Journal of Universal Computer Science, vol. 11, no. 6 (2005), 1054-1082
submitted: 25/8/04, accepted: 4/6/05, appeared: 28/6/05 © J.UCS

Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models

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)

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

Amin Shali
(Department of Electrical and Computer Engineering
University of Tehran
Karegar Ave., Tehran, Iran
shali@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, with encapsulated active objects which communicate asynchronously, is generally recognized to be well-suited for representing concurrent and distributed systems. In this paper we discuss the actor-based language Rebeca which is based on a formal operational interpretation of the actor model. Its Java-like syntax and object-based style of modeling makes it easy to use for software engineers, and its independent objects as units of concurrency leads to natural abstraction techniques necessary for model checking. We present a front-end tool for translating Rebeca to the languages of existing model checkers in order to model check Rebeca models. Automated modular verification and abstraction techniques are supported by the tool.

Keywords: actor model, reactive systems, model checking, modular verification, abstraction techniques.
Category: D.2.2, D.2.4

1 Introduction

Formal verification approaches are used to ensure the correctness of concurrent and distributed systems. A formal verification approach involves a behavioral model 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 [35, 18].

A behavioral model can be described by a modeling language or a programming language. Modeling languages provide a high level, abstract description language for designing software systems. As modeling languages do not need to be implemented, they allow various degrees of formality. For example, UML definitions [6] themselves do not provide a rigorous formal semantics. In practice, their formal semantics is given by various tools supporting UML [20]. On the other hand modeling languages like CSP [28], CCS [40], and I/O Automata [32] come with a formal semantics which is used for analysis, like FDR [43] which supports CSP. In general, existing modeling languages are often rather mathematical and therefore difficult to use for software engineers, or, on the contrary, rather informal, where supporting tools provide the formal semantics of at least a subset of the language.

On the other hand, programming languages are associated to an execution platform and are used for implementing and executing software systems. For example, the Bandera Tool Set [1], and the Java PathFinder [26] are used to analyze Java programs, whereas C programs can be analyzed by SLAM [4] and MAGIC [17]. But programs implementing real systems are usually too heavy and detailed, and applying formal verification approaches on the concrete level is impossible. Hence, different abstraction techniques on both data and control are needed to make the analysis process possible [22].

We also distinguish verification modeling languages, which are designed with the intention of verifying software systems, like Promela [5] and SMV [2]. As such they require a formal semantics on which the analysis process is build up. In general, these modeling languages are designed to be suitable for applying model checking techniques and are not necessarily based on a software development paradigm. They are sometimes used as target languages for abstractions which are generated from other modeling or programming languages, like in Bandera Tool Set [1], and the Java PathFinder [26].

Most of the modeling languages require an explicit mapping between different levels of abstractions: We either need to move to a higher level of abstraction to analyze an executable program, or given a verified but abstract model, we need to do some refinements to derive an executable program. Apart from the problems which are raised by the difference in abstraction level of the modeling, programming, and verification modeling languages, there are also problems in analysis processes. Two basic methods of analysis are 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. 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.

Deductive methods need a high interaction with theorem provers and hence a high expertise. Model checkers suffer from the state explosion problem, when the number of system components grows. Property preserving abstractions, and also compositional verification techniques are proposed as a solution. In abstraction techniques we mainly ignore some of the details, and for that we somehow prove that these details would not affect the properties which are of our interest. In compositional verification techniques we use the conventional divide and conquer strategy, to decompose the system into components, verify their properties, and deduce the property of the system from the properties of its constituents. There are some tools which support abstraction techniques, like Bandera [1] and the tool explained in [29], and some which support special kind of compositional verification, like Mocha [11].

In this paper we discuss an actor-based modeling language Rebeca (*Reactive Objects Language*) which has a formal foundation, presented in [48, 51]. This formal foundation provides a reference model for concurrent computation, based on an operational interpretation of the actor model [27, 7]. Reactive systems are modeled in Rebeca as a set of reactive objects (called rebec for reactive objects), which are executed concurrently. Rebecs are encapsulated objects with no shared variables and are instantiated from reactive classes. Communication between rebecs are by asynchronous message passing. This model of reactive systems makes Rebeca in particular suitable to serve as a platform for developing object-based concurrent systems in practice.

Rebeca is also supported by Rebeca Verifier tool, as a front-end, which translates Rebeca codes into languages of existing model-checkers and thus, allows to verify their properties [50, 49, 53]. Rebeca Verifier is an environment to create Rebeca models, edit them, and translate them into SMV [2] or Promela [5]. Also, the user can enter the properties to be verified at the Rebeca code level. The temporal logic supported by the tool for specifying the properties is based on the specification language of the back-end model checkers. The output code can be model checked by NuSMV [2] or Spin [5] respectively. Modular verification and abstraction techniques [11, 10, 30] are used to reduce the state space and make it possible to verify complicated reactive systems. Based on a Rebeca model, one can choose a subset of reactive objects in the model as a component. The tool then automatically generates the component model, as a Rebeca model, which can be translated to SMV and/or Promela as well. To build the component model out of components, a general environment is simulated by allowing all possible interactions.

