Scientia Iranica, Vol. 12, No. 1, pp 55–65
© Sharif University of Technology, January 2005

SCIENTIA
IRANICA

Integrating Model Checking and Deduction for Rebeca

M. Sirjani*and A. Movaghar[1]

Rebeca is an actor-based language for modeling concurrent and distributed systems. Its Java-like syntax makes it easy-to-use for practitioners and its formal foundation is a basis to make different formal verification approaches applicable. Compositional verification and abstraction techniques are used in formal verification of Rebeca models to overcome state explosion problems. The main contribution of this paper is to show how model checking and deduction are integrated for verifying certain properties of these models. Deduction is used to prove that abstraction techniques preserve a set of behavioral specifications in temporal logic and is also used in applying the compositional verification approach, on the basis of the model checked components.

INTRODUCTION

Reactive systems are increasingly used in applications where failure is unacceptable. 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 [1-4].

Object-oriented modeling is an appropriate approach for representing reactive systems, which usually exhibit concurrency and are distributed. The actor model [5,6] is a better candidate than passive object-oriented programming languages, because of its promotion of independent computing entities to support migration, distribution, dynamic reconfiguration, openness and efficient parallel execution.

Two basic approaches for verifying properties are model checking and deductive methods. A combination of these two approaches can be exploited in a number of ways, for example, in abstraction and compositional verification.

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 the authors' knowledge, little has been done on verifying actor languages (related work is discussed in the following section). In this paper, an actor-based model, called Rebeca (Reactive Objects Language) has been developed for describing reactive systems. Methods for specifying properties and verifying their correctness are presented. A major obstacle to the use of automatic verification methods is the problem of state explosion. Specially, in the authors' model, because of the encapsulated constructs used, there may be a very large state space in model checking, even for simple systems. Modularity and different kinds of abstraction techniques are used to overcome the state-explosion problem. The main contribution of this paper is to show how model checking and deduction are integrated to apply some abstraction techniques and a compositional verification approach, well-suited to the presented actor-based model.

Modeling a system in Rebeca requires one to specify reactive-object templates and a finite set of object instances that run in parallel. In Rebeca, actor-based concepts are used for the specification of reactive systems and their communications; components are introduced as an additional structure for verification purposes; and formal semantics are provided for the model and components being comprised of their states, communications, state transitions and the knowledge of accessible interfaces. For formal verification of Rebeca models, different abstraction techniques are used, which preserve a set of behavioral specifications in temporal logic and reduce the state space of a model, making it more suitable for model checking techniques. Deduction is used to establish the soundness of these abstraction techniques by proving certain relations between the constructs; model checking and

*. *Corresponding Author, Department of Computer Engineering, Sharif University of Technology, Tehran, I.R. Iran.*

1. *Department of Computer Engineering, Sharif University of Technology, Tehran, I.R. Iran.*

deduction are integrated in applying a compositional verification approach to model check properties of the components of a system and to deduce global properties from these local properties. A tool is provided for translating models into target languages of existing model checkers, NuSMV [7] and Spin [8], enabling model checking of open, distributed systems.

The language used, 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, there are classes, from which active objects are instantiated. Classes serve as templates for state, behavior, and the interface access; adding reusability in both the modeling and verification processes.

Rebeca and the above features are explained in the following sections.

Outline of the Paper

In the following section, the work that is related to the authors' approach of modeling and analyzing open systems is reviewed. Then, the modeling language, Rebeca, and its syntax and formal semantics for closed Rebeca models are presented. After that, open Rebeca models, called components, are introduced. In the section of Integrating Model Checking and Deduction, compositional verification and abstraction techniques used in the authors' approach are explained. Following this, weak simulation is defined as an abstraction technique applied to Rebeca components, used in the compositional verification of the model. Furthermore, a simple example is used to show how reactive systems can be modeled in Rebeca. A formal verification approach is applied to this example and the results gained by using the authors' tool are shown; the tool works by automatic translation of abstracted Rebeca models into SMV [7]. The final section shows the direction of future work.

RELATED WORK

Object-oriented models for concurrent systems have been widely proposed since the 1980s. The actor model was originally introduced by Hewitt [9] as an agent-based language. It was later developed by Agha [5,6] 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 [6,10,11].

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

As far as is known, little has been done on the formal verification of actors [12,13]. In Rebeca, an imperative view of actors is proposed, trying to have a more practical model, as well as a firm theoretical basis.

