of H.
   (b) For the second condition, let $H(s_C, s_{C'})$ and $l \in L_C$ such that $(s_C, l, s_{C_1}) \in T_C$.

i. If $l \notin L_{C'}$ then $s_{C'}$ stays unchanged, i.e., $s_{C'_1} = s_{C'}$ and we still have $H(s_{C_1}, s_{C'_1})$. But $l \notin L_{C'}$ means that l is a message to rebecs in the component X, i.e., $l = (p, r, m),\ r \in I_X,\ r \notin I_{C'}$. In this case m will be executed and so the variables of C' ($V_{C'}$) remain unchanged, and also messages that may be sent by m are not put into the multi-queue of C'. Thus, $q_{C'}$ won't be changed either and therefore $H(s_{C_1}, s_{C'_1})$.

ii. If $l \in L_{C'}$, it means that $r \in I_{C'}$, when $l = (p, r, m)$. We have to show that l is enabled in $s_{C'}$, and then also show that $s_{C'_1} \uparrow s_{C_1}$. First, we show that l is enabled in $s_{C'}$ in all the possible conditions:

- l is external for both C and C'. We know that $I_{C'} \subseteq I_C$, so $I'_C \subseteq I'_{C'}$ and the set of external messages to C is a subset of external messages to C'. Thus, l is enabled in $s_{C'}$.
- l is internal for C and external for C'. It means that l is a message coming from a rebec in X, e.g., $p \in X$. When l is an external message for C', it is always enabled in all states, so it is enabled in $s_{C'}$.
- l is internal for both C and C'. We know that $H(s_C, s_{C'})$, so $s_{C'} \uparrow s_C$ and also $q_{C'} \uparrow q_C$. From the definition of projection we know that if l is on the top of the queue in s_C, it has to be on the top of the queue for $s_{C'}$ too. Thus, l is enabled in $s_{C'}$.

Second, we prove that $s_{C'_1} \uparrow s_{C_1}$ is the same for all three cases.

- execution of m causes the same changes on variables of both components (just the variables in r); and
- it may send some messages to rebecs in C', causing the same changes in both queues of s_C and $s_{C'}$; or it may send messages to rebecs in X, making $s_{C'_1}.q_{C'}$ to be different from $s_{C_1}.q_C$ but still guaranteeing $q_{C'} \uparrow q_C$ and so $s_{C'_1} \uparrow s_{C_1}$.

2. Now we show that $s_{C'_0} \uparrow s_{C_0}$. This follows from the definition of the initial state in the operational semantics of components: $s_{C'_0}.\mathcal{V}_{C'} \subseteq s_{C_0}.\mathcal{V}_C$; furthermore, $s_{C'_0}.q_{C'} \uparrow s_{C_0}.q_C$, because there are only init messages in both of them.

Definition 2. ***(Satisfaction relation)*** 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.

We have the following theorem which restricts the corresponding theorem of Clark et al [11] to safety properties.

Theorem 2. ***(Property preservation)*** If C' weakly simulates C, then for every safety property specified by an LTL-X formula ϕ (LTL without the next operator), with atomic propositions on variables in C', $C' \models \phi$ implies $C \models \phi$.

Compositional verification Using Theorem 2 we have the following corollary for compositional verification of LTL-X safety properties. $R = ||_{i=1}^{n} X_i$ is the parallel composition of n components $X_i, i = 1, ..., n$ and we have $I_R = \bigcup_{i=1}^{n} I_{X_i}$.

Corollary 1. (Compositional verification) Let $R = ||_{i=1}^{n} X_i$ and φ_{X_i} be a safety property of X_i specified in LTL-X. In order to show that φ_R is a property of system R, it suffices to find properties for each X_i, such that,

1. For $i = 1, \ldots, n, \quad \varphi_{X_i}$ is a property of X_i, and
2. $(\bigwedge_{i=1}^{n} \varphi_{X_i}) \Rightarrow \varphi_R$ is valid.

