FORMAL SPECIFICATION AND VERIFICATION OF CONCURRENT AND REACTIVE SYSTEMS

By

Marjan Sirjani

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
SHARIF UNIVERSITY OF TECHNOLOGY
AZADI AVE., TEHRAN IRAN
JUNE 2004

SHARIF UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF
COMPUTER ENGINEERING

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Formal Specification and Verification of Concurrent and Reactive Systems**" by **Marjan Sirjani** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: June 2004

External Examiner: ______________________
Dr. Meibodi

Research Supervisor: ______________________
Ali Movaghar

Examing Committee: ______________________
Seyed Hassan Mirian

____
Dr. Ardeshir

____
Dr. Sharifi

To my husband, my children, and my mother.

Table of Contents

List of Tables

List of Figures

Abstract

Rebeca (*Reactive Objects Language*) is an actor-based language for modeling concurrent and distributed systems. Providing a formal foundation, Rebeca is designed in an effort to bridge the gap between formal verification approaches and real applications. Its Java-like syntax and object-based style of modeling makes it easy to use for software engineers. A front-end tool is developed as an integrated environment to create Rebeca models and translate them into existing model-checker languages.

The encapsulated structure of reactive objects, and the asynchronous communication mechanism (with no blocking from either sender or receiver), lead to a natural modular design and loosely coupled modules. This is exploited to apply compositional verification and abstraction techniques and reduce the state space, and hence, make it possible to verify complicated reactive systems. In the compositional verification approach, sub-systems are defined based on an user-defined decomposition of the model. Sub-systems are more abstract than the model itself, and so we can reduce the state space of the model which makes it more amenable to model checking techniques. Using weak simulation relation between the constructs, it is proved that the abstraction techniques preserve a set of behavioral specifications in temporal logic.

Rebeca is then extended with a formal concept of components to provide a general framework which integrates both synchrony and asynchrony. Components are used to encapsulate a set of internal reactive objects. Components interact only via asynchronous and anonymous messages, while the internal reactive objects interact by asynchronous and also synchronous message passing mechanisms.

Semantics of Rebeca is specified compositionally, by mapping Rebeca models into the coordination language, Reo, and using Constraint Automata as its semantics. Modeling the coordination and communication mechanisms between reactive objects is done by Reo circuits, and the behavior of each reactive object is specified by constraint automata as a black-box component within the Reo circuit.

The Rebeca Verifier tool is used to model check typical simple case studies and some medium-sized case studies. The Experimental results show that where the computing paradigm in modeling distributed and concurrent systems is asynchronous, and the interprocess communication mechanism is message passing, Rebeca can be used easily and efficiently. Also, there are patterns of models in which our compositional verification approach can be applied in a scalable way and thus a significant reduction can be gained in verification of desired properties.

Acknowledgements

I would like to thank my supervisor, Professor Ali Movaghar, for his constant guidance and support during this research. Also, I would like to thank the referees, Professors Mohammad Ardeshir, Rassoul Jalili, Mohammad Reza Meybodi, Seyyed Hassan Mirian, and Mohsen Sharifi for their careful reading and suggestions on the thesis.

I wish to thank Michael Huth for his helpful suggestions and discussions; Ramtin Khosravi for his valuable comments on actor semantics; Mohammad Reza Mousavi for his help on the compositional verification approach for Rebeca; Frank de Boer for showing interest in Rebeca and suggesting extended Rebeca; Christel Baier for carefully reading the theorems and checking my proofs; Dennis Dams for his helpful discussions on the tool; Farhad Arbab and Jan Rutten for their support and inspiring comments. Marcello Bonsangue, Farzan Fallah, Jozef Hooman, and Clemens Kupke also helped me by their useful comments.

I would also like to thank Rebeca Group at University of Tehran and Sharif University of Technology, Amin Shali, Mohammad Mahdi Jaghouri, Hamed Iravanchi, Elham Mousavi, Mona Mojdeh, Sara Forghanizadeh, Niloufar Razavi, Fatemeh Alavizadeh, Niusha Hakimipour, Hossein Hojjat, Ehsan KhamesPanah, Azin Moallem , Elham Moazen, Hootan Nokhost, and Kamyar Rafati, not only for their help in developing and using the tool but also for their smart and creative comments. I wish to thank Salar Mesdaginia, Mohammad Sadegh Makarem, Mehdi Niamanesh and Hamid Reza Shahriari for spending some time studying and working on Rebeca.

Many thanks to my husband whom without his support I never could finish my work, my children who never understood why their mother still has to study, and my mother who sacrifices a lot during these years. Finally, many thanks to my father who taught me to respect two things: truth and mathematics, and my sister from whom I learned a lot in absence of my father.

Chapter 1

Introduction

1.1 Formal Modeling and Verification of Reactive Systems

Reactive systems are systems which have ongoing interactions with their environments, accepting requests and producing responses [40]. Such systems are increasingly used in applications where failure is considered as fatal, such as electronic commerce, high-speed communication networks, traffic control systems, avionics, and automated manufacturing. 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 formal verification.

A formal verification method consists of three major components: a model for describing the behavior of the system, a specification language to embody correctness requirements, and an analysis method to verify the behavior against the correctness requirements [41, 41, 17, 30].

Object-oriented modeling is widely used for representing reactive systems, with amenability to concurrency and distribution. The actor model [28, 6, 8] is a better candidate than customary object oriented models, because the units of distribution and concurrency are objects themselves and not threads, as in Java. This provides a simpler and more natural concurrency model. The actor model also promotes independent computing entities to support migration, distribution, dynamic reconfiguration, openness, and efficient parallel execution.

Much work has been done on formal methods with different kinds of models for system behavior and different verification approaches; also, the actor model is used in different ways for modeling open, distributed systems. But to the best of the author's knowledge, little is done on verifying actor languages (related work is discussed in Section 1.2).

In this thesis we present a formal method for specifying and verifying properties of reactive systems, using an actor-based model *Rebeca* [1] [53, 56]. Rebeca is inspired by the

[1]*Reactive Objects Language*

actors paradigm, but goes well beyond it by adding the concept of *components* and the ability to analyze a group of active objects as a component. Also, we have *classes* that active objects are instantiated from. Classes serve as templates for state, behavior, and the access interface; adding reusability in both modeling and verification process. Our method is supported by a front-end tool for the translation of Rebeca models into languages of existing model checkers. In order to cope with the problem of the state space explosion we propose a compositional verification approach which exploits the modular features of Rebeca models and their decompositionality into components.

More specifically, the key features of this thesis are:

- using the *actor-based* asynchronous event-driven model for the specification of reactive systems;
- introducing *components* as open (sub-)systems as a basis for compositional verification;
- presenting a *formal semantics* for the model and components, comprising their states, communications, state transitions, and the knowledge of accessible interfaces, which provides a formal basis for proving the correctness of our abstraction and reduction techniques;

- using different *abstraction techniques* based on the computing paradigm of the language which preserve a set of behavioral specifications in temporal logic, and which reduce the state space of a model, making it more suitable for model checking;

- establishing the soundness of these abstraction techniques by proving a *weak simulation* relation between the constructs;

- enriching Rebeca with a formal concept of reusable components and an additional communication mechanism based on synchronous message-passing, proposing extended Rebeca;

- presenting a tool for translating Rebeca models into target languages of existing model checkers, enabling *model checking* of actor-based models;

- modeling case studies in Rebeca and applying the *compositional verification* approach, using the specified abstraction techniques;

- providing compositional semantics of Rebeca, using the coordination language Reo to model the coordination and communication between reactive objects, and Constraint Automata to model each reactive object.

In Figure 1.1, we summarize the language, verification approach, underlying theories, and tool features, together with their relationships.