The integration of model checking and deduction has been used in different ways in the analysis of models of concurrency [14,15]. Abstraction is a methodology that combines deductive and algorithmic techniques. Abstraction can be used to reduce problems to model-checkable form, where deductive tools are used to construct valid abstract descriptions or to justify that a given abstraction is valid. Another approach is compositional verification in which deduction is used to prove global system properties by composing local system properties (with a smaller state space) that have been proved using model checking.

Kesten and Pnueli mentioned modularization and abstraction as the keys to practical formal verification, using a fair Kripke structure as the computational model for reactive systems and temporal logic as a requirement specification language [16]. Rajan, Shankar and Srivas [17] illustrated an application, where model checking is applied to a finite state abstraction of a system, where abstraction is justified by means of theorem proving. Saidi and Shankar presented a general abstraction/refinement algorithm that preserves the full $\mu-$calculus as the basis for an integration of abstract interpretation, model checking and proof checking [18]. A formal framework for modular description and verification of fair transition systems is presented and several deductive proof techniques to establish and re-use modular properties are proposed by Finkbeiner and Manna [19].

Clarke, Long and McMillan used interface processes to model the environment for a component in their compositional verification approach [20]. They modeled systems as finite transition systems and used CTL (Computation Tree Logic) to specify their properties. Input-output automata for modeling asynchronous distributed systems are introduced by Lynch and Tuttle [21,22]. They showed how to construct modular and hierarchical correctness proofs for their models. Alur and Henzinger proposed RML (Reactive Modules Language) for modeling a system and used a subset of linear temporal logic and alternating-time temporal logic, to specify its properties [23]. RML supports compositional design and verification.

All of these methods can be viewed as formal verification methods, consisting of a model for describing the behavior of the system, a property specification language and a method to verify the correctness of properties. Rebeca can be considered in the same category, using an actor-based model for describing the behavior of the system, temporal logic for specifying properties and a method, using deduction and model checking, for verifying the correctness of properties. The complex semantics of Rebeca cause the state explosion problem to be more severe in model checking, but its actor-based nature leads to straightforward techniques for modularization and abstraction. The contribution is integrating model checking and deduction and using the inherent decoupling of the modules in Rebeca for overcoming the state space explosion problem, thus, providing an actor-based language suitable for modeling concurrent and distributed systems and familiar for practitioners on which formal verification can be applied.

Many models, including those mentioned above, have tools for facilitating their analysis. For example, Mocha is the model checker for RML [24]. Two of the most widely known tools for model checking are SMV [7] and Spin [8]. The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. Spin supports the LTL model checking of distributed systems. Spin uses a high level language, called PROMELA (PROcess MEta LAnguage), to specify system descriptions.

A tool has, also, been developed for translating Rebeca to SMV. It enables one to model check Rebeca models, both in closed and open forms. This tool is used to show that the compositional verification approach reduces the state space in many cases [25].

PROPOSED MODEL: REBECA

The model proposed here [26] 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. Class declarations are added to the syntax, which act like templates for states, behavior and interfaces of active objects. Also, there is the notion of a component as a set of concurrently executing active objects.

The authors' objects are reactive and self-contained. Each of them are called 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 it 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 method is invoked and the message is deleted from the queue. One may refer to the messages as "method invocation requests".

Each rebec is instantiated from a class and has a single thread of execution. A model, representing a set of rebecs, is defined as a closed system. It is composed of rebecs, which execute concurrently and interact with each other. Components are introduced as open systems, consisting of subsets of rebecs in a model.

The execution of a method is triggered by removing its "method invocation request" from the top of the queue and results in an atomic execution of its body, which cannot be interleaved by any other rebec. Note that this coarse granularity of the interleaving of rebecs 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 state space and makes the model simpler.

Syntax

The syntax for classes (reactive-object templates), rebecs (class instantiations) and models (parallel composition of rebecs), is presented in Figure 1 (a simple example is shown in Figure 2 and is explained further). The syntax of a <class> definition is similar to the one in Java, except for the syntactic entity <interfaces> that precedes the body of every class definition. In <methodssig> of the interface, a rebec specifies what kind of services it offers to the world. Only methods from this interface are intended to be possibly known to other rebecs.