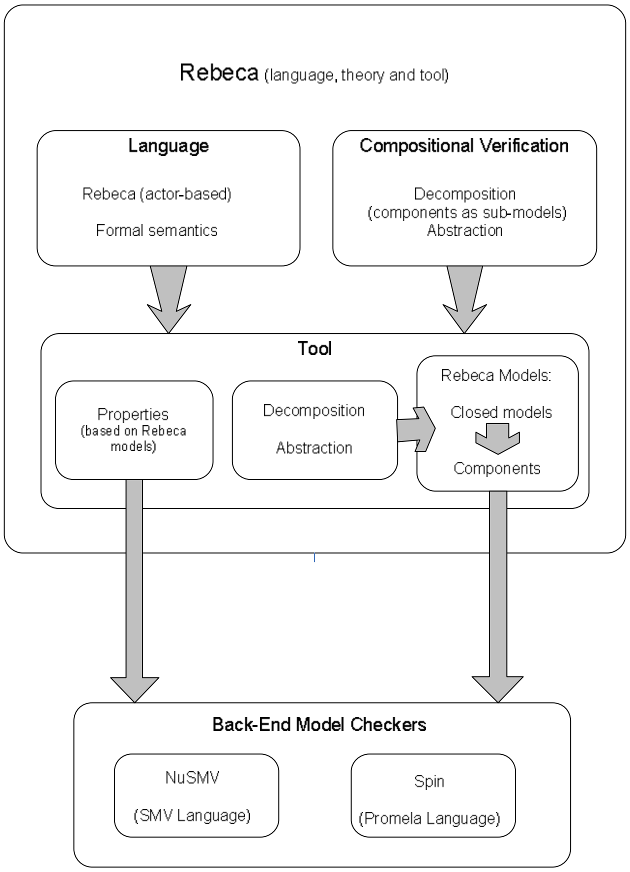

Figure 1: Rebeca: language, theory and the Rebeca Verifier tool

Figure 1 is a block diagram showing the language, verification approach, under lined theories, and tool features, together with their relationships. In this paper our focus is on the tool and our experimental results. Interesting work has been done on formalizing the actor model [9, 36, 55, 25], and there are other concurrent models supported by tools for modeling and verifying reactive systems, but to the best of our knowledge little is done about the formal verification of the actor model and automating it.

Outline of the paper. In the following section, we discuss the related work. Section 3 explains Rebeca, its syntax and semantics, and the compositional verification approach applied on it. Section 4, shows the architecture and implementation of Rebeca Verifier, the mapping algorithm to SMV and Promela and an overview of abstraction and modular verification approach. A simple example is used through the paper to explain Rebeca models, our approach in compositional verification, and using the tool. More case studies and experimental results are shown in Section 5. Section 6 is a short conclusion and an overview of our future work.

2 Related Work

The NASA's Java PathFinder [26] is a translator from a subset of Java to Promela [5]. Its purpose is to establish a framework for verification and debugging of Java programs based on model checking. The Bandera Tool Set [1] 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 including SMV, Spin, and Java PathFinder. SLAM [4] is a Microsoft's project for verification of C programs and debugging software via static analysis. 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 various abstraction techniques [4, 15]. A methodology supported by a tool MAGIC is presented in [17], to verify C programs against finite state machine specifications. The approach is compositional, trying to decompose the verification of large systems into subproblems of manageable complexity.

In [29], the implementation of a tool is described which translates SDL-specificationsto DTPromela (the discrete time extension of Promela). They assumed that the system is working in a timed chaotic environment, and they used some abstraction techniques to embed the environment into system. The abstraction techniques are similar to our method in the compositional verification approach.

RML (Reactive Module Language) [13], proposed by Alur and Henzinger, is a formal modeling language based on shared variables and synchronous computation. RML is supported by the model checker Mocha [11] and a subset of linear temporal logic is used to specify its properties. RML supports compositional design and compositional verification approach which is assume-guarantee.

Process algebras, such as CCS [38] and CSP [28], are modeling languages for communicating concurrent systems. Process-algebra terms serve as denotations of specifications and implementations alike, and the correctness is typically proved by showing that an implementation refines a specification. FDR (Failures/Divergences Refinement) is the proof and analysis tool for CSP [43]. Input-output automata for modeling asynchronous distributed systems [33, 34] are introduced by Lynch and Tuttle. They showed how to construct modular and hierarchical correctness proofs for their models.

Object-oriented modeling languages are proposed for representing reactive and concurrent systems. Actor model and POOL (Parallel Object-Oriented Language) [14] are two examples of these languages. Actor model is assumed as the first agent-based language introduced by Hewitt [27], and then developed as a functional concurrent object-based language [7, 9]. Actor model is powerful in modeling [39] and rich in theory [55, 54]. There are works done on reasoning about actor [44] and POOL models [21], but as far as we know, there is hardly any work on the tool-supported formal verification of these models.

Rebeca is a modeling language based on the powerful yet simple paradigm of actor model. Rebeca is developed to be a "verify while design language", inheriting the modeling power of actor model, and also incorporating some abstraction techniques in the semantics to make the analysis process more efficient. Rebeca is an operational interpretation of actor, with a java-like, object-based syntax which makes it easy-to-use for software engineers in modeling and also offer a simple refinement strategy. The naturally decomposable model and independent modules can be exploited in formal verification and model checking.

3 Rebeca

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. The actor model was first explained as a simple functional model [7, 8, 9], but several imperative languages have also been developed based on it [42, 58, 57].

Rebeca [48, 52] is an actor-based language, with independent reactive objects, communicating by asynchronous message passing, and using unlimited buffers for messages. 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 of messages. Each message specifies a unique method to be invoked when the message is serviced. Each rebec has an unbounded buffer, called a queue, for arriving messages.

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. When a message is read from the queue, its method is invoked and the message is removed 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. We consider the execution of a method atomic. Sending a message within a method execution is not considered to be a transition, per se. This leads us to coarse grained transitions. 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 2. The syntax of a `<reactiveclass>` definition is similar to Java, except for the definition of `<knownobjects>`. The rebecs included in the `<knownobjects>` part of a reactive class, are those rebecs to which this reactive class may send messages. Figure 3 is an example of a Rebeca model.

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>`) which are sending messages, assignments (`<assignment>`), if-statements (`<conditional>`), object creation (`<create>`), and sequential composition.

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

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

Figure 2: Reactive class, rebec and model definition syntax

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 the list of *knownobjects*. 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.

Example 1 A Rebeca model: The Bridge Controller. Figure 3 shows a simple model in Rebeca. We use this example also in other sections to explain the modeling and verification approach. Consider a bridge with a track where only one train can pass at a time. There are two trains, entering the bridge in opposite directions. A bridge controller uses red lights to prevent any possible collision of trains, and also guarantees that each train will finally pass the bridge.

Each rebec which is instantiated from the reactive class *BridgeController* has two known rebecs, *t1* and *t2*, both of type *Train*. The messages sent from this rebec are sent to *t1* and *t2*. the reactive class *BridgeController* has two state variables which show if a train is waiting from either side of the bridge, and two state variables which indicate the status of the signals. There are three message servers, one is *initial* for initializing the state variables, and the other two, *Arrive* and *Leave* are for servicing the messages come from the trains. As indicated, the sender of each message is implicitly known, the keyword *sender* in the latter message servers has the value of the sender of each message.