We can prove for $i = 1, \ldots, n$, $X_i \models \varphi_{X_i}$ by model checking. After that if $(\bigwedge_{i=1}^{n} \varphi_{X_i}) \Rightarrow \varphi_R$ then φ_R is a property of R.

There are no conditions on selected components. But, obviously it is better to put highly interacting rebecs in a component. It would also be better to select loosely coupled components for model checking in order to decrease the number of external messages. Sometimes, we need to share some rebecs between some components. Theorem 2 holds in this situation too. Hence, we can use Corollary 1.

Sometimes a system consists of similar components in which case we can use a kind of generalization. We say two components are similar when they consist of the same number of rebecs and for each rebec in one there is a corresponding rebec in the other component, and both rebecs are instantiated from the same class. Since all instances of a class have similar properties, so have all similar components. The modeler chooses a component which its parallel composition with a number of other similar ones makes up the total system. S/he verifies the property of this component by model checking and it is generalized to other similar ones. Then, the rest is done by using Corollary 1.

Example 5.3. (Producer-consumer: verification of a property using abstraction)
The critical section in producer/consumer example is the buffer which is a shared object. The system safety requirement is that at any given time the producer and consumer do not access the buffer simultaneously. It is specified in LTL-X as follows:

$$\varphi_{sys} = \Box(!bufferManager.empty \wedge !bufferManager.full) \rightarrow \\ !(bufferManager.nextProduce = bufferManager.nextConsume)$$

Here, the property of the system is localized to a property of one rebec: the $bufferManager$. So, we can pick the $bufferManager$ as the component and the rest of the system as its environment. The desired property is proven by model checking and shows that the system satisfies its safety requirement. The reachable states generated by NuSMV in model checking this example, consisting of one producer and two consumers is 6936. Using our compositional verification approach and assuming the $bufferManager$ as a component the reachable states will reduce to 2562. By increasing the number of consumers to four we have 817 million reachable states for the closed world model and $17,930$ reachable states using compositional verification approach.

In Example 5.3, we only use Theorem 1 and Theorem 2 to prove the property of the system by model checking a component. The following example shows how we need to use also Corollary 1 in order to prove the desired property.

Example 5.4. (Compositional verification of a property)
Consider the dining philosophers example. There are n philosophers at a round table. To the left of each philosopher there is a fork, but s/he needs two forks to eat. Of course only one philosopher can use a fork at a time. If the other philosopher wants it, s/he just has to wait until the fork is available again. The system safety requirement is that at any given time two neighboring philosophers cannot both hold the fork between them.

If we denote the ith philosopher by $Phils_i$ and the ith fork by $Fork_i$, and each philosopher has two boolean variables FR and FL, indicating if the right and the left fork is in its hand, then the safety property of the system is specified in LTL-X as follows ($\oplus$ denotes addition in mod n, and n is 4 in our example):

$$\varphi_{sys} = \Box(\bigwedge_{i=0}^{n-1} \neg(Phils_i.FR \wedge Phils_{i\oplus 1}.FL))$$

By composing $Phils_0$, $Phils_1$ and $Forks_1$ as a component, we have:

$$\varphi_{Phils_0||Forks_1||Phils_1} = \Box(\neg(Phils_0.FR \wedge Phils_1.FL))$$

This property is proven by model checking and we need to use Corollary 1 to conclude the property of the system, φ_{sys}:

$$\bigwedge_{i=0}^{n-1} \varphi_{Phils_i||Forks_{i\oplus 1}||Phils_{i\oplus 1}} \Rightarrow \varphi_{sys}$$

Here we have 4 similar components, $Phils_i||Forks_{i\oplus 1}||Phils_{i\oplus 1}, i = 0, ..., 4$Å philosopher is shared between each two components, and all of these components have similar properties. The model checking results and reduction in the state space can be find at *http://khorshid.ut.ac.ir/∼rebeca*.