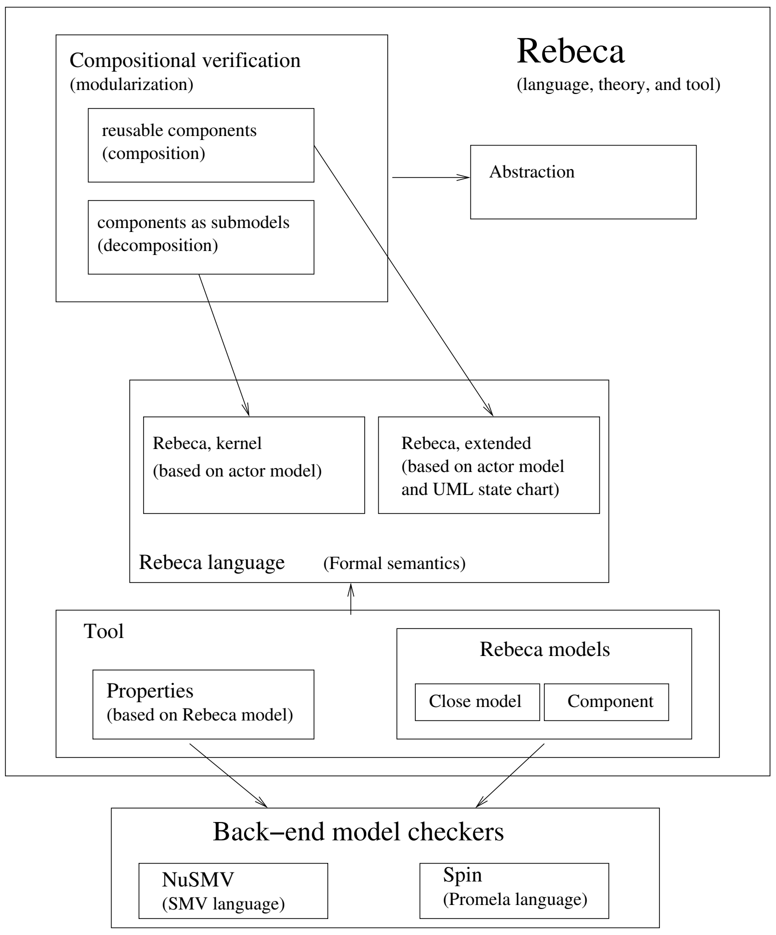


Figure 1.1: Rebeca: Language, Theory and Tool

1.2 Related Work

Verification techniques and corresponding tools have been developed for analyzing programs and models. Programs implementing real systems are usually too heavy and detailed for applying formal verification approaches. Hence, different abstraction techniques on both data and control are used to make the analysis process possible. On the other hand, formal modeling languages may be too abstract or too mathematical and not easy to be used by software engineers. Model checker tools, like SMV [1] and Spin [4], are developed with their own specific modeling languages which can be used directly for modeling systems and verifying their properties. Their modeling languages are designed to be suitable for applying model checking techniques and are not based on a formal semantics nor on a software development paradigm. They are sometimes used as back-end languages to which modeling or programming languages are translated.

Rebeca is different, by providing a powerful yet simple paradigm based on actor model, and an easy to use, java-like, object-based syntax for software engineers in modeling, and also a naturally decomposable model and independent modules which can be exploited in formal verification and model checking.

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

Examples of languages which provide a high-level of abstraction are CSP [2] [29], CCS [3] [45, 46], I/O Automata [39], and RML [4] [12]. RML is supported by the model checker Mocha [10]; and FDR [5] [49] is the proof and analysis tool for CSP.

Model Checking Existing Languages On the other hand, verification techniques and corresponding tools have also been developed for existing programming languages. For example, the NASA's Java PathFinder [27] is a translator from a subset of Java to Promela [4]. Its purpose is to establish a framework for verification and debugging of Java programs based on model checking. The Bandera Tool Set [23] 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

[2]Communicating Sequential Processes
[3]Calculus of Communicating Systems
[4]Reactive Modules Language
[5]Failures/Divergences Refinement

model-checking tools. SLAM [3] is a Microsoft's project for verification of C programs and debugging software via static analysis. The tools mentioned here, in principle can be applied directly to the verification of the actual implementation. However in practice such verification is only possible after an application of certain abstraction techniques to both the data and control [23].

Model Checkers Another approach is to use the language of a model checker itself in modeling concurrent and distributed systems. Some of these tools are successfully used in analyzing real systems, like NuSMV [1] and Spin. The NuSMV system is a tool for checking finite state systems against specification in the temporal logic LTL [6] and CTL [7] [24]. Spin is a widely distributed software package that supports the formal verification of distributed systems. Spin uses a high level language to specify systems descriptions, called Promela [8] and LTL is its specification langauge. However these languages are designed for model checking purposes and their formal semantics are usually not explicitly given. Using these tools also needs certain expertise.

Apart from the identification of suitable language characteristics which mainly concern modeling issues like the level of abstraction, modularity and usability for practitioners, the

[6]Linear Temporal Logic
[7]Computational Tree Logic
[8]Process meta language

two main approaches in formal verification both have their own deficiencies: Model checking in general suffers from the state-space explosion problem and deductive verification techniques require a high expertise and intensive interaction with the underlying theorem prover. In general, compositionality allows one to master both the complexity of the design and verification of software models. Decomposing a model into sub-models, verifying the properties of sub-models, and deducing the overall property is the main idea in compositional verification methods. Compositional verification can be exploited effectively only when the model is naturally decomposable [22].

Compositional Verification Compositional verification has been used in different ways in the analysis of models of concurrency. Abadi and Lamport [35, 5] explained an approach for composing specifications and verifying their properties. They used TLA[9] as their modeling language and also for describing properties, and applied assume-guarantee reasoning for compositional verification. Clarke, Long and McMillan [18] used interface processes to model the environment for a component. They modeled systems as finite transition systems and used CTL to specify their properties.

Input-output automata for modeling asynchronous distributed systems are introduced by Lynch and Tuttle [38, 39]. They showed how to construct modular and hierarchical

[9]Temporal Logic of Actions

correctness proofs for their models. Kesten and Pnueli [33] mentioned modularization and abstraction as the keys to practical formal verification, using fair Kripke structure as the computational model for reactive systems and temporal logic as a requirement specification language. An extension of bisimulation in a compositional proof of correctness of a protocol is used by Larsen and Milner in [37]. Alur and Henzinger [12] proposed RML for modeling a system and used a subset of linear temporal logic, alternating-time temporal logic, to specify its properties. RML supports compositional design and verification. Its compositional verification approach is assume-guarantee.

Actor Model Object-oriented models for concurrent systems have widely been proposed since the 1980s [6, 16, 21]. The *actor* model was originally introduced by Hewitt [28] as an agent-based language. It was later developed by Agha [6, 7, 8] 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 [8, 42, 59, 60, 26]. The actor model was first explained as a simple functional model [6, 7, 8], but several imperative languages have also been developed based on it [48, 64, 63]. Besides its theoretical basis, the actor model and languages provide a very useful framework for understanding and developing open distributed systems.

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

To the best of author's knowledge, there is hardly any work on the tool-supported formal verification of actors [56, 50]. In order to integrate the practice of software engineering and formal verification, Rebeca provides a rigorous semantic basis for an imperative view of actors. It is designed based on a powerful yet simple paradigm; providing the basic necessary constructs in a Java-like syntax which is easy to use for practitioners. In this thesis we show how to exploit the event-driven computation model of Rebeca in automated abstraction and compositional verification techniques which preserve LTL-X [10] and ACTL [11]

[10]LTL without next operator

[11]Universal fragment of CTL

properties.

A tool for translating Rebeca to SMV and Promela enables us to model check Rebeca codes both in closed and open forms. We have used our tool to show that our compositional verification approach reduces the state space in many practical cases [55]. A similar approach in using abstraction technique for model checking SDL [12] systems is discussed in [31].