The <body> first lists its fields (in <statevar>) and, then, declares its local methods, which may, themselves, contain local variables (in <method>). Variables are typed and method declarations follow a standard syntax. Unlike in Java, methods have no call-back mechanism and, therefore, no return type. The core language for statements (<statement>) allows remote method invocation requests (<mir>), assignments (<assignment>), if-statements (<conditional>), while-statements

```
<classes>       ::=  {<class> }*
<class>         ::= class<classname>( {<var>}*):
                      { <interface>
                        <body> }
<interface>     ::= interface :
                        [<methodssig>]
<methodssig>    ::=  {<methodid> ({<var> }*);}*
<body>          ::= body :
                     <statevar>
                      {<method>}*
<statevar>      ::=  {<var>}*
<var>           ::= <varid>:<vartype>;
<method>        ::= <methodid> (  {<var> }*)
                       { {<var>*}
                        {<statement>; }* }
<statement>     ::= <mir>      | <assignmnt> |
                     <conditiontal> | <iteration> | <create>
<mir>           ::= send ( <varid>, <methodid> ({<varid> }* ))
<rebecs>        ::= rebecs: {<rebecid>:(<rebecid>*)}*
<model>         ::= model=||({<rebecid>}*)
```

Figure 1. Class, rebec and model definition syntax.

```
class Philosopher:(Forkl,Forkr:Fork) {
  interface:
    Permit();

  body:
    boolean eating;
    boolean FL, FR;
    Arrive()
    {
      send (Forkl, Request());
      send (Forkr, Request());
      send (self, Eat());
    }
    Permit()
    {
     if (sender == Forkl)
        FL = true;
      else
        FR = true;
    }
   Eat()
    {
      if (FL && FR)
      {
        eating=true;
        send (self, Leave());
      }
      else
        send (self, Eat());
    }
    Leave()
    {
     FL = false;
      FR = false;
      eating = false;
      send (Forkl,Release());
      send (Forkr,Release());
      send (self, Arrive());
    }
    init()
    {
     FL = false;
      FR = false;
      eating = false;
      send (self, Arrive());
    }
  }
```

```
class Fork:(Phill,Philr:Philosopher) {
  interface:
    Request();
    Release();
  body:
    boolean busy;
    boolean requester;
    Request() {
      if (sender <> self)
        if (sender == Phill) requester = true;
        else requester = false;
     if (busy) send(self,Request());
      else {
      busy = true;
      if (requester) send (Phill,Permit());
       else send (Philr,Permit());
      }
    }

   Release()
    {busy = false; }

    init()
    { busy = false; }
  }
  rebecs:
   Phils0:Philosopher(Forks0,Forks1);

    Phils1:Philosopher(Forks1,Forks2);

    Phils2:Philosopher(Forks2,Forks3);

   Phils3:Philosopher(Forks3,Forks0);

   Forks0:Fork(Phil0,Phil3);

   Forks1:Fork(Phil1,Phil0);

   Forks2:Fork(Phil2,Phil1);

    Forks3:Fork(Phil3,Phil2);


  model = || ( Phils0, Phils1, Phils2, Phils3,
            Forks0, Forks1, Forks2, Forks3);
```

Figure 2. Dining philosophers system.

(<iteration>), object creation (<create>) and sequential composition.

In <mir>, 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 class definition has, at least, one main method, named init, which is the first method executed by each rebec. In Rebeca, a <model> is a finite collection of rebecs that are (created and then) run in parallel. In declaring a rebec, the bindings to its known rebecs are specified in its parameter list.

State variables of each rebec are declared in its body. Idle waiting can be modeled by sending messages to self. All the rebecs in the model are instantiated from classes and their bindings to known rebecs are specified when instantiated. The model is a parallel composition of declared rebecs.

Operational Semantics for Closed Models

Figure 1 shows that a Rebeca <model> is a finite collection of rebecs $r_1, r_2, \cdots, r_n$, running in parallel $||r_i$. A model is called closed, if all <mir> requests within a <model> are addressed to and originate from rebecs within that model. Otherwise, a model is open and is called a component. Components are discussed further in the following section.

A closed <model> determines a labelled transition system, $M = (S, L, T, s_0)$, with state set, S, signature of action labels, (L), transition relation, $(T \subseteq S \times L \times S)$, and initial state, $s_0 \in S$:

- The state space of the model, S, is the set:

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

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, is an unbounded buffer that stores all incoming method, invocation requests (<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 method invocation requests that are part of the target rebec behavior;
- 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 $i (1 \le i \le n)$, 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 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 set in Formula 1 to the following set:

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

    where i_{new} ranges over the new rebecs created within that method invocation and s' is an element in the set represented in Formula 2;
- The initial state, s_0, is the one where each rebec has executed the declarations of all its fields and its init method is the sole element in its inbox.