Known rebec of an instance of the reactive class *Train* is an instance of the reactive class *BridgeController*. The state variable *onTheBridge* of the *Train* shows the status of the train, whether it is on the bridge or not. The message server *initial* initializes the state variable, and also initiates the model by sending the *Passed* message to itself. The messages *Passed* and *ReachBridge* are sent by *Train* to itself. The message *YouMayPass* comes from the bridge controller. Three rebecs are instantiated in the main part of the model, two instances from the *Train* and one from the *BridgeController*.

More complicated examples can be found in Rebeca Home Page [3], including a similar example consisting of more trains, and also examples with dynamic creation of rebecs.

3.2 Semantics

The operational semantics of Rebeca is defined using a labeled transition system [52], 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 the semantics of Rebeca, we first formalize the definitions of a rebec, a model, and their constituents. 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 $\mathcal{V}_M = \bigcup_{i \in I_C} \mathcal{V}_i$.

```
reactiveclass BridgeController(5) {
    knownobjects{Train t1; Train t2;}

    statevars{
        boolean isWaiting1; boolean isWaiting2;
        boolean signal1;    boolean signal2;
    }
    msgsrv initial() {
        signal1 = false; isWaiting1 = false;
        signal2 = false; isWaiting2 = false;
    }
    msgsrv Arrive() {
        if (sender == t1) {
            if (signal2 == false) {
                signal1 = true;
                t1.YouMayPass();
            } else { isWaiting1 = true; }
        } else {
            if (signal1 == false) {
                signal2 = true;
                t2.YouMayPass();
            } else { isWaiting2 = true; } }
    }
    msgsrv Leave() {
        if (sender == t1) {
            signal1 = false;
            if (isWaiting2) {
                signal2 = true;
                t2.YouMayPass();
                isWaiting2 = false; }
        } else {
            signal2 = false;
            if (isWaiting1) {
                signal1 = true;
                t1.YouMayPass();
                isWaiting1 = false; } }
    }
}
```

```
  reactiveclass Train(3) {
    knownobjects{BridgeController
                          controller;}
    statevars {boolean onTheBridge;}
    msgsrv initial() {
        onTheBridge = false;
        self.Passed();
    }
    msgsrv YouMayPass() {
        onTheBridge = true;
        self.Passed();
    }
    msgsrv Passed() {
        onTheBridge = false;
        controller.Leave();
        self.ReachBridge();
    }
    msgsrv ReachBridge() {
        controller.Arrive();
    }
}
main {
    Train train1(theController);
    Train train2(theController);
    BridgeController theController
                 (train1, train2);
      }
```

Figure 3: The Bridge Controller Example

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.

In the labeled transition system $M = (S, L, T, s_0)$ which denotes the semantics of a Rebeca model, we have the followings (for a more detailed formal definition refer to [52]):

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, 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 the inboxes of the rebecs, 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 [7]: 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 2 The Bridge Controller: state transitions. The bridge controller uses its state variables to keep the value of the red lights on each side, and has flags to know whether a train is waiting on each side of the bridge or not. When the *initial* message server of a train is executed, a *Passed* message is sent to self. Serving this message causes a message *Leave* to be sent to the bridge controller and a message *ReachBridge* to be sent to self. Method *ReachBridge* sends an *Arrive* message to the bridge controller. By receiving the message *Arrive*, in the case that the light for the other side of the bridge is red, the bridge controller gives the permission to the requester to pass the bridge by sending it a *YouMayPass* message. If the light for the other side of the bridge is green, then the train cannot pass and a flag is set to indicate that the train is waiting. By receiving *YouMayPass* message, a train sends a *Passed* message to itself. By receiving a *Leave* message, the bridge controller sends a *YouMayPass* message to the other train in the case that it is waiting to pass and sets the lights properly.

3.3 Abstraction techniques and 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 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. The fundamental problem of composing specifications is to prove that a composite system satisfies its specification if all its components satisfy their specifications [31, 18, 37]. In general, compositional verification may be exploited effectively when the model is naturally decomposable, so, a model consisting of inherently independent modules is suitable for compositional verification [19].

In Rebeca, the feature of loosely coupled modules is exploited to introduce different abstraction techniques and a compositional verification approach for verifying the properties of Rebeca models. For instance abstracting the specifications from queue contents, atomic execution of methods, reducing the environment from its complete behavior to its set of sent messages, and abstracting the queue from incoming messages from environment. The background theory can be found in [48, 47].

In computer system design, we distinguish between closed and open systems. A *closed* system is a system whose behavior is completely determined by the state of the system. An *open* system is a system that interacts with its environment and whose behavior depends on this interaction. The problem of model checking open systems is different from model checking closed systems and is called module checking [30].

In compositional verification of Rebeca models, a closed model is decomposed into components. A component is a subset of rebecs of the system, and the remainder is the environment of the component. We proposed a method in modeling the environment and model checking the component in [48, 47]. The state and behavior of rebecs in a component are fully modeled, but the state and behavior of the rebecs of the environment are abstracted. The environment is modeled by sending arbitrary messages to the rebecs in the component. These messages are called external, and according to the nonpredictable behavior of the environment they are assumed to be present in all the states. In other words with respect to the environment, a component behaves like an I/O automata [34], where inputs from the environment are always enabled. So, the queues are abstracted from external messages. Using a weak simulation relation, it is proved that the safety properties specified in LTL-X (Linear Temporal Logic without next operator) [23] which are satisfied for a component are preserved for the system as well. Thus, the properties of the components are proved by model checking and are used to prove the property of the system by deduction. Choosing a component is done by the modeler and depends on the properties to be proved. There is no general approach in decomposing the system in components, components have to be selected carefully to lead to a smaller state space [31]. 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 [10].

Example 3 The Bridge Controller: abstraction and compositional verification. In the bridge controller example, we can consider a component consisting of the bridge controller and one of the trains, say *train1*. The environment is then the other train, *train2*. The external messages coming to the internal rebecs of the component, the controller in this example, are *Arrive* and *Leave*. The behavior of the controller and *train1* are fully modeled, and in each state we have always two additional enabled transitions corresponding to the external messages.