1.3 Thesis Outline

Chapter 2 presents the modeling language Rebeca and its syntax and formal semantics for Rebeca models. Model checking Rebeca models, compositional verification, and components as open systems are explained in Chapter 3. Weak simulation, as an abstraction technique applied to Rebeca components, and the theorems used to formally justify our compositional verification approach are defined in this chapter. Chapter 4 explains the Rebeca model enriched by reusable components and the additional communication mechanism of synchronous message passing inside these components. In Chapter 5, we introduce our tool for automatic translation of Rebeca models into existing model-checking languages, SMV

[12] Specification and Description Language

and Promela, and the capabilities of the tool to automatic abstraction and modular verification of Rebeca models. Some case studies are presented in Chapter 6. Typical examples are modeled in Rebeca, and are model checked by the tool. In some examples we show how we can have a significant state space reduction using our compositional verification approach. Chapter 7 shows the compositional semantics of Rebeca, using mapping of Rebeca to the coordination language Reo and Constraint Automata. Chapter 8 concludes the thesis and shows the direction of possible future work.

Chapter 2

Rebeca: The Modeling Language

2.1 Introduction

Rebeca (*Reactive Objects Language*) [53, 56] is an actor-based language [28, 6] with a formal foundation. It can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model. It is also a platform for developing object-based concurrent systems in practice. Formal verification approaches are used to ensure correctness of concurrent and distributed systems.

Rebeca is similar to the actor model in that it has independent active objects, asynchronous message passing, unbounded buffers for messages, dynamically changing topology, and dynamic creation of active objects. We add class declarations to the syntax; classes act like templates for states, behavior, and interfaces of active objects. Also, we have the notion of a component as a set of concurrently executing active objects, and the role of internal and external active objects differs from the one in the original actor model [6]. Our components are sub-models which are the result of decomposing a closed model in order to apply compositional verification, and should not be confused with the concept of components in component-based modeling which are independent modules with well-defined interfaces.

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

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

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

2.2 Syntax

The syntax for reactive classes (reactive-object templates), rebecs (reactive class instantiations), and models (parallel composition of rebecs) is presented in Figure 2.1. The syntax of a `<reactive class>` definition is similar to Java, except for the definition of `<knownobjects>`. The rebecs included in the `<knownobjects>` part of a reactive class definition, are those rebecs whose message servers may be called by instances of this reactive class.

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

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

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

We use Producer-Consumer as a simple running example through this chapter to show the syntax and semantics of Rebeca. We start with a simple version and discuss different features of Rebeca by extending this example.

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

Figure 2.1: Reactive Class, Rebec and Model Definition Syntax

Example 1 (Producer-Consumer: a Rebeca model) *There is a buffer in which a producer puts its products and a consumer which takes the products from it. The producer cannot put a product in a full buffer and a consumer cannot take a product from an empty buffer. Also, the buffer is a critical section that both the producer and the consumer cannot put and take the products in/of the buffer at the same time.*

The system consists of reactive classes: *Buffer*, *Producer*, and *Consumer*, that are templates for defining a buffer, a producer, and a consumer (see Figure 2.2). The known rebecs of the Buffer are the Producer and the Consumer, and the known rebec of the Producer and the Consumer is only the Buffer. The Producer and the Consumer do not send messages to each other directly.

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

State variables are followed by message servers. Each reactive class includes an initial method as explained earlier. The Buffer has two message servers provided to get the

```
reactiveclass BufferManager(4) {
  knownobjects {
    Producer producer;
    Consumer consumer;
  }
  statevars {
    boolean empty;
    boolean full;
    boolean producerWaiting;
    boolean consumerWaiting;
    int bufferCount;
    int nextProduce;
    int nextConsume;
  }
  msgsrv initial() {
    bufferCount = 2;
    empty = true;
    full = false;
    producerWaiting = false;
    consumerWaiting = false;
    nextProduce = 0;
    nextConsume = 0;
  }
  msgsrv giveMeNextProduce() {
    if (!full)  {
      producer.produce(nextProduce);
    }
    else {
      producerWaiting = true;
    }
  }
  msgsrv giveMeNextConsume() {
    if (!empty) {
      consumer.consume(nextConsume);
    }
    else {
      consumerWaiting = true;
    }
  }
  msgsrv ackProduce() {
    nextProduce = (nextProduce + 1) %
bufferCount;
    if (nextProduce == nextConsume) {
      full = true;
    }
    empty = false;
    if (consumerWaiting) {
      consumer.consume(nextConsume);
      consumerWaiting = false;
    }
  }
  msgsrv ackConsume() {
    nextConsume = (nextConsume + 1) %
bufferCount;
    if (nextConsume == nextProduce) {
      empty = true;
    }
    full = false;
    if (producerWaiting) {
      producer.produce(nextProduce);
      producerWaiting = false;
    }
  }
}
```

```
reactiveclass Producer(2) {

  knownobjects {
    BufferManager bufferManager;
  }
  statevars {
  }
  msgsrv initial() {
    self.beginProduce();
  }

  msgsrv produce(int bufNum) {
    bufferManager.ackProduce();
    self.beginProduce();
  }

  msgsrv beginProduce() {
    bufferManager.giveMeNextProduce();
  }
}
reactiveclass Consumer(2) {

  knownobjects {
    BufferManager bufferManager;
  }
  statevars {
  }

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

  msgsrv consume(int bufNum) {
    bufferManager.ackConsume();
    self.beginConsume();
  }

  msgsrv beginConsume() {
    bufferManager.giveMeNextConsume();
  }

}

main {
  BufferManager bufferManager(producer,
consumer):();
  Producer producer(bufferManager):();
  Consumer consumer(bufferManager):();
}
```

Figure 2.2: Producer-Consumer Example

requests of the Producer and Consumer. Two other message servers get the acknowledgements of the Producer and Consumer and make the pointers and full/empty indicators up to date.

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

2.3 Semantics

The operational semantics of a reactive system can be defined as a labeled transition system. Labeled transition system [41] is a quadruple of a set of states (S), a set of labels (L), a transition relation on states (T), and a set of initial states of the system (S_0).

To define operational semantics of Rebeca, we first formalize the definitions of a rebec, a model, and their constituents (Figure 2.3). A rebec, r_i, with a unique identifier i, is defined as a triple $< V_i, M_i, K_i >$, where V_i is the set of its state variables, M_i is the set of its methods identifiers, and K_i is the set of all known rebecs of r_i.

For a Rebeca model, there is a universal set $\mathcal{I}$ of all *rebec identifiers* that are involved in the model, and a universal set $\mathcal{K}$ of all *known rebecs* of all members of $\mathcal{I}$.

A message *msg* is defined as: $msg = < sendid, i, mtdid >$, where *sendid* is the identifier of the sender, *i* is the identifier of the receiver, and *mtdid* denotes the method of receiver r_i which is called when the message is received. For the sake of simplicity, we ignore the message parameters in our semantics definition.

$\mathcal{U}$ is the set of all possible values for all types of variables that can be defined in a rebec, $\mathcal{V}_i = \{v | v : V_i \rightarrow \mathcal{U}\}$ is the set of possible values for variables of rebec *i*, and $\mathcal{V}_M = \bigcup_{i \in I_C} \mathcal{V}_i$.

Each rebec has a queue which can be defined as a finite sequence of messages. We denote the set of all finite sequences on a given set A as $seq(A)$. The mailbox of a component is like a multi-queue consisting of all the queues of its rebecs and including all the messages that have been sent from internal rebecs and have not yet been received.

Operational semantics of a Rebeca model is defined as a labeled transition system $M = (S, L, T, s_0)$, and is shown in Figure 2.4.

The state space of the model is

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

Figure 2.3: Summary of Definitions

$$\prod_{i=1}^{n} (S_i \times q_i), \tag{2.3.1}$$

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

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

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

Figure 2.4: Operational Semantics of a Rebeca Model.