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 a code, <mir> requests may be issued and rebecs may be created. In the authors' semantics, method invocation requests (<mir>) are the sole mechanism for communication between these rebecs. Regarding the infinite behavior of the semantics, communication is assumed to be fair [5]: All <mir> requests eventually reach their respective inboxes and will, eventually, be invoked by the corresponding rebec.

COMPONENTS IN REBECA

In Rebeca, for verification purposes, one may decompose a closed model and think of one part as the open system and the remainder as the environment that makes the overall system closed. This decomposition determines which rebecs in the model have to be modeled with state and behavior and which rebecs may be abstracted such that they only send messages.

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

This decomposition process abstracts the model considerably: Only internal rebecs are fully modeled; external rebecs are only modeled in their capacity to request remote method invocations. So, they are only modeled as the set of external messages that can be sent by them. This set of external messages represents the environment for the component. Instead of putting external messages in an internal inbox, they may be processed at any time, up to fair interleaving with the processing of requests in the inbox. This makes the model more understandable and 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 [22], where inputs from the environment are always enabled.

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.

With the decomposition technique, the universe of rebecs is always known. The active classes in the closed system designate 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, one is able to compose them into another component. The resultant 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.

INTEGRATING MODEL CHECKING AND DEDUCTION

In formal verification, one tries to prove or disprove that a model satisfies some specifications. There are two basic approaches of 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).

Model checking and deduction both have strengths and weaknesses. Model checking can be done automatically, but is limited by the state explosion problem. Deductive approaches, using theorem proving, require a considerable amount of manual guidance and high expertise. By integrating these two technologies, one can take advantage of both. Abstraction can be used to reduce state space and transform a problem to a model-checkable form. Deduction can be used to justify that a given abstraction preserves a set of properties. In compositional verification, these two methods can be combined in such a way that desirable features of each are retained, while minimizing their shortcomings. In compositional verification, the goal is to check the properties of the components of a system and deduce the global properties from these local properties. The main difficulty with this approach is that local properties are often not preserved at the global level.

A property-preserving abstraction of a model is another model whose properties can be mapped back to the first one. If a model, A, is a property-preserving abstraction of a model, M, which preserves a certain set of properties, including ϕ, and, if the property, ϕ, holds for A, i.e., $A \models \phi$, then, one can conclude that ϕ holds for M, i.e. $M \models \phi$. If it is proven that ϕ holds for A by model checking and, if A can be automatically constructed from M, then, one has a powerful verification method [16-18]. In compositional verification, the specification of a system is decomposed into the properties of its components, which are then verified separately. If one deduces that the system satisfies each local property and show that the conjunction of the local properties implies the overall specification, then, one can conclude that the system satisfies this specification too. There has been a strong trend to use compositional approaches in formal verification of systems [12,27-29].

In general, compositional verification may be exploited more effectively when the model is naturally decomposable [30,31]. In particular, a model consisting of inherently independent modules is suitable for compositional verification. The actor-based model provides such independent modules because of the asynchronous communication mechanism, which involves only an explicit non-blocking send operation.

Weak Simulation

The state explosion problem may be avoided by using techniques that replace a large component by a smaller component, which satisfies the same properties. In general, a notion of equivalence or preorder among structures is needed, guaranteeing that two components satisfy the same set of formulas in a given logic, or that certain properties are preserved. The set of properties which are preserved depends on the notion of defined equivalence or preorder.

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 may 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 may have behaviors that are not possible in the original component. Because of the over-approximation of the transition relation, abstractions usually preserve only universally quantified path temporal formulas in logics, such as LTL or $\forall CTL$ formulas.

Now, the weak simulation relation among components in this model is explained. For the sake of simplicity, here, dynamic creation and dynamic topology are ignored both in the closed model and the components. Therefore, referring to the operational semantics of the models in the previous section, the state space, S, does not expand dynamically from Formula 1 to Formula 2.

External messages coming into the component are present in all the states and one can imagine that they are like the members of a set that is constantly attached to all the states in the corresponding labelled transition system. So, in each state, there are 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, one does not need to consider it in each state.

To define the weak simulation relation between two components, the operational semantics definition, the component definition and the following notations are used. 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. Each variable has one valuation in each state. For a state s_C, $s_C.\mathcal{V}_C$ denotes the set of these valuations for each one of the variables in that state. The inbox of component C is defined as a multi-queue, where 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 previously, 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.