6. Using a Tool for Model Checking Rebeca

Rebeca Group in Tehran and Sharif Universities developed two front-end tools for model checking Rebeca models. These tools translate Rebeca models to SMV and Promela code, then these codes can be model checked by the back-end model checkers NuSMV [1] and Spin [2]. An integrated tool, Rebeca Verifier, is also developed which includes the front-end translator tools and also support the compositional verification approach [42].

For automatic abstraction, the modeler selects a subset of rebecs in a Rebeca model to create a component. This will generate an open system. The rebecs which are now interacting with the outside world and their interface with the environment are all determined and visualized. The tool determines the external rebecs which interact with the component as its environment, and a Rebeca code is automatically generated for this component model.

Rebeca Verifier is used to apply compositional verification approach on some case studies. Rebeca code and the properties that are checked for each case study can be found at *http://khorshid.ut.ac.ir/∼rebeca* or at *http://mehr.sharif.edu/∼msirjani/rebeca*. The results show how our approach reduces the state space in these case studies significantly.

7. Conclusion and Future Work

Actor-based modeling can help the modeler via its encapsulated constructs and formal verification can be used to design more reliable systems. Compositional verification is needed to apply formal verification in practice. It is most effective when the model is modular and the modules are encapsulated and loosely coupled.

In Rebeca, we have independent reactive objects called *rebec*s which run concurrently and communicate by asynchronous message passing. There is an unbounded message queue for each rebec. We have classes for declaring the rebecs in the model. Therefore it is possible to reuse the code and simplify the verification process. A system can be decomposed into components that are executed concurrently. We can first verify properties of these components, specified in LTL-X or ACTL by model checking, and then conclude the overall system property using these latter results.

We use abstraction and symmetry to tackle state explosion problem. The asynchronous nature of message passing in Rebeca, let us to use coarse-grained transitions which reduce the state space and make the model simpler. Abstracting from message queues in specifying system properties introduces some kind of abstraction. We also use symmetry to simplify our verification process when there are replicated components in the system.

We defined our compositional verification approach on the simplified version of Rebeca ignoring certain dynamic behavior. Reformulating the specification language and the developed theory of compositional verification, considering dynamic object creation is planned for the future. Currently we are limited by the data types provided by the back-end model checkers and the size of state space generated. Investigating other abstraction techniques to handle unbounded queues is considered in our future project to model check Rebeca models directly.

Another line of research concerns a very promising extension of our compositional verification method to high-level components which generalize rebecs to *sets of reactive classes* with well-defined interfaces. Abstracting from the internal class structure such a component behaves as a rebec in Rebeca. This work is carried out in the context of the IST-2001-33522 EU project Omega [13] on the correct development of real-time embedded systems in UML.

Acknowledgement

We wish to thank Michael Huth for his helpful suggestions and discussions. Ramtin Khosravi also made valuable comments on actor semantics and Mohammad Reza Mousavi helped us on the compositional verification approach for Rebeca. Also, we wish to thank Farhad Arbab, Christel Baier, Marcello Bonsangue, Dennis Dams, Farzan Fallah, Jozef Hooman, Clemens Kupke, and Jan Rutten for their useful comments. Rebeca Group in University of Tehran, specially Hamed Iravanchi and Mohammad Mahdi Jaghouri supported us continuously, not only in developing the tool but also by their smart, creative, and inspiring comments. We wish to thank them all.

References

[1] NuSMV User Manual, Availabe through http://nusmv.irst.itc.it/NuSMV/ userman/index-v2.html.

[2] Spin User Manual, Available through http://netlib.bell-labs.com/netlib/spin/whatisspin.html.

[3] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, USA, 1990.

[4] Agha, G.: The Structure and Semantics of Actor Languages, in: *Foundations of Object-Oriented Languages* (J. W. de Bakker, W.-P. de Roever, G. Rozenberg, Eds.), Springer-Verlag, Berlin, Germany, 1990, 1–59.