- 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 rebecs' inboxes , change the local state, and create new rebecs;

- if new rebecs are created in the invocation of m, then the state space S *expands dynamically* from the one in (2.3.1) to

$$\Big(\prod_{i_{new}}(S_{i_{new}} \times q_{i_{new}})\Big) \times \prod_{i=1}^{n}(S_i \times q_i), \tag{2.3.2}$$

  where i_{new} ranges over the new rebecs created within that method invocation and s' is an element of (2.3.2);

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 rebecs. Regarding the *infinite* behavior of our semantics, communication is assumed to be fair [6]: 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 (Producer-Consumer: Initial state) *In Example 1, in the initial state there are a buffer, a producer and a consumer with their initial methods in their inboxes. So, we have three enabled transitions. Execution of the initial methods may cause sending messages to others or to self, and/or setting field variables.*

In the initial method of the rebec Buffer instance variables are initialized; and in the initial methods of the rebecs Producer and Consumer messages beginProduce and begin-Consume are sent to self.

Example 3 (Producer-consumer: state transitions) *Here we mention some of state transitions of the system.*

- *After execution of the initial method of the Producer, we have a beginProduce message in its inbox. When the beginProduce message in the inbox of Producer is selected to be served, it is popped from the inbox and its code is executed by sending* *the messages giveMeNextProduce to the Buffer. This message is added to the inbox of the Buffer.*

- *Execution of giveMeNextProduce method by the Buffer depends on the state variable $full$. If the $full$ variable is false, it causes sending a Produce message to the Producer, if the $full$ variable is true the state variable producerWaiting is changed to true to show that the Producer is waiting for an empty place to put its product in.*

Example 4 (Producer-Consumer: Dynamic creation and topology) *Another version of Producer-Consumer example is shown in Figure 2.5. In this example we show dynamic creation of rebec Buffer, and dynamic changing topology. Unlike in Example 1, where there is one instance of rebec Buffer, and a constant number of buffer elements available, here we allow dynamic creation of instances of rebec Buffer. Each buffer has one buffer element and a pointer to the next buffer. At the initial state one buffer is created, after one production the producer makes the buffer element of this buffer full. Then, for the next product another buffer is created. Buffer rebecs are like the nodes of a link list which are created on demand. The rebec Consumer starts to consume from the first node and moves forward in this link list. For the sake of simplicity, we do not model releasing of the consumed nodes.*

As shown in Figure 2.5, if a buffer is full and a Producer asks for a space by sending readyToProduce, then a new buffer is created and its known objects are set to be the Producer and Consumer. Then, the rebec id of this new rebec is sent to the Producer as a parameter of the message setBuffer. Execution of the message server of setBuffer will change the known object of the Producer to the Buffer which is newly created. This is an example of dynamic changing topology.

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

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

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

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

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

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

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

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

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

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

reactiveclass Consumer(2) {

   knownobjects {
      Buffer buffer;
   }
   statevars {
   }

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

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

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

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

Figure 2.5: Producer-Consumer Example with Dynamic Behavior

Chapter 3

Compositional Verification in Rebeca

3.1 Introduction

In formal verification we try to prove or disprove that a model satisfies some specifications. There are two basic approaches to this 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) [11].

One of the most important problems in model checking is the state-explosion problem. Compositional verification is a way to tackle this problem. In compositional verification the goal is to check properties of the components of a system and deduce global properties from these local properties. The main difficulty with this approach is that local properties are often not preserved at the global level [18].

In compositional verification, the specification of a system is decomposed into the properties of its components which are then verified separately. If we deduce that the system satisfies each local property, and show that the conjunction of the local properties implies the overall specification, then we can conclude that the system satisfies this specification too [36, 17, 44]. There has been a strong trend to use compositional approaches in formal verification of systems [34, 58, 61, 56]. The closest approach to our work is [51].

An overview In its simplest form assume a system consists of two modules P and Q which communicate with each other and also with their environment. We show this system as $P||Q$. If φ_P is the specification of P ($P \models \varphi_P$) and φ_Q is the specification of Q ($Q \models \varphi_Q$), we would like to reason according to the following rule [43]:

$P \models \varphi_P$

$Q \models \varphi_Q$

$\varphi_P \wedge \varphi_Q \Rightarrow \varphi$

$P||Q \models \varphi$

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

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

3.2 Model Checking Rebeca Models

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

Property specification language We use temporal logic as our property specification language. A *temporal formula* is constructed out of *state formulas* (assertions) to which we apply boolean connectives and temporal operators. State formulas are propositions defined over standard operations and relations over V, where $V = \bigcup_{i \in I} V_i$. We naturally do not consider the message queue contents in our state formulas. So, the properties are based on state variables of each rebec in the model. For model checking we abstract from the dynamic rebec creation and dynamic changing topology and consider it as the future work.

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

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

A front-end tool for model checking For model checking Rebeca codes we developed a tool which is explained in Section 5.2. Using this tool we can translate Rebeca codes to SMV [1] or Promela [4] and model check it by existing model checkers. In these tools, we have bounded data types, bounded message queues and in the future version a bounded number of rebec creation. Property specification language is based on the specification languages of back-end model checkers: LTL and CTL. The execution of each method is accomplished as an atomic operation. Message blocks are not implemented yet. An ongoing project is developing another tool for generating the state space from Rebeca codes and then model check it directly.

Example 5 (Producer-consumer: Model checking the code) *The producer-consumer Rebeca model in Figure 2.2 (with static behavior) is translated to SMV using our tool and then it is model checked using NuSMV. The total state space generated by NuSMV includes* $2.14e14$ *states and the reachable state are* 374 *states. The safety properties:*
$\Box!(buffer.full \wedge buffer.empty)$ *and*
$\Box(!buffer.empty \wedge !buffer.full) \rightarrow$
$!(buffer.nextProduce = buffer.nextConsume)$
are checked and are both true.

3.3 Compositional Verification in Rebeca: Components

In general, compositional verification may be exploited more effectively when the model is naturally decomposable [22]. In particular, a model consisting of inherently independent modules is suitable for compositional verification. Our actor-based model provides such independent modules because of the asynchronous communication mechanism which involve only an explicit non-blocking send operation.

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

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

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

Abstracting the queues from external messages Instead of putting external messages in an internal's inbox, they may be processed at any time, up to fair interleaving with the processing of requests in the inbox. This makes the model checking more efficient. Formally the behavior of the environment of a component is modeled by additional transitions which describe its messages sent to the component. In other words with respect to the external environment, a component behaves like an I/O automata [39], where inputs from the environment are always enabled.

External messages attached to components External messages coming into the component are present in all the states and we can imagine that they are like the members of a set that is constantly attached to all the states in the corresponding labeled transition system. In this way we abstract from buffering the external messages, and we do not need to have a special rebec or component modeling an environment. The environment of each component is modeled as extra transitions, added to operational semantics of a component. It is shown in the definition of the set of transition relations in the labeled transition system of Figure 3.2.

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

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

If in the code of a rebec, there is an explicit reference to the sender of a message then the behavior of the receiver depends on the sender of the message and our abstraction no more preserves the original behavior. For the sake of simplicity, in the definition of operational semantics of a component (Figure 3.2), we do not consider dynamic creation of rebecs (nor internal and external).

Deciding on how to decompose Internal rebecs constitute the “focus” of a particular analysis. Determination of such a focus may often be the result of intuition and experience with similar patterns of open systems and depends on the properties which have to be proved. It is the responsibility of the modeler and cannot be fully automated, although some work has been done in automating this process and eliminating user guidance [9]. There is no general approach in decomposing the system in components, components have to be selected carefully to lead to a smaller state space [36]. In many cases, specially when there is symmetry in the model, we can reduce the state space significantly.

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

The definitions for components are formalized in Figure 3.1 and operational semantics of a component C is summarized in Figure 3.2 [53].