A projection relation between two states is also defined. State $s_{C'}$ is a projection of state s_C (denoted by $s_{C'} \upharpoonright s_C$), 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, the weak simulation relation is now defined.

Definition 1 (Weak Simulation)

Given two components C and C' of a given model, represented by labelled 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 its 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 $s_{C'_1} = s_{C'}$ or one has $(s_{C'}, l, s_{C'_1}) \in T_{C'}$, where $H(s_{C_1}, s_{C'_1})$. (Note that one has $s_{C'_1} = s_{C'}$, if $l \notin L_{C'}$, which is stuttering; and $(s_{C'}, l, s_{C'_1}) \in T_{C'}$, if $l \in L_{C'}$.)
2. The authors believe 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, a theory is introduced, which provides a formal justification for the 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

For any two components C' and X of a model C (defined on the same universal set of rebecs), C' weakly simulates $C = C'||X$.

Proof

Consider $H = \{(s_C, s_{C'}) \in S_C \times S_{C'} | s_{C'} \uparrow s_C\}$. It requires to be shown: (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) $s_{C'} \uparrow s_C$, by 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 one still has $H(s_{C_1}, s_{C'_1})$. 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. 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 $r \in I_{C'}$, where $l = (p, r, m)$. One has to show that l is enabled in $s_{C'}$ and, then also show that $s_{C'_1} \uparrow s_{C_1}$. First, it is shown that l is enabled in $s_{C'}$, in all possible conditions:
         - l is external for both C and C'. It is known 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'. It is known that $H(s_C, s_{C'})$, so $s_{C'} \uparrow s_C$ and, also, $q_{C'} \uparrow q_C$. From the definition of projection, it is known that, if l is on the top of the queue in s_C, it has to be on top of the queue for $s_{C'}$ too. Thus, l is enabled in $s_{C'}$.

         Second, it is proven 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);
         - 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 $s_{C'_1}.q_{C'}$ 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, it is shown 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, one says that $C \models \phi$, iff ϕ holds for all computations of C.

There is the following theorem, which restricts the corresponding theorem of Clark et al. [3] to safety properties.

Theorem 2

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

Using this theorem, one has the following corollary for compositional verification of LTL-X properties. $R = ||_{i=1}^{n} X_i$ is the parallel composition of n components, $X_i, i = 1, \cdots, n$ and one has $I_R = \bigcup_{i=1}^{n} I_{X_i}$.

Corollary 1

Let $R = ||_{i=1}^{n} X_i$ and φ_{X_i} be a property of X_i specified in LTL-X, e.g., $X_i \models \varphi_{X_i}$. In order to show that φ_R is a property of system R, e.g. $R \models \varphi_R$, it suffices to find properties for each X_i, such that:

1. For $i = 1, \cdots, n, \varphi_{X_i}$ is a property of X_i, e.g., $X_i \models \varphi_{X_i}$,
2. $(\bigwedge_{i=1}^{n} \varphi_{X_i}) \Rightarrow \varphi_R$ is valid.

One can prove for $i = 1, \cdots, n, X_i \models \varphi_{X_i}$ by model checking. After that, if $(\bigwedge_{i=1}^{n} \varphi_{X_i}) \Rightarrow \varphi_R$, then, $R \models \varphi_R$. In each system, deduction shall be used to prove that this formula is valid. Automated theorem provers can be used for proving this formula.

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, one needs to share some rebecs between some components. Theorem 2 holds in this situation too. Hence, one can use this corollary.

Sometimes, a system consists of similar components in which one can use a kind of generalization. It is said that two components are similar when they consist of the same number of rebecs and, for each rebec in one, there is one, and only one, corresponding rebec in the other component and both rebecs are instantiated from the same class. Hence, there are also similar sending/receiving connections between rebecs in similar components. Since all instances of a class have similar properties, so have all similar components. This is due to the existing symmetry in the model. A component is a permutation of its similar components over the rebec identifiers set and so their semantics are equivalent. The modeler chooses a component whose 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.

CASE STUDY

Rebeca is used to model the dining philosophers example. This system is discussed in various texts [16,32,33] and can serve as a simple example for showing how to use the proposed method.

A Rebeca Model

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. Figure 2 shows a solution for the dining philosophers problem, with $n = 4$, coded in Rebeca.