4 The Rebeca Verifier Tool

The Rebeca Verifier [50, 49, 53] provides an integrated environment to create Rebeca models and corresponding components, specify properties, and translate models and components to SMV or Promela. Using the tool, a user can create, edit and debug Rebeca codes, such that the code can be successfully translated to one of the back-end model-checker languages. The required properties can be expressed at Rebeca source code level, using temporal specification patterns based on the specification language of the back-end model checkers. These properties can also be automatically translated to the specification language of the selected back-end model checker. The output code can be model checked by NuSMV or Spin.

Modular verification is supported by the tool. The user designates the component to be verified, and then the tool automatically generates a closed model and translates it to the language of back-end model checkers. Properties should be specified based on the variables in the component. The rebecs in the rest of the model are abstracted and their state variables and message queues are not included in the generated code. Figure 4, shows the use case diagram of the system, including creating models and components, specifying properties, and translating them into SMV or Promela.

The UML component diagram of the tool is shown in Figure 5. Rebeca Verifier is written in Java and consists of components: Property handler, Component generator, and Code generators which use Property parser, Model parser and JGraph packages. We used SableCC [24] for generating the parser. SableCC produces shift-reduce parsers for LALR(1) grammars expressed in EBNF format. Parsers generated by SableCC produce abstract syntax tree (AST) of the input code. Component generator, and SMV and Promela code generators uses this AST to navigate in the Rebeca source code and build the SMV or Promela result code. The user can also specify a LTL or CTL property based on rebecs variables. The property handler, changes this property to the suitable form to be used by NuSMV or Spin.

Component generator also includes a model viewer to visualize the model using JGraph package. In the visualized model, the user can select 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 component composed by its environment makes a closed system, called a component model, which can be automatically generated by the tool.

Although the property-preserving abstraction technique is used to prevent an unbounded amount of external messages coming into the queue, but still the queue may grow unboundedly by putting messages which are sent by internal rebecs. The back-end model checkers do not support unbounded data types, so we need a limit for each rebec queue. 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.

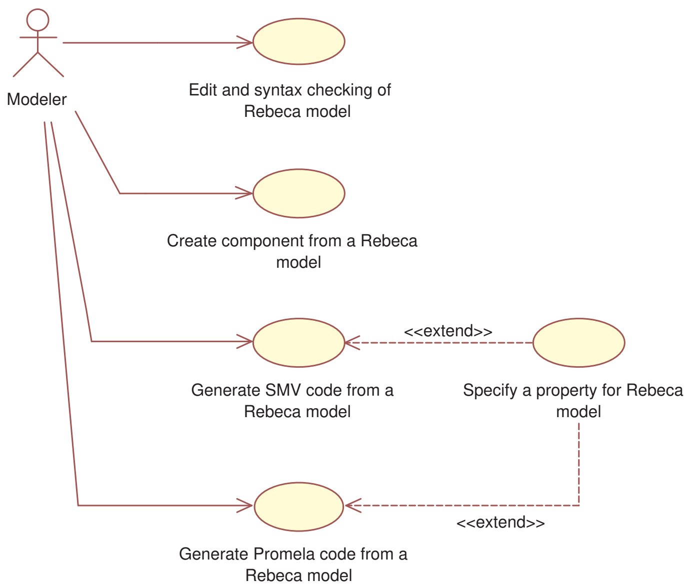

Figure 4: Use Case Diagram of Rebeca Verifier

4.1 Rebeca, SMV, and Promela

The goal in developing the tool is integrating automated verification tools in the actual practice of software engineering by means of the high-level actor-based modeling language Rebeca. We want to support automatic verification of Rebeca models as well as using and evaluating the compositional verification approach. The main difference between Rebeca and SMV or Promela is not their expressive power or efficiency in model checking. One essential difference is the actor-based modeling paradigm of Rebeca which is convenient in modeling event-driven, asynchronous systems. The other essential difference is the compositional verification approach that can be applied on Rebeca models, and increases the efficiency of formal verification. The modeling paradigm of Rebeca can be further used in designing special algorithms and increase the performance of model checking, using symmetry. This can be exploited only after implementing Rebeca direct model checker (an ongoing project) which does not use back-end model checkers. There are also other detailed differences which are explained in the following.

The behavioral model in NuSMV is described by a Kripke structure, and there is a precise mapping from Rebeca semantics in labeled transition system

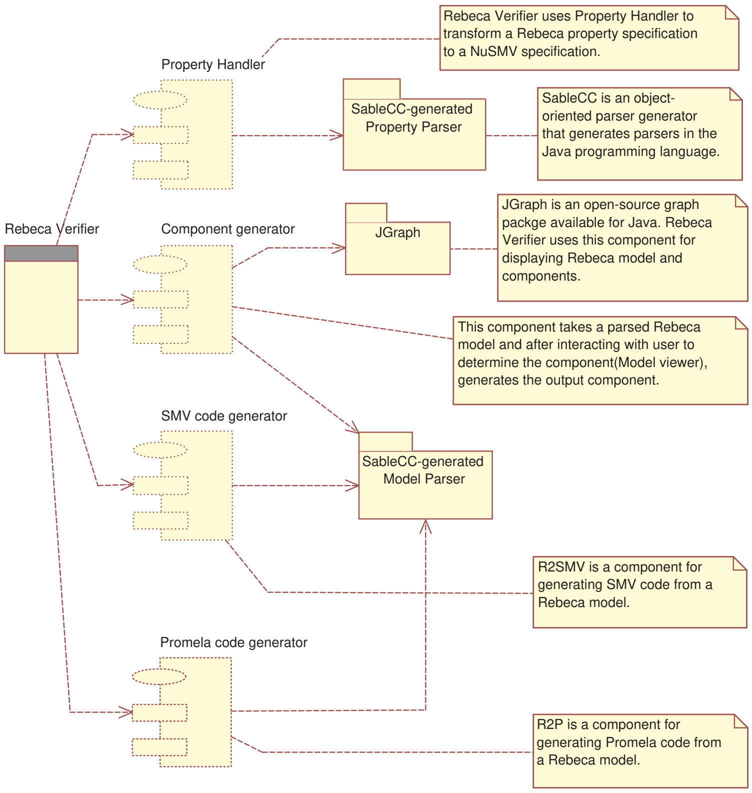

Figure 5: Component Diagram of Rebeca Verifier

into SMV Kripke structure. In compositional verification we have a nondeterministic choice between taking an internal message from the queue, or taking an external message from the set of external messages. Using Rebeca to SMV translator we are able to model check Rebeca models and also applying the compositional verification approach. But, SMV does not support structured data types, and even in using arrays only constant indexes are allowed. Another short come is lack of sequential composition in a process. Moving from simple case studies to more complicated ones we need higher modeling constructs, and supporting these features in Rebeca to SMV translator is not efficient. Although, using Rebeca to SMV translator is still efficient in modeling models with simple data components.