Example 6 (A component in a Rebeca model) *In our producer-consumer example with no dynamic behavior (Figure 2.2), we can take rebecs $buffer$ and producer as an open*

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

Figure 3.1: Summary of Definitions for Components

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

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

Figure 3.2: Operational Semantics of a Component.

component, and the consumer as environment. This component can be denoted by $buffer||producer$. *The external messages coming to the component are ackConsume and giveMeNextConsume messages from the consumer to the buffer. We assume these messages are always enabled.*

Example 7 (Composition of components in a Rebeca model) *If we compose two components* $buffer||producer$ *and* $buffer||consumer$, *we will have* $buffer||producer||consumer$. *It is the union of internal rebecs which made a closed system here. Internal and external messages can be obtained knowing the universe of rebecs and internal rebecs.*

3.4 Formal Justifications

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

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

Weak simulation and property preservation Now, we explain the weak simulation relation among components in our model. Here, the model is simplified by ignoring dynamic creation and dynamic topology. Therefore, referring to the operational semantics of models in Section 2.3, the state space S won't expand dynamically from formula (2.3.1) to formula (2.3.2).

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

To define the weak simulation relation between two components, we use the operational semantics definition in Section 2.3 and the component definition in this section, and the following notation. A component C is a set of rebecs, the set of identifiers of internal rebecs of C is denoted by I_C and its state by s_C. The set of valuations for variables of component C in state s_C is denoted by $s_C.\mathcal{V}_C$. The inbox of component C is defined as a multi-queue, each queue is defined as a finite sequence of messages corresponding to an internal rebec as the receiver. The multi-queue of component C in state s_C is denoted by

$s_C.q_C$. As explained in Section 2.3, a label is a message of the form $< sendid, i, mtdid >$, where *sendid* is the identifier of the sender rebec, i is the identifier of the receiver rebec, and *mtdid* designates the method of i to be executed.

We also define a projection relation between two states. State $s_{C'}$ is a projection of state s_C (denoted by $s_{C'} \upharpoonright s_C$), 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 $q_{C'} \upharpoonright q_C$), if $I_{C'} \subseteq I_C$ and for each $i \in I_{C'}$ the sequence of messages $< sendid, i, mtdid >$ in q_C, ignoring messages with $sendid \in I_C - I_{C'}$, is the same as the sequence of messages in $q_{C'}$.

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

Definition 1**. **(Weak simulation) Given two components C and C' of a given model, represented by labeled transition systems $(S_C, T_C, s0_C)$ with signature of action labels L_C and $(S_{C'}, T_{C'}, s0_{C'})$ with signature of action labels $L_{C'}$, such that $I_{C'} \subseteq I_C$:

1. A relation $H \subseteq S_C \times S_{C'}$ is a weak simulation relation between C and C' if and only if for all $s_C \in S_C, s_{C'} \in S_{C'}$, if $H(s_C, s_{C'})$, then the following conditions hold:

   (a) $s_{C'} \upharpoonright s_C$.

   (b) for every state s_{C_1} and label $l \in L_C$ such that $(s_C, l, s_{C_1}) \in T_C$, there is a state $s_{C'_1}$

with the property that $s_{C'_1} = s_{C'}$ (if $l \notin L_{C'}$) or $(s_{C'}, l, s_{C'_1}) \in T_{C'}$ (if $l \in L_{C'}$) and $H(s_{C_1}, s_{C'_1})$.

2. We say that C' weakly simulates C (denoted by $C \leq C'$) if there exists a weak simulation relation H between C and C' such that $H(s_{C_0}, s_{C'_0})$.

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

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

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

1. To show that H is a weak simulation:

(a) $s_{C'} \upharpoonright s_C$ is in the definition of H.

(b) For the second condition, let $H(s_C, s_{C'})$ and $l \in L_C$ such that $(s_C, l, s_{C_1}) \in T_C$.

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

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

   - l is external for both C and C'. We know that $I_{C'} \subseteq I_C$, so $I'_C \subseteq I'_{C'}$ and the set of external messages to C is a subset of external messages to C'. Thus, l is enabled in $s_{C'}$.
   - l is internal for C and external for C'. It means that l is a message coming from a rebec in X, e.g., $p \in X$. When l is an external message for C', it is always enabled in all states, so it is enabled in $s_{C'}$.
   - l is internal for both C and C'. We know that $H(s_C, s_{C'})$, so $s_{C'} \upharpoonright s_C$ and

also $q_{C'} \upharpoonright q_C$. From the definition of projection we know that if l is on the top of the queue in s_C, it has to be on the top of the queue for $s_{C'}$ too. Thus, l is enabled in $s_{C'}$.

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

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

2. Now we show that $s_{C'_0} \upharpoonright s_{C_0}$. 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, $s_{C'_0}.q_{C'} \upharpoonright s_{C_0}.q_C$, because there are only init messages in both of them.

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

We have the following theorem which restricts the corresponding theorem of Clark et al [17] to safety properties.

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

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

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

1. For $i = 1,\ldots,n,\quad \varphi_{X_i}$ is a property of X_i, and

2. $(\bigwedge_{i=1}^{n} \varphi_{X_i}) \Rightarrow \varphi_R$ is valid.

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

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

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

Example 8 (Producer-consumer: verification of a property using abstraction) *The critical section in producer/consumer example is the buffer which is a shared object. The system safety requirement is that at any given time the producer and consumer do not access the buffer simultaneously. It is specified in LTL-X as follows:*

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

$$!(bufferManager.nextProduce = bufferManager.nextConsume)$$

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

In Example 8, we only use Theorem 1 and Theorem 2 to prove the property of the system by model checking a component. The following example shows how we need to use also Corollary 1 in order to prove the desired property.

3.5 An Example: Dining Philosophers

We use Rebeca to model the dining philosophers example. This system is discussed in various texts [29, 49, 33] and can serve as a simple example for showing how to use our 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 3.3 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 3.3). Our 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