The system consists of a Philosopher class that is a template for defining philosophers and a Fork class that is a template for forks (see Figure 2). This model consists of four philosophers and four forks. The known rebecs of each philosopher are its left and right forks and known rebecs of each fork are its left and right philosophers.

Some State Transitions

Here, some of the state transitions in the example and how the execution of method servers causes state transitions are explained:

- In the dining philosophers example, in the initial state, there are four philosophers and four forks with their init methods in their inboxes. So, we have eight enabled transitions. Execution of the init methods may cause sending messages to others or to self and/or setting field variables;
- After execution of the init method of Phils2, one has an arrive message in its inbox. When the arrive message in the inbox of Phils2 is selected to be served, it is popped from inbox$_2$ and its code is executed by sending three messages, a Request to Forks2, a Request to Forks3 and an Eat to itself. These method invocation requests are added to corresponding inboxes.

A Component in a Rebeca Model

In the dining philosophers example, one can take rebecs Phils0, Forks1 and Phils1 as an open component and other rebecs as the environment. This component can be denoted by Phils0||Forks1||Phils1. The only external messages coming to the component are Permit messages from Forks0 to Phils0 and from Forks2 to Phils1. It is assumed that these messages are always enabled.

Composition of Components in a Rebeca Model

If one composes two components, Phils0||Forks1||Phils1 and Phils1||Forks2||Phils2, one will have Phils0||Forks1||Phils1||Forks2||Phils2. It is the union of internal rebecs. Internal and external messages can be obtained knowing the universe of rebecs and internal rebecs.

Compositional Verification of Mutual Exclusion Property

The system safety requirement is that at any given time two neighboring philosophers cannot both hold the fork between them. It is specified in LTL-X as follows ($\oplus$ denotes addition in mod n and n is 4 in our example):

$$\varphi_{\text{sys}} = \Box(\bigwedge_{i=0}^{n-1} \neg(\text{Phils}_i.FR \wedge \text{Phils}_{i\oplus 1}.FL)).$$

One decides how to decompose the system according to the above property, which is the required system property. It is required to deduce the system property, φ_{sys}, from the properties of the components. So, Phils0||Phils1||Forks1 is considered as a component and the following property is proven by model checking:

$$\varphi_{\text{Phils0}||\text{Forks1}||\text{Phils1}} = \Box(\neg(\text{Phils0}.FR \wedge \text{Phils1}.FL)).$$

This property is proven by model checking using our tool. The tool can, automatically, generate the abstract model of the component out of the closed model and, then translate it to SMV. The SMV code is, then model checked by NuSMV model checker. Considering four similar components, $\text{Phils}_i||\text{Forks}_{i\oplus 1}||\text{Phils}_{i\oplus 1}, i = 0, \cdots, 4$ (with a shared philosopher between each pair of overlapping components), one has:

$$\varphi_{\text{Phils}_i||\text{Forks}_{i\oplus 1}||\text{Phils}_{i\oplus 1}} = \Box(\neg(\text{Phils}_i.FR \wedge \text{Phils}_{i\oplus 1}.FL)),$$

and, then using deduction, one can easily prove that:

$$\bigwedge_{i=0}^{n-1} \varphi_{\text{Phils}_i||\text{Forks}_{i\oplus 1}||\text{Phils}_{i\oplus 1}} \Rightarrow \varphi_{\text{sys}}.$$

By Corollary 1, in order to show that φ_{sys} is a property of sys, it suffices to find valid properties for each component, such that the conjunction of these properties yields to φ_{sys}. Thus, by what is shown above, one can conclude that φ_{sys} is a property of sys.

Using Deduction to Prove the Mutual Exclusion Property

In this example, it is obvious that the following formula holds:

$$\begin{aligned}
&\Box(\neg(\text{Phils0}.FR \wedge \text{Phils1}.FL)\\
&\wedge\Box(\neg(\text{Phils1}.FR \wedge \text{Phils2}.FL)\\
&\wedge\Box(\neg(\text{Phils2}.FR \wedge \text{Phils3}.FL)\\
&\wedge\Box(\neg(\text{Phils3}.FR \wedge \text{Phils0}.FL))\\
&\Rightarrow \Box(\neg(\text{Phils0}.FR \wedge \text{Phils1}.FL)\\
&\wedge\neg(\text{Phils1}.FR \wedge \text{Phils2}.FL)\\
&\wedge\neg(\text{Phils2}.FR \wedge \text{Phils3}.FL)\\
&\wedge\neg(\text{Phils3}.FR \wedge \text{Phils0}.FL)).
\end{aligned}$$

This will satisfy Condition 2 of Corollary 1. In this case, proving this formula is an easy deduction in linear temporal logic. But, for proving more complicated formulas, automated theorem provers can be used.