[5] Agha, G., Mason, I., Smith, S., Talcott, C.: A Foundation for Actor Computation, *Journal of Functional Programming*, **7**, 1997, 1–72.

[6] Alur, R., de Alfaro, L., Henzinger, T. A., Mang, F. Y. C.: Automating Modular Verification, *Proceedings of CONCUR: 10th International Conference on Concurrency Theory*, LNCS 1664, Springer-Verlag, Berlin, 1999, 82-97.

[7] Alur, R., Henzinger, T. A.: *Computer Aided Verification*, Technical Report Draft, 1999.

[8] Alur, R., Henzinger, T. A.: Reactive Modules, *Formal Methods in System Design: An International Journal*, **15**(1), July 1999, 7–48.

[9] Alur, R., Henzinger, T. A., Mang, F. Y. C., Qadeer, S.: MOCHA: Modularity in Model Checking, *Proceedings of CAV'98*, LNCS 1427, Springer-Verlag, Berlin, 1998, 521-525.

[10] America, P., de Bakker, J., Kok, J.N., Rutten, J.J.J.M.: Denotational Semantics of a Parallel Object-Oriented Language, *Information and Computation* , **83**(2), 1989, 152–205.

[11] Clarke, E. M., Grumberg, O., Peled, D. A.: *Model Checking*, The MIT Press, Cambridge, Massachusetts, 1999.

[12] Clarke, E. M., Long, D. E., McMillan, K. L.: Compositional Model Checking, *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989, 353-362.

[13] Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML, *Proceedings of Formal Methods for Components and Objects*, LNCS 2852, Springer-Verlag, Berlin, Leiden, The Netherlands, 2003, 71-98.

[14] de Boer, F.S.: A Proof System for the Language POOL, *Proceedings of the REX School/Workshop Foundations on Object-Oriented Languages*, LNCS 489, Springer-Verlag, Berlin, 1991, 124-150.

[15] Dwyer, M., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C., Robby, , Visser, W., Zheng, H.: Tool-Supported Program Abstraction for Finite-State Verification, *Proceedings of the 23nd International Conference on Software Engineering*, 2001, 177-187.

[16] E.A. Emerson: Temporal and Modal Logic, *Handbook of Theoretical Computer Science* (J. van Leeuwen, Ed.), B, Elsevier Science Publishers, Amsterdam, 1990, 996-1072.

[17] Gaspari, M., Zavattaro, G.: An Actor Algebra for Specifying Distributed Systems: The Hurried Philosophers Case Study, *Proceedings of Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, LNCS 2001, 2001, 428-444, ISSN 0302-9743.

[18] Havelund, K., Pressburger, T.: Model Checking Java Programs using Java PathFinder, *International Journal on Software Tools for Technology Transfer*, **2**(4), 2000, 366-381.

[19] Hewitt, C.: *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.

[20] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[21] Ioustinova, N., Sidorova, N., Steffen, M.: Closing Open SDL-Systems for Model Checking with DTSpin, *Proceedings of FME'2002*, LNCS 2391, Springer-Verlag, Berlin, Germany, 2002, 531-548, ISSN 0302-9743.

[22] Kesten, Y., Pnueli, A.: Modularization and Abstraction: The Keys to Practical Formal Verification, *Proceedings of 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, LNCS 1450, Springer-Verlag, Berlin, Germany, 1998, 54-71, ISSN 0302-9743.

[23] Kupferman, O., Vardi, M. Y., Wolper, P.: Module Checking, *Information and Computation*, **164**(2), 2001, 322–344.

[24] Lamport, L.: Composition: A Way to Make Proofs Harder, *Proceedings of COMPOS: International Symposium on Compositionality: The Significant Difference*, LNCS 1536, Springer-Verlag, Berlin, Germany, 1997, 402-407.

[25] Larsen, K. G., Milner, R.: A Compositional Protocol Verification Using Relativized Bisimulation, *Information and Computation*, **99**(1), July 1992, 80–108.

[26] Lynch, N.: *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CS, 1996, ISBN SBN 1-55860-348-4.