- 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 *Phils*2, we have an *Arrive* message in its inbox. When the *Arrive* message in inbox of *Phils*2 is selected to be served, it is popped from $inbox_2$ and its code is executed by sending three messages, a *Request* to *Forks*2, a

```
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 3.3: Dining Philosophers Example

Request to *Forks*3, and an Eat to itself. These *method invocation requests* are added to corresponding inboxes.

A component in a Rebeca model In our dining philosophers example we can take rebecs *Phils*0, *Forks*1 and *Phils*1 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 *Forks*0 to *Phils*0 and from *Forks*2 to *Phils*1. We assume these messages are always enabled.

Composition of components in a Rebeca model If we compose two components $Phils0||$ $Forks1||Phils1$ and $Phils1||Forks2||Phils2$, we 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):

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

According to the above property as the system property to be proved, we decide how to decompose the system. We want to deduce the system property, φ_{sys}, from the properties of the components. So, we consider $Phils0||Phils1||Forks1$ as a component and prove the following property by model checking:

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

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 $Phils_i||Forks_{i\oplus 1}||Phils_{i\oplus 1}, i = 0,\ldots,4$ (with a shared philosopher between each pair of overlapping components), we have:

$$\varphi_{Phils_i||Forks_{i\oplus 1}||Phils_{i\oplus 1}} = \Box(\neg(Phils_i.FR \wedge Phils_{i\oplus 1}.FL))$$

and then using deduction we can easily prove that:

$\bigwedge_{i=0}^{n-1} \varphi_{Phils_i||Forks_{i\oplus1}||Phils_{i\oplus1}} \Rightarrow \varphi_{sys}$

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 conjunction of these properties yields to φ_{sys}. Thus, by what we showed above, we 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:

$\Box(\neg(Phils0.FR \wedge Phils1.FL) \wedge \Box(\neg(Phils1.FR \wedge Phils2.FL) \wedge$

$\Box(\neg(Phils2.FR \wedge Phils3.FL) \wedge \Box(\neg(Phils3.FR \wedge Phils0.FL)) \Rightarrow$

$\Box(\neg(Phils0.FR \wedge Phils1.FL) \wedge \neg(Phils1.FR \wedge Phils2.FL) \wedge$

$\neg(Phils2.FR \wedge Phils3.FL) \wedge \neg(Phils3.FR \wedge Phils0.FL))$

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.

Chapter 4

Extended Rebeca

4.1 Introduction

A model in Rebeca consists of a set of *rebec*s [1] which are concurrently executed. Rebecs are encapsulated active objects, with no shared variables. Each rebec is instantiated from a *class* and has a single thread of execution which is triggered by reading messages from an unbounded queue. Each message specifies a unique method to be invoked when the message is serviced. When a message is read from the queue, its method is invoked and the message is deleted from the queue. Note that reading messages, thus, drives the computation of a rebec. Rebecs do not provide an explicit control over the message queue. In kernel Rebeca (before extension), active objects communicate only via asynchronous message passing. Because of this asynchronous communication mechanism, with only an asynchronous send operation and no explicit receive operation, methods can be executed

[1] $\underline{re}active\ o\underline{b}j\underline{ec}t$

atomically. Dynamic changing topology and dynamic rebec creation is defined in formal semantics of Rebeca.

We exploited the specific features of this actor-based model of computation in a compositional verification technique for model-checking safety properties [56]. In this chapter, we enrich the actor-based model of computation with a formal concept of components. Components encapsulate their internal structure which is given by a set of rebecs. The methods of these rebecs however provide an additional communication mechanism based on synchronous message-passing. The interaction between the rebecs of a component is encapsulated. The concept of components in this chapter shouldn't be confused by what we introduced in Chapter 3. Here, components are no more sub-models which are the result of decomposing a closed model in order to apply compositional verification. Components in this chapter, present independent modules with well-defined interfaces which can be used in modeling as well as verification.

The motivation is to provide a general framework which integrates in a formally consistent manner, both synchrony and asynchrony. We introduce components for integrating different communication patterns (synchronous and asynchronous), at different levels of abstraction. At the highest level of abstraction, components only interact asynchronously via broadcasting anonymous messages. At a lower level of abstraction (within a component), computations, on the one hand are driven by asynchronous messages, and on the other hand can be synchronized by a handshaking communication mechanism.

The external observable behavior of a component is described by an interface. This interface specifies a set of provided and required message signatures which is the union of provided and required messages of internal rebecs. These signatures specify the message name and the types of its parameters. In order to enforce encapsulation, we do not allow a rebec class as a parameter-type. A rebec of a component can only send instances of the required message signatures. These messages will be broadcasted to all the other components of the system. On the other hand, upon receiving a message from environment, a component will broadcast the instances of the provided massage signatures to all internal rebecs. Each internal rebec of the component will store these provided messages in its message queue, only if the service is offered by it.

4.2 Syntax

Rebeca is a class-based language in which classes are templates for instantiating rebecs with specified interfaces, instance variables, and method definitions. Variables are strongly typed. In order to increase the modeling power of actor-based languages, we extend the asynchronous communication mechanism of Rebeca with synchronous message passing and a mechanism for broadcasting anonymous messages. Synchronous messages are specified only as a signature specifying the name of the message and the types of its parameters.

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

Definition 1 (Basic definitions) .

- The predefined types T: *Int* for integers, *Bool* for Booleans, and *Reb* for rebec names, i.e., identities of the active object in Rebeca.

- The set *Var* is the set of variables of type T with typical elements $x_1, x_2, \ldots, x_n$, including instance variables and also local variables. We show local variables by $u_1, \ldots, u_n$, values by $v_1, \ldots, v_n$, and rebec names by $r, r', \ldots$.

- The set *Val* is the union of all the values for all the types, i.e., all the values for type *Int,* $\{True, False\}$ for type *Bool,* and all the rebec names for type *Reb*.

- The set *Mes* is the set of messages with typical elements $m, m_1, \ldots, m_n$.

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

Definition 2 (Syntax of extended Rebeca) .

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

$$a ::= \; x = e \;\mid\; x = new(A) \;\mid\; [x.]m(e_1, ..., e_n) \;\mid\; receive(m_1, ..., m_n)$$

$$S ::= \; a; S \;\mid\; a$$

$$mtd ::= \; m(u_1 : t_1, \ldots, u_n : t_n)[: S]$$

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

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

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

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

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

4.3 Semantics of Rebecs

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

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

Definition 3 (Local configuration)

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

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

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

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

- a label $r.m(v_1,\ldots,v_n)$, where r denotes the executing rebec itself, indicates the reception of the message $m(v_1,\ldots,v_n)$.

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

Definition 4 (Local transition for processing message queue)

When the point of control is at the end of a method, its execution is finished which is denoted by nil. If there is a message at the top of the rebec's queue it is popped and the corresponding method is called for execution. The parameter values are substituted before execution. It is worthwhile to observe here that we don't have recursion in methods so we don't need to worry about fresh local variables. The above is formalized by the following transition: $\langle r,\sigma,nil,q.m(v_1,\ldots,v_n)\rangle \xrightarrow{\tau} \langle r,\sigma',S,q\rangle$

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

Definition 5 (Local transition for assignment)

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

$$\langle r,\sigma,x=e;S,q\rangle \xrightarrow{\tau} \langle r,\sigma',S,q\rangle,$$

where $\sigma' = \sigma\{\sigma(e)/x\}$ and $\sigma(e)$ denotes the value of expression e in σ.

Definition 6 (Local transitions for send)

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

- $\langle r,\sigma,m(e_1,\ldots,e_n);S,q\rangle \xrightarrow{m(\bar{v})} \langle r,\sigma,S,q\rangle$

  where $\bar{v} = (v_1,\ldots,v_n)$, and $v_i = \sigma(e_i)$.

- $\langle r,\sigma,x.m(e_1,\ldots,e_n);S,q\rangle \xrightarrow{r'.m(\bar{v})} \langle r,\sigma,S,q\rangle$

  where $\sigma(x) = r'$, $r \neq r'$, $\bar{v} = (v_1,\ldots,v_n)$, and $v_i = \sigma(e_i)$.

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

  where $\sigma(x) = r$, and $v_i = \sigma(e_i)$.

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

Definition 7 (Local transitions for receive)

We distinguish between the reception of synchronous and asynchronous messages:

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

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

- Asynchronous messages for which the receiving rebec does not have a corresponding receiver are simply discarded:

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

- Finally, we have the following transition which described the reception of a synchronous message:

$$\langle r, \sigma, receive(m_1, \ldots, m_n); S, q\rangle \xrightarrow{r.m(\bar{v})} \langle r, \sigma', S, q\rangle$$

where, given the method definition $m(u_1,\ldots,u_n)$ and $\bar{v} = (v_1,\ldots,v_n)$,

$\sigma' = \sigma\{v_1/u_1,\ldots,v_n/u_n\}$.

Definition 8 (Local transition for creation)

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

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

4.4 Components

A component encapsulates its internal structure which is given by a set of rebecs.

Definition 9 (Component configuration)

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

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

which indicate sending and receiving anonymous asynchronous message *m*, respectively. Communications between rebecs of a component are hidden.

Definition 10 (Component transition for internal communication)

The following transition describes internal synchronous and asynchronous message passing,

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

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

Definition 11 (Component transition for send)

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

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

Definition 12 (Component transition for receive)

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

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

Definition 13 (Component transition for creation)

The following rule describes the creation of an internal rebec.

$$\frac{l_i \xrightarrow{r_{n+1}} l'_i}{\{l_1,\ldots,l_i,\ldots,l_n\} \xrightarrow{\tau} \{l_1,\ldots,l'_i,\ldots,l_n,l_{n+1}\}}$$

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

Definition 14 (Component internal transition)

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

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

A global model simply consists of a set of components.

Definition 15 (Global configuration)

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

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

Definition 16 (Global transition for communication)

This transition describes the broadcasting mechanism of asynchronous anonymous messages.

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

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

Definition 17 (Global internal transition)

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

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

4.5 An Example: Bridge Controller

Here, we explain a simple example to show our modeling approach. Consider a bridge with a one-way track where only one train can pass at a time. This example can be easily extended to multiple tracks. Trains enter the bridge from its left side, pass it, and exit from the right side. Rebeca code for this example is shown in Figure 4.1. We model the two ends of the bridge by two objects controlling these ends. These objects are described by the classes *leftController* and *rightController*. The rebecs *theLeftCtrl* and *theRightCtrl* are instantiated from these two classes and together form a component. Trains are modeled by the *Train* class. Many trains can be instantiated from this class, but in this example we only have two trains instantiated. Each single train instance is modeled as a component. Trains announce their arrival by broadcasting the anonymous message *Arrive(MyTrainNr)* to the Controller component. To this message only the *leftController* will react by broadcasting the *YouMayPass(MyTrainNr)* message after which the *leftController* waits for the synchronous message passed. The message *YouMayPass(MyTrainNr)* will be received by both trains, however only the train identified by *MyTrainNr* will enter the bridge (after the test the other train will remove the message from its queue and wait for the next message), Passing the bridge is modeled by broadcasting the message *Leave* to the Controller component. To this message only the *rightController* will react by sending the synchronous

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




activeclass rightController() {
    knownobjects { leftController left; }
    provide { Leave; }
    request {}
    statevars { int trainsout; }
    msgsrv initial() {
        trainsout = 0;
    }
    msgsrv Leave() {
        trainsout = trainsout + 1;
        left.passed();
    }
}
{train2};
```