Promela is a rich modeling language which supports structured data types, and various communication mechanisms. All the modeling features of Rebeca are supported in Rebeca to Promela translator. The differences between Rebeca and Promela are originated from their different object-based and process-based paradigm in modeling. In Spin, the properties of a model can only be defined based on the global variables. Thus, the state variables of rebecs are mapped to global arrays in Promela.

In the following, we give a mapping between Rebeca constructs and SMV and Promela constructs. By extending these mappings to the run-time configurations of the respective languages we can establish in a straightforward manner a one-to-one correspondence between the computation steps of a Rebeca program and its translation.

4.2 Translating Rebeca to SMV

NuSMV [2] is a symbolic model checker which verifies the correctness of properties for a finite state system. The system should be modeled in the input language of NuSMV, called SMV, and the properties should be specified in CTL or LTL. The only data types in the language are finite ones, including booleans, scalars and fixed arrays. A SMV code is a set of *Module* definitions, including a *main* module. *Processes* are instantiated from *Modules*, and are used to model interleaving concurrency. The program executes a step by non-deterministically choosing a *process*, then executing all of the assignment statements in that process in one step. The main control structure in SMV is the *next-case* statement. Using this statement, the programmer can specify the next value of a variable, according to the current value of all variables in the code.

In Rebeca Verifier, the SMV code generator is used to produce SMV codes from Rebeca models [50]. The mapping from Rebeca constructs to SMV is shown in Table 1. The basis of our translation algorithm is the operational semantics of Rebeca, formalized as the labeled transition system explained in Section 3. Each class in Rebeca is translated to a module in SMV and for each rebec a process is defined. In Rebeca, concurrency is modeled by interleaved execution of rebecs which are nondeterministically chosen. This is mapped to interleaved execution of processes in SMV. Each transition in executing a Rebeca model is mapped to a process being executed in SMV model. What a process does exactly follows the Rebeca semantics: taking a message from the top of the array which denotes the queue, and executing the corresponding statements of the message server. The initial state of the SMV model is mapped on Rebeca model, by putting the init message in the message queue.

The state variables of a rebec are mapped to the local variables of the correspondent process. Each method of a rebec is executed in an atomic step in a SMV process. All the changes to a specific variable in a process, under different conditions, shall be indicated in one *next-case* statement. So, all the assignments

Rebeca construct	SMV construct
class	module
rebec	process
known objects	parameters of the process
message queue	array
message server	distributed in the code of a process
state variables of a rebec	local variables of a process

Table 1: Mapping Rebeca Constructs to SMV

to one variable in different methods of a rebec are mapped into one *next-case* statement. There is a variable in the translated SMV code which specifies the method that is currently executed. This variable is used to set up the correct condition in the *case* part of the *next-case* statement. To be able to translate a Rebeca code into SMV, we do not allow loops, and multiple assignments to the same variable in a method. A send statement is adding the corresponding message to the array in SMV.

Message queues are translated into arrays in SMV. With no variable indexes for arrays in SMV, the translated code becomes very long. In our translation procedure, we considered some optimizations to generate an efficient code in SMV with the minimum reachable states while not violating Rebeca semantics. For instance, we need to manipulate empty entries in the message queue in a way not to produce a dummy new state. Modeling the message queue as a structured variable increases the number of state variables considerably and it may cause state explosion quickly.

Instead of defining fixed length arrays for all rebecs, we let the modeler to define the length of the queue. A queue-overflow variable (corresponding to each rebec) is maintained in SMV code and can be checked as a property. Often, in our case studies, we had to increase the length of the queues to allow proper executions without the queue overflow.

4.3 Translating Rebeca to Promela

Spin [5] is a model checker that supports the design and verification of asynchronous process systems. Process interactions can be specified in Spin with rendezvous primitives, asynchronous message passing through buffered channels, shared variables, and also the combination of them.

In the Rebeca Verifier, the Promela code generator is used to produce Promela codes from Rebeca models. The mapping from Rebeca constructs to Promela is shown in Table 2. Each class in Rebeca is a proctype in Promela, and each rebec is a process. Each method of a rebec is mapped to an atomic block in

Rebeca construct	Promela construct
class	proctype
rebec	process
known objects	parameters of the process
message queue	channel
message server	atomic block
state variables of a rebec	global variables
non-deterministic assignment	if-selection
synchronous message	zero length channel

Table 2: Mapping Rebeca Constructs to Promela

the corresponding process in Promela. The message queues can easily be modeled by channels, according to the length specified by modeler. Within an infinite loop in a process, the message channel is read for the next message to be served. After receiving a message, the atomic block associated to that message will be executed. Processes (rebecs) are instantiated in the init process of Promela. Rebeca to Promela translator supports extended Rebeca which is presented in [46]. Extended Rebeca enriches Rebeca with a formal concept of components in modeling and provides an additional communication mechanism based on synchronous message-passing, for each synchronous message there is a zero-length (rendezvous) channel in Promela code.

A major difference between an object-based code and a Promela code concerns the state variables. In Spin, properties can only be specified on global variables. In Rebeca we do not have global variables, and our properties are based on state variables of rebecs. In the mapping algorithm, all state variables in Rebeca are mapped to global variables in Promela.