Model Checking Rebeca Code Using the Authors' Tool

A tool has been developed by the authors for automatic translation of Rebeca models to SMV [25]. Using this tool, some Rebeca examples were translated into SMV and, then NuSMV were used to check their safety properties. Data types supported by our tool are limited to those provided by NuSMV.

In all examples, there were bugs in the code, which were found by model checking. Some of the bugs were simple ones in initializing variables and some were more serious in communication and synchronization between rebecs. In [25], some examples are presented and it is shown how the state space is reduced by using compositional verification.

Here, the results obtained by using the tool to model check the dining philosopher example and, also, the results of applying the compositional verification approach to the same example are shown. CPU time and memory, used by NuSMV for computing total and reachable states, are summarized in Tables 1 and 2. Comparing data in Tables 1 and 2 shows that modeling the components, instead of the whole system, can help to reduce the number of reachable states. It can be seen that increasing the number of philosophers and forks will increase the size of total and reachable states in the example. The mutual exclusion property, discussed earlier, is satisfied for the code in Figure 2.

FUTURE WORK

Data abstraction in model checking Rebeca codes is, now, based on the back-end model checker approaches. The same data types are provided, as in SMV and PROMELA. In future work, for direct model checking of Rebeca codes, one also needs to consider the abstract interpretation of supported data types.

Table 1. Closed-world approach: Results generated by NuSMV.

Model	Reachable States	Total States	CPU Time (mm:ss)	Memory Usage (KByte)
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

Table 2. Component-based approach: Results generated by NuSMV

Model	Reachable States	Total States	CPU Time (mm:ss)	Memory Usage (KByte)
2 Phils, 1 Fork 2 External Forks	4132	1.16E+21	00:02	14076

Currently, the authors are working on extending Rebeca and the compositional verification method to an actor-based language with a mechanism for synchronous communication of signals. This extension involves new send statements of the form send(<var>,<signalid>,(<value>*)), which describe the emission of the signal <signalid>(<value>*) to the actor, denoted by var. This signal emission has to synchronize with the execution by the actor, denoted by var, of a corresponding receive statement receive(<signalid>*), which contains the particular signalid. Such a synchronization involves the transmission of the actual parameters, which will be stored by the receiver in the formal parameters of the signal. In extending Rebeca, high-level components are also provided, which generalize rebecs to sets of rebec-classes with well-defined interfaces. Abstracting from its internal class structure, such a component behaves as a rebec in Rebeca. Extending Rebeca is carried out in the context of the IST-2001-33522 EU project, Omega, on the correct development of real-time embedded systems in UML. This work will also involve the extension of Rebeca to real-time.

ACKNOWLEDGMENT

The authors wish to thank Michael Huth for his insightful comments on an earlier draft of this paper. The first author also wishes to thank the office for International and Scientific Cooperation of Sharif University of Technology and the British Council for providing the partial scholarship to visit Imperial College, London in Spring 2002.

REFERENCES

1. Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, Berlin, Germany (1992).
2. Manna, Z. and Pnueli, A., *Temporal Verification of Reactive Systems (Safety)*, Springer-Verlag, Berlin, Germany (1995).
3. Clarke, E.M., Grumberg, O. and Peled, D.A., *Model Checking*, The MIT Press, Cambridge, Massachusetts, USA (1999).
4. Huth, M.R. and Ryan, M., *Logic in Computer Science: Modelling and Reasoning About Systems*, Cambridge University Press, London, UK (2002).
5. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, USA (1990).
6. Agha, G., Mason, I., Smith, S. and Talcott, C. "A foundation for actor computation", *Journal of Functional Programming*, **7**, pp 1-72 (1997).
7. NuSMV user manual. ewblock available through http: //nusmv.irst.itc.it/NuSMV/ userman/index-v2.html.
8. Spin user manual, available through http://netlib.bell-labs.com/netlib/spin/what isspin.html.
9. 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).
10. Talcott, C. "Actor theories in rewriting logic", *Theoretical Computer Science*, **285**(2), pp 441-485 (Aug. 2002).
11. Gaspari, M. and Zavattaro, G. "An actor algebra for specifying distributed systems: The hurried philosophers case study", In *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, **2001**, pp 216-246, Springer-Verlag, Berlin, Germany (2001).
12. Sirjani, M., Movaghar, A. and Mousavi, M.R. "Compositional verification of an object-based reactive system", In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01)*, pp 114-118, Oxford, UK (April 2001).