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

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

     {theLeftCtrl, theRightCtrl};
 }
```

Figure 4.1: Bridge Controller Example, Modeled in Extended Rebeca

message passed to the *leftController* which enables the *leftController* to receive new *Arrive* messages. Note that thus no trains are allowed to enter the bridge (by executing *GoOnThe-Bridge*) while the *leftController* is suspended). Two variables, *trainsin* and *trainsout*, are added to the code for verification purposes, explained in Section4.6.

In Figure 4.1, encapsulation of rebecs in a component and also three types of message passing can be seen. The two left and right controllers of the bridge are tightly coupled and are encapsulated in a component. It allows the synchronous message passing between them. Trains are independent objects and can communicate by broadcasting asynchronous messages. It is also shown that the broadcasted messages are only serviced by the provider rebecs.

Further, in Section 4.6, we will show the application of our modular verification approach on this example.

4.6 Formal Verification of Properties

Formal verification of properties for components, is a problem of *model checking of open systems*. By an *open system*, we mean a system that interacts with its environment and whose behavior depends on this interaction; unlike a *closed system*, whose behavior is completely determined by the state of the system. The crucial point in model checking an open system, which is usually referred to as *module checking,* is modeling the environment. To model the nondeterminism, an environment can be modeled as a general process with arbitrary behavior [62, 34, 10, 9].

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

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

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

Definition 18 (Environment of a component)

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

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

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

Definition 19 (Queue abstraction)

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

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

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

Theorem 1 (Correctness of the queue abstraction)

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

Now, we will proceed by defining the weak simulation relation between two models in Rebeca, first one consists of a component and the general environment, and the second one consists of that component with any arbitrary component. Here, according to Theorem 1, we can develop the preorder relation upon the abstracted model. Next, we define a general definition for weak simulation, and then continue by applying the definition on our specific models.

Definition 20 (Weak Simulation)

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

1. A relation H is a R-simulation between Σ_1 and Σ_2, where $H, R \subseteq S_1 \times S_2$, if and only if for all s_1 and s_2, if $H(s_1, s_2)$ then the following conditions holds:
   (a) $R(s_1, s_2)$.
   (b) For every state s'_1 such that $(s_1, s'_1) \in T_1$ we have $(s'_1, s_2) \in H$ (stuttering), or else there is a state s'_2 with the property that $(s_2, s'_2) \in T_2$ and $(s'_1, s'_2) \in H$.
2. We say that Σ_2 R-simulates Σ_1 (denoted by $\Sigma_1 \leq \Sigma_2$) if there exists a R-simulation H between Σ_1 and Σ_2 such that $H(I_1, I_2)$.

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

Based on Definitions 18 , 19 and 20, we have the following theorem.

Theorem 2 (Weak simulation between models)

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

The formal proof is quite similar to one in [53]. Intuitively, it is based on the fact that in each state of the transition system Σ_2, the enabled transitions are a subset of enabled transitions in the correspondent state of the transition system Σ_1. By correspondent states, we mean the states in Σ_1 and Σ_2 which satisfy the simulation relation, starting from the initial state. This is because of the definition of general environment which covers all the possible messages, and hence transitions.

Definition 21 (Satisfaction relation)

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

2. Given a CTL formula ϕ, we say that $C \models \phi$ iff ϕ holds in the initial state of C.

We have the following theorem from [17].

Theorem 3 (Property preservation)

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

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

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