4.4 Creating Components and Module Checking

The compositional verification approach for Rebeca models is explained in section 3. For compositional verification we need to model check a component, which is a subset of the closed model and build an open model itself. Then, use our theory to prove the desired properties for the whole model. For model checking an open model we need to simulate the environment, this is called modular model checking or module checking [56, 30]. Simulating and abstracting the environment as a set of external messages, are done automatically by Rebeca Verifier.

To create a component, the whole model is visualized and the modeler can select a subset of rebecs in the model as 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 open component which is composed by its environment makes up a closed system, called a component model. 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. Each external rebec is modeled in the Rebeca code of the component model by indicating the messages that are sent by it. The SMV code then can be generated from the component model.

External rebecs are not modeled as processes, so all of their state variables are removed from the model. In the internal rebecs which could receive messages from outside, a fair nondeterministic choice has to be made between internal message on top of the queue, and all the external messages present. Also, the code that changes the message queues of external rebecs are removed because these are messages sent to external rebecs which are no more present.

4.5 The Bridge Controller Example

Here we use our running example to describe using the tool for module checking. In modular verification of Rebeca codes, a component is generated by decomposing a model into components. The environment is defined as a set of external messages, and external messages can be derived from provided messages of all internal rebecs of a component. As the whole system is generated first, all the possible senders of a message are known.

A component is chosen by the modeler based on the property to be proven, in a way that the overall property of the system is derivable from components properties. In this approach, we can prove the properties of the different components of a model, which may include shared rebecs, and use deduction to prove the required property of the system.

In the bridge controller example, the required properties of the system are that at any moment only one train should be on the bridge (mutual exclusion: safety), trains should finally pass the bridge (no deadlock: progress), and both trains finally pass the bridge (no starvation: progress). So the system properties are specified in LTL (Linear Temporal Logic) [23] as follows:

- Mutual exclusion: $\Box!(train1.OnTheBridge\ \&\&\ train2.OnTheBridge)$
- No deadlock: $\Box\Diamond(train1.OnTheBridge \mid\mid train2.OnTheBridge)$
- No starvation: $\Box(\Diamond(train1.OnTheBridge)\ \&\&\ \Diamond(train2.OnTheBridge))$

Here, we can decompose the model into two components, each with *BridgeController* and one of the trains in it. Because of the symmetry present in the model, it is enough to consider one of the components, model check it, and then use deduction to prove the overall property of the system out of component properties proved by model checking. Figure 6 shows a snapshot of the system, creating the required component. For the component in Figure 6, the state variables of rebec *train1* are abstracted away. So, we need to rephrase the properties according to the state variables of *BridgeController* and *train2*:

- Mutual exclusion: $\Box\ !(theController.signal1\ \&\&\ theController.signal2)$
- No deadlock: $\Box\Diamond(theController.signal1\ ||\ theController.signal2)$
- No starvation: $\Box(\Diamond(theController.signal1)\ \&\&\ \Diamond(theController.signal2))$

These rephrased properties are proved by model checking. We also prove the property:

- $\Box\ (theController.signal2 \rightarrow \Diamond(train2.OnTheBridge))$

Using the rephrased properties and the latter property, the system's properties are proved accordingly.

In the next section, the state space generated for model checking bridge controller example (and other examples) are presented and compared with the module checking the components, and the amount of state space reduction is shown.

5 Experimental Results

Rebeca Verifier is used to model check typical simple case studies as well as some medium-sized case studies (like the IEEE CSMA/CD protocol [41, 16]). We selected typical case studies from [34] and also from the case studies which are model checked by existing model checkers. For example we modeled *leader election* (both LCR and HS algorithms) [34], *the commit problem* [34], *trains and the bridge controller* [12], *dining philosophers* [28, 43, 34], *readers and writers*, and *gossiping girls*. These case studies are translated to SMV or Promela or both, and are included at Rebeca Home page [3]. The compositional verification approach is applied on some of these case studies and the state space reduction is evaluated.

Note that comparing SMV and Promela or their corresponding model checkers is not our goal. The goal is to examine the expressive power of Rebeca in modeling typical cases of different computing paradigms in modeling distributed and concurrent systems; and evaluating the compositional verification approach and find the patterns on which this approach works efficiently; and also investigate and extend the tool capabilities. Our experimental results, shown in the following tables, express the benefits of our compositional approach for the considered applications. Comparing NuSMV and Spin is one of our future works.

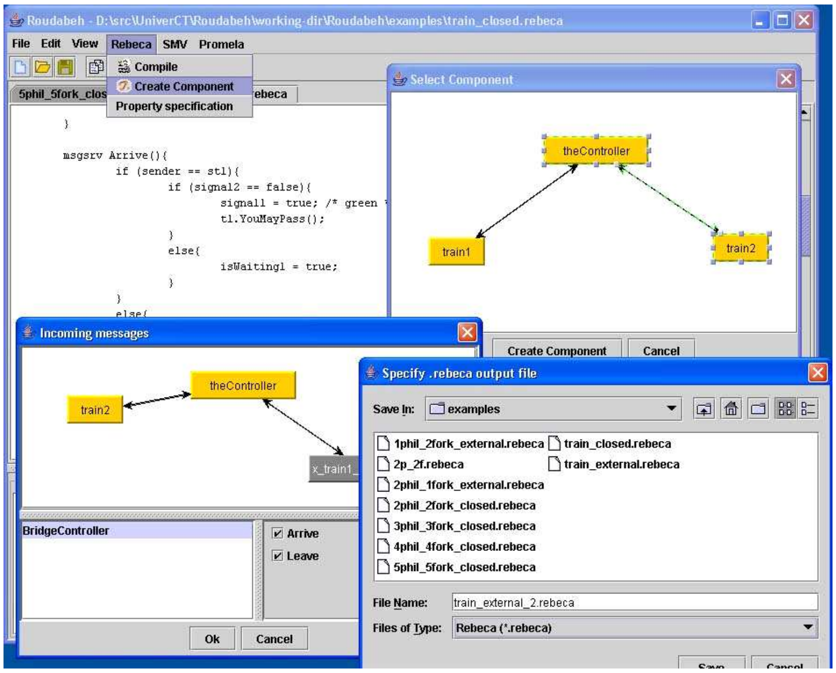

Figure 6: A Snapshot of the Tool, Creating a Component from Bridge Controller Example.