13. Schacht, S. "Formal reasoning about actor programs using temporal logic", In *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, **2001**, pp 445-460, Springer-Verlag, Berlin, Germany (2001).

14. Rushby, J. "Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving", In *Proceedings of Theoretical and Practical Aspects of SPIN Model Checking*, Dams, D., Gerth, R., Leue, S. and Massink, M., Eds., Lecture Notes in Computer Science, **1680**, pp 1-11, Springer-Verlag, Berlin, Germany (1999).

15. Uribe, T.E. "Combinations of model checking and theorem proving", In *Proceedings of Workshop on Frontiers of Combining Systems*, Lecture Notes in Computer Science, **1794**, pp 151-170, Springer-Verlag, Berlin, Germany (2000).

16. Kesten, Y. and Pnueli, A. "Modularization and abstraction: The keys to practical formal verification", In *Proceedings of MFCS-98*, pp 54-71, Lecture Notes in Computer Science, **1450**, Springer-Verlag, Berlin, Germany (1998).

17. Rajan, S., Shankar, N. and Srivas, M.K. "An integration of model checking with automated proof checking", In *Proceedings of CAV'95*, P. Wolper, Ed., Lecture Notes in Computer Science, **939**, pp 84-97, Springer-Verlag, Berlin, Germany (1995).

18. Saidi, H. and Shankar, N. "Abstract and model check while you prove", In *Proceedings of CAV'99*, N. Halbwachs and D. Peled, Eds., Lecture Notes in Computer Science, **1633**, pp 443-453, Springer-Verlag, Berlin, Germany (1999).

19. Finkbeiner, B., Manna, Z. and Sipma, H.B. "Deductive verification of modular systems", In *Proceedings of CAV'99*, Lecture Notes in Computer Science, **1536**, pp 239-275, Springer-Verlag, Berlin, Germany (1998).

20. Clarke, E.M., Long, D.E. and McMillan, K.L. "Compositional model checking", In *Proceeding of the Fourth IEEE Symposium on Logic in Computer Science*, pp 353-362, Pasific Grove, CA, USA (1989).

21. Lynch, N.A. and Tattle, M. "An introduction to input/output automata", *CWI-Quarterly*, **2**(3), pp 219-246, Amsterdam, The Netherlands (1989).

22. Lynch, N.A., *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CS, USA (1996).

23. Alur, R. and Henzinger, T.A. "Computer aided verification", *Technical Report Draft* (1999).

24. Alur, R., Henzinger, T.A., Kupferman, O. and Vardi, M.Y. "Alternating refinement relations", In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, Lecture Notes in Computer Science, **1466**, pp 163-178, Springer-Verlag, Berlin, Germany (1998).

25. Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M. and Shali, A. "Model checking Rebeca by SMV", In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'03)*, pp 233–236, Southampton, UK (April 2003).

26. Sirjani, M. and Movaghar, A. "An actor-based model for formal modelling of reactive systems: Rebeca", *Technical Report CS-TR-80-01*, Tehran, Iran (2001).

27. McMillan, K. *Verification of Digital and Hybrid Systems*, Springer-Verlag, Berlin, Germany (2000).

28. Kupferman, O., Vardi, M.Y. and Wolper, P. "Module checking", *Information and Computation*, **164**(2), pp 322-344 (2001).

29. Tsay, Y. "Compositional verification in linear-time temporal logic", In *Proceedings of FOSSACS 2000*, Lecture Notes in Computer Science, **1784**, pp 344-358, Springer-Verlag, Berlin, Germany (2000).

30. De Roever, W.P. "The need for compositional proof systems: A survey", In W.P. Roever, H. Langmaack, and A. Pnueli, Eds., *Compositionality: The Significant Difference*, Lecture Notes in Computer Science, **1536**, pp 1-22, Springer-Verlag, Berlin, Germany (1998).

31. De Roever, W.P., de Boer, F.S. et al., *State-Based Proof Theory of Concurrency: From Noncompositional to Compositional Methods*, Cambridge University Press, Tracts in Theoretical Computer Science (2001).

32. Hoare, C.A.R., *Communications Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, USA (1985).

33. Roscoe, W.A., *Theory and Practice of Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, USA (1998).