[27] Lynch, N. A., Tuttle, M. R.: *Hierarchical correctness proofs for distributed algorithms*, Technical Report MIT/LCS/TR-387, MIT, 1987.

[28] Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, Berlin, Germany, 1992.

[29] Manna, Z., Pnueli, A.: *Temporal Verification of Reactive Systems (Safety)*, Springer-Verlag, Berlin, Germany, 1995.

[30] Mason, I. A., Talcott, C. L.: Actor languages: Their syntax, semantics, translation, and equivalence, *Theoretical Computer Science*, **220**(2), June 1999, 409–467, ISSN 0304-3975.

[31] McMillan, K.: *Verification of Digital and Hybrid Systems*, Springer-Verlag, Berlin, Germany, 2000.

[32] McMillan, K. L.: A methodology for hardware verification using compositional model checking, *Science of Computer Programming*, **37**(1–3), May 2000, 279–309, ISSN 0167-6423.

[33] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, *Information and Computation*, **100**(1), September 1992, 1–77.

[34] Ren, S., Agha, G.: RTsynchronizer: language support for real-time specifications in distributed systems, *ACM SIGPLAN Notices*, **30**(11), November 1995, 50–59, ISSN 0362-1340.

[35] de Roever, W. P., Langmaack, h., Pnueli, A., Eds.: *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures*, LNCS 1536, Springer-Verlag, Berlin, 1998.

[36] Roscoe, W. A.: *Theory and Practice of Concurrency*, Prentice-Hall, 1998.

[37] Schacht, S.: Formal Reasoning about Actor Programs Using Temporal Logic, *Proceedings of Concurrent Object-Oriented Programming and Petri Nets*, LNCS 2001, Springer-Verlag, Berlin, 2001, 445-460.

[38] Sidorova, N., Steffen, M.: Embedding Chaos, *Proceedings of Static Analysis Symposium (SAS'01)* , LNCS 2126, 2001, 319–334, ISSN 0302-9743.

[39] Sirjani, M., Movaghar, A.: *An Actor-Based Model for Formal Modelling of Reactive Systems: Rebeca*, Technical Report CS-TR-80-01, Tehran, Iran, 2001.

[40] Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M., Shali, A.: Model Checking Rebeca by SMV, *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'03)*, Southampton, UK, April 2003, 233-236.

[41] Sirjani, M., Movaghar, A., Mousavi, M.: Compositional Verification of an Object-Based Reactive System, *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01)*, Oxford, UK, April 2001, 114-118.

[42] Sirjani, M., Shali, A., Jaghoori, M., Iravanchi, H., Movaghar, A.: A Front-End Tool for Automated Abstraction and Modular Verification of Actor-Based Models, *Proceedings of ACSD 2004*, IEEE Computer Society, 2004, 145-148.

[43] Stahl, K., Baukus, K., Lakhnech, Y., Steffen, M.: Divide, abstract and model-check, *Proceedings of the 5th International SPIN Workshop on Theoretical Aspects of Model Checking*, LNCS 1680, Springer-Verlag, Berlin, 1999.

[44] Talcott, C.: Composable Semantic Models for Actor Theories, *Higher-Order and Symbolic Computation*, **11**(3), December 1998, 281–343, ISSN 1388-3690.

[45] Talcott, C.: Actor theories in rewriting logic, *Theoretical Computer Science*, **285**(2), August 2002, 441–485, ISSN 0304-3975.

[46] Tsay, Y.: Compositional Verification in Linear-Time Temporal Logic, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS'00)*, LNCS 1784, 2000, 344–358, ISSN 0302-9743.

[47] Varela, C., Agha, G.: Programming Dynamically Reconfigurable Open Systems with SALSA, *ACM SIGPLAN Notices*, **36**(12), 2001, 20–34.

[48] Yonezawa, A.: *ABCL: An Object-Oriented Concurrent System*, Series in Computer Systems, MIT Press, 1990.