In the following we shortly explain a number of case studies for which compositional verification approach is applied and state space reduction is gained. Module checking by Rebeca Verifier is currently supported by SMV code generator, the model checking process is done by NuSMV 2.1.2, executed on Windows XP professional, CPU: Athlon XP 1700+, with 512 MB RAM.

Safety, deadlock and starvation properties are first checked for the close model. In all the examples, there were bugs in our code which were found by model checking. Some of the bugs simply were in initializing variables and some were more serious ones, in communication and synchronization between rebecs. The CPU time and memory used by SMV for computing total and reachable states are shown for each case study. Also, the components that are selected and model checked are given. These results show that how modeling the components instead of the whole system can help in reducing the reachable states. Rebeca code and the properties that are checked for each case study can be found at Rebeca Home page.

Trains and the bridge controller. This is our running example shown in Figure 3.

Model checking results are summarized in table 3. It can be seen that total state space is reduced in the order of 10^4, but the number of reachable states is slightly increased. Number of rebecs present in the component is less than the rebecs present in the close model, and so the number of state variables are less in the component. The number of reachable states is increased because of the external messages that are always present in a component model, but are not really sent in the close model.

We checked the queue-overflow condition and found out that queue length of two for the trains and four for the bridge controller is enough for preventing overflow.

Approach	Model	Reachable states	Total states	CPU time (mm:ss)	Memory (KByte)
Closed-world	2 Trains/Controller	203	5.16e+13	00:00	8956
Component-based	1 Train/Controller (an ext. Train)	231	2.38e+09	00:00	8612

Table 3: Trains and the Controller: Closed-World Compared to Component-Based Approach (results generated by NuSMV)

Dining philosophers. We modeled the dining philosophers example as a case study and translated it into SMV using the tool. 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.

In the close system, there are eight rebecs, four philosophers and four forks. The component includes two philosophers and one fork, so we have three internal rebecs, and only two external ones. Other rebecs do not send any messages to internal rebecs of the specified component. These two external rebecs are two forks adjacent to the internal philosophers. The reduction in state space is significant in this example and is shown in Table 4. Only in the close model with two philosophers and two forks, the reachable states are less than reachable states of the component. This is again caused by external messages which made the enabled transitions more than the real enabled transitions in a close world. But total states are less because of the reduction in number of variables.

This case study can be considered as a prototypical example of a general problem consisting of a set of reactive objects arranged in a ring-shape topology;

representing a resource allocation problem involving allocation of pairwise shared resources in this ring of objects. The model in Rebeca is scalable without any changes in the code of philosophers or forks, as the links between rebecs do not change by increasing the number of rebecs (see figure 7 which is a snapshot of the tool creating a component consisting of two philosophers and one fork). Thus, the properties which are satisfied for the component preserves for the model consisting of any number of rebecs.

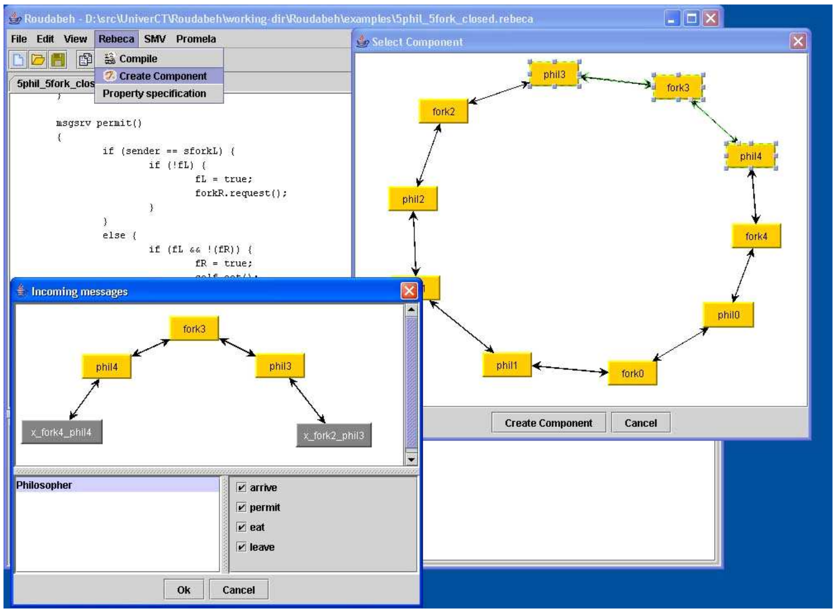

Figure 7: A Snapshot of the Tool, Creating a Component from Dining Philosophers Example.

Readers and writers This is the typical example of a data buffer that multiple readers can read from it, but only one writer can write into it. Here, we need a message queue of length two for both readers and writers, and four for the data buffer. This case study can be considered as a prototypical example of a problem consisting of a critical section and requesters arranged in a star-like topology around it.

Approach	Model	Reachable states	Total states	CPU time (mm:ss)	Memory (KByte)
Closed-world	2 Phils/2 Forks	285	3.28e+22	00:00	11136
	3 Phils/3 Forks	14671	8.79e+36	00:12	19304
	4 Phils/4 Forks	390720	1.80e+52	06:28	38700
Component-based	2 Phils/1 Fork (2 External Forks)	4132	1.16e+21	00:02	14076

Table 4: Dining Philosophers: Closed-World Compared to Component-Based Approach (results generated by NuSMV)

Approach	Model	Reachable states	Total states	CPU time (mm:ss)	Memory (KByte)
Closed-world	3 Readers/1 Writer	3293	2.60e+23	00:02	18288
Component-based	Data Buffer (external R/W)	180	1.81e+09	00:00	8664

Table 5: Readers and Writer: Closed-World Compared to Component-Based Approach (results generated by NuSMV)

6 Conclusion and Future Work

In this paper we have shown how tools for model checking and compositional verification can be integrated in the actual practice of software engineering by means of the actor-based modelling language Rebeca. We generate a front-end tool, Rebeca Verifier, for translating Rebeca models to SMV or Promela. Our tool supports modular verification, enabling the modeler to model check components derived from decomposing Rebeca models. This is used in our compositional verification approach. Abstraction techniques are applied to overcome state explosion problem.

Rebeca group at Tehran and Sharif universities has already worked on several prototypical case studies including medium-sized case studies like IEEE CSMA/CD protocol. This protocol is used in multiple access shared media environments, which use a shared bus for connecting a number of independent computers. More information can be obtained from Rebeca Home Page [3].

Modeling and verifying security protocols using Rebeca is an ongoing project, for example Mitnick attack is modeled in Rebeca to show how an attacker may chain simple attacks to construct a complex distributed attack. A server, a client, their TCP agents and an attacker are modelled. According to the asynchrony of computer networks, they can be naturally modeled by Rebeca. *Availability* is one of the required properties, which is shown to be violated by syn-flood attack.

The Mitnick attack is also simulated to show that an unsafe command can be executed on the server.

Modular structure of Rebeca allows for an incremental development of the tool. We started with Rebeca kernel, as a pure actor-based language, which describes a set of rebecs in a flat structure, communicating by asynchronous message passing. SMV and Promela code generators are both implemented for this kernel language. Promela code generator also supports synchronous message passing which is added to Rebeca as an extension to support globally asynchronous and locally synchronous systems [46]. The Promela code generator, will soon support the dynamic features of Rebeca, including dynamic rebec creation and dynamic changing topology. Direct model checking of Rebeca models is an ongoing project. Without using back-end model checkers we can exploit Rebeca modularity more efficiently in model checking algorithms and introduce other abstraction techniques.

Acknowledgement

This research is financially supported by the grant number 22210001/1/01 of Deputy of Research Office of Tehran University.

References

1. Bandera. http://www.cis.ksu.edu/santos/bandera.
2. NuSMV user manual. availabe through http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html.
3. Rebeca. http://khorshid.ut.ac.ir/~rebeca.
4. SLAM. http://research.microsoft.com/slam.
5. Spin user manual. available through http://netlib.bell-labs.com/netlib/spin/whatisspin.html.
6. UML: Unified Modeling Language Specification version 1.4, Sept. 2001, OMG document formal/01-09-67. Availabe through http://www.omg.org/technology/documents/formal/uml.htm.
7. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1990.
8. G. Agha. The structure and semantics of actor languages. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, Berlin, Germany, 1990.
9. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
10. R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automating modular verification. In *CONCUR: 10th International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 82–97. Springer-Verlag, Berlin, Germany, 1999.
11. 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.
12. R. Alur and T.A. Henzinger. Computer aided verification. Technical Report Draft, 1999.

13. R. Alur and T.A. Henzinger. Reactive Modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.
14. P. America, J. de Bakker, J. N. Kok, J. J. M. M. Rutten. Denotational Semantics of a Parallel Object-Oriented Language. In *Information and Computation*, 83(2): 152–205, 1989.
15. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *MSR-TR-2004-8. Invited talk/paper for Integrated Formal Methods 2004*, 2004.
16. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for BDD-based verification of real-time systems. In Hunt W.A., Jr. Somenzi, and F. Somenzi, editors, *Proceedings of CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 122–125. Springer-Verlag, Berlin, Germany, 2003.
17. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 385–395, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.
18. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
19. W. P. de Roever, h. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1998.
20. W. Damm, B. Josko, A. Pnueli and A. Votintseva, Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, Springer-Verlag, Berlin, 2003.
21. F.S. de Boer, 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.
22. M.B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C.S. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23nd International Conference on Software Engineering*, pages 177–187, 2001.
23. E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.
24. E. Gagnon and L. Hendren. SableCC – an object-oriented compiler framework. In *Proceedings of TOOLS 1998*, pages 140–154. Springer-Verlag, Berlin, 1998.
25. 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.
26. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
27. 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.
28. C. A. R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.
29. 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.
30. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.

31. L. Lamport. Composition: A way to make proofs harder. In *Proceedings of COMPOS: International Symposium on Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–407. Springer-Verlag, Berlin, Germany, 1997.
32. N. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
33. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT, 1987.
34. N.A Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CS, 1996.
35. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems (Safety).* Springer-Verlag, Berlin, Germany, 1995.
36. I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220(2):409–467, June 1999.
37. K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, May 2000.
38. R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.
39. R. Milner. Elements of interaction: Turing award lecture. *Communications of the ACM (CACM)*, 36(1):78–89, Jan 1993.
40. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.
41. J. Parrow. Verifying a CSMA/CD-protocol with CCS. In *Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification*, pages 373–387, Atlantic City, New Jersey, 1988. North-Holland.
42. S. Ren and G. Agha. RTsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Notices*, 30(11):50–59, November 1995.
43. W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.
44. S. Schacht. Formal reasoning about actor programs using temporal logic. *Proceedings of Concurrent Object-Oriented Programming and Petri Nets*, LNCS 2001, pages 445-460, Springer-Verlag, Berlin, 2001.
45. I. Schinz, T. Toben, Ch. Mrugalla and B. Westphal, The Rhapsody UML Verification Environment. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, IEEE September 2004.
46. M. Sirjani , F. de Boer, A. Movaghar and A. Shali. 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)*, to appear. IEEE Computer Society, 2005.
47. M. Sirjani and A. Movaghar. Simulation in Rebeca. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 923–926. CSREA Press, USA, 2002.
48. M. Sirjani and A. Movaghar. An actor-based model for formal modelling of reactive systems: Rebeca. Technical Report CS-TR-80-01, Tehran, Iran, 2001.
49. M. Sirjani, A. Movaghar, H. Iravanchi, M. Jaghoori, and A. Shali. Model checking in Rebeca. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 1819–1822. CSREA Press, USA, 2003, June 2003.
50. 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.
51. M. Sirjani, A. Movaghar, and M.R. Mousavi. Compositional verification of an object-based reactive system. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01)*, pages 114–118, Oxford, UK, April 2001.
52. 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.

53. 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.
54. C. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, December 1998.
55. C. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, August 2002.
56. M. Y. Vardi. Verification of open systems. *Lecture Notes in Computer Science*, 1346:250–267, 1997.
57. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
58. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* Series in Computer Systems. MIT Press, 1990.