Execution of a method is no more an atomic operation, it may be interrupted as a consequence of sending or receiving a synchronous message, i.e., sender and receiver have to wait at the handshaking point until the other pair arrives. Therefore, it is no more the case that every transition is only taking a message from top of a rebec's queue and execute its corresponding method. It should be first checked whether the rebec is in a hold state at a rendezvous point, waiting for a matching send or receive to happen. Also, while executing a method, a rebec may reach a handshaking point for which another rebec is already waiting. In this case, the values of the parameters are passed to the receiver and the method execution is continued. Also, a flag for the pair is set, indicating that the handshaking has been taken place and the hold rebec can continue its execution in its next turn.

Verifying Properties of the Bridge Controller Consider the Bridge controller model in Figure 4.1, explained in Section 4.5. Safety and progress properties of the model can be verified by translating the Rebeca code into SMV or Promela using our tool described in the next section. Mutual exclusion is that at any moment only one train should be on the bridge, progress is that trains should finally pass the bridge, and no starvation is both trains finally pass the bridge. These properties can be checked using the state variable *OnTheBridge* of trains. The LTL formula for checking these properties are the followings ($\Box$ denotes *always* and $\Diamond$ denotes *finally*) :

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

These properties should also be translated to the specification languages of NuSMV and Promela. For module checking, we consider the controller component. Our purpose is to check its properties in all the possible conditions, i.e., in a general environment. In this example, a general environment is an environment sending to controller component, all of its provided messages in a nondeterministic way. The provided messages are *Arrive* serviced by *leftController* and *Leave* serviced by *rightController*. Our tool supports module checking components by modeling the abstracted environment. Here, it means that in those states where the statement to be executed is nil, two transitions corresponding to execution of methods *Arrive* and *Leave* are always enabled.

In Module checking the controller component, we remove all other rebecs including their state variables and queues. So, we cannot reach *OnTheBridge* variables to check the properties. In this case, state variables *trainsin* of *theLeftCtrl* and *trainsout* of the *theRightCtrl* can be used to check the safety property:

$\Box\ (theLeftCtrl.trainsin - theRightCtrl.trainsout \leq 1)$. To check the deadlock property, a variable has to be added to show sending *YouMayPass* message.

Chapter 5

A Tool for Model Checking Rebeca

5.1 Introduction

Rebeca Verifier is an environment to create Rebeca models, edit them, and translate them into SMV [1] or Promela [4]. Also, modeler 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 [1] or Spin [4] respectively. Modular verification and automatic abstraction is also supported by the tool. 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 as well. To build the component model out of components, a general environment is simulated by allowing all possible interactions. Figure 5.1 is a block diagram showing the language, verification approach, under lined theories, and tool features, together with their relationships.

5.2 The Rebeca Verifier

The Rebeca Verifier [55, 54, 57] 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 also 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. Modular verification

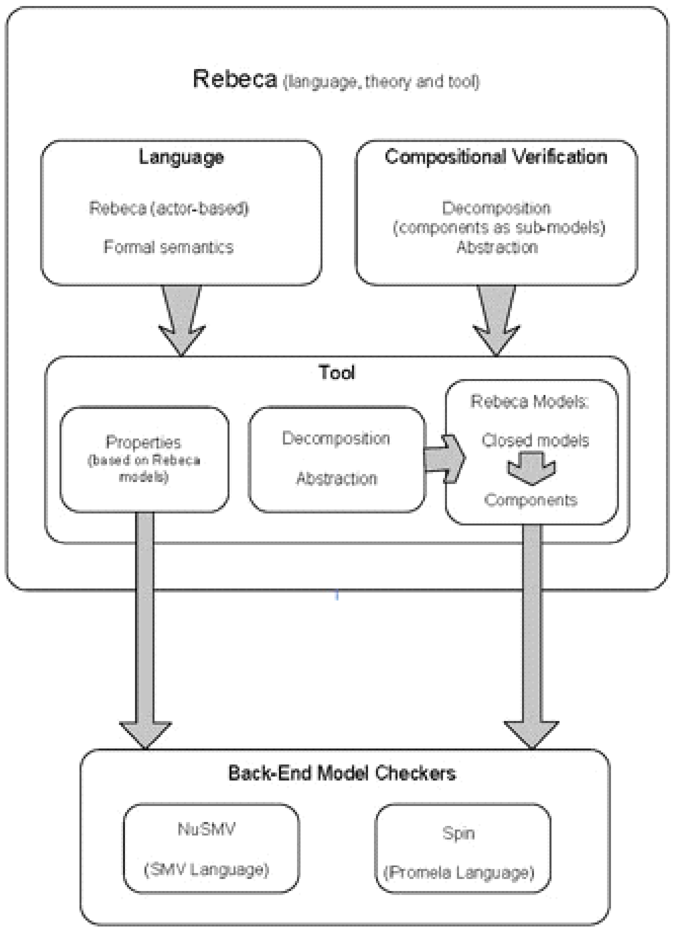


Figure 5.1: Rebeca: Supported by Tool

supported by tool, is further described in Section 5.5. Figure 5.2, 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 5.3. 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 [25] 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 generater 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. A simple example in Section 5.6 shows the approach.

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.

5.3 Translating Rebeca to SMV

NuSMV [1] 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

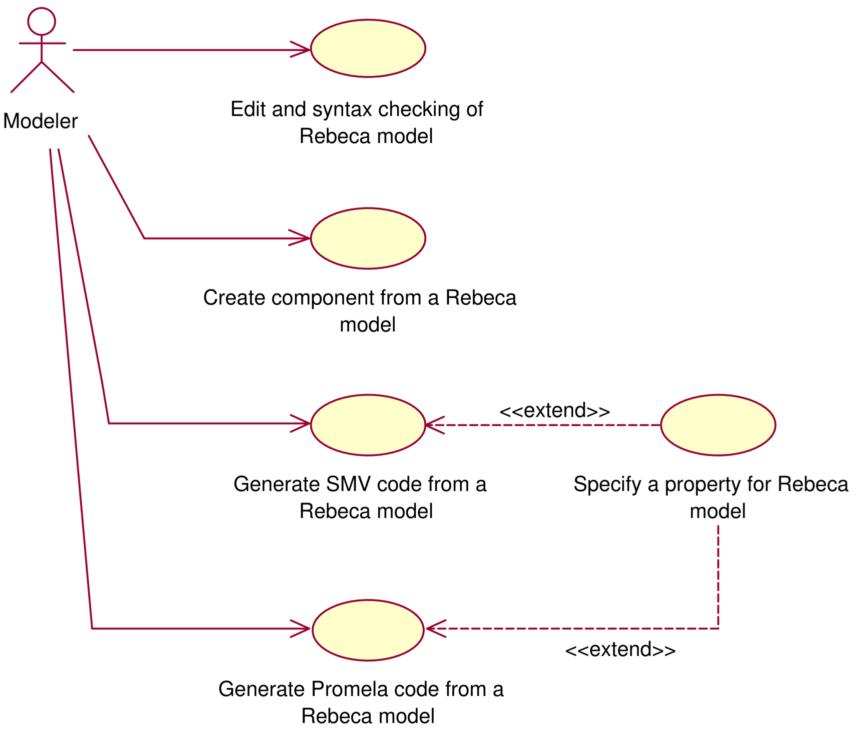

Modeler
Edit and syntax checking of Rebeca model
Create component from a Rebeca model
Generate SMV code from a Rebeca model
<<extend>>
Specify a property for Rebeca model
Generate Promela code from a Rebeca model
<<extend>>

Figure 5.2: Use Case Diagram of Rebeca Verifier

in that process in parallel. 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 generater is used to produce SMV codes from Rebeca models [55]. The mapping from Rebeca constructs to SMV is shown in Table 5.1. Each class in Rebeca is translated to a module in SMV and for each rebec a process is defined.

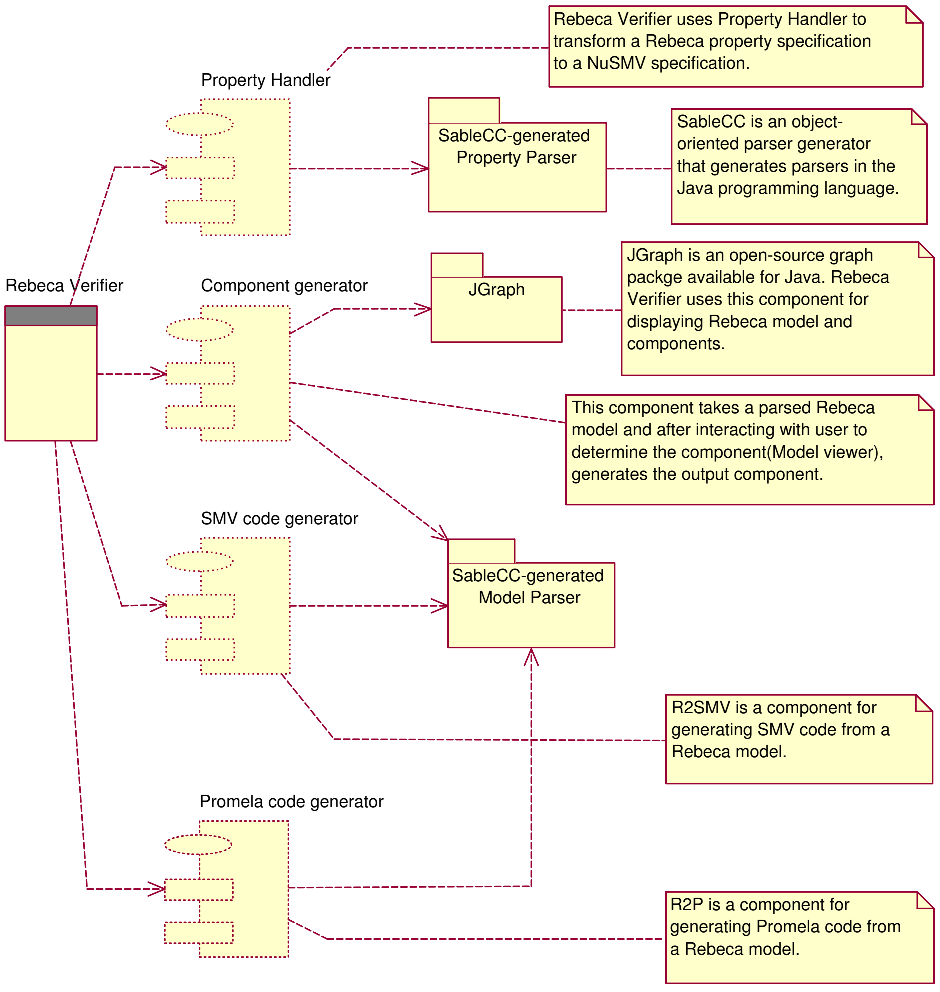

Rebeca Verifier uses Property Handler to transform a Rebeca property specification to a NuSMV specification.
Property Handler
SableCC-generated Property Parser
SableCC is an object-oriented parser generator that generates parsers in the Java programming language.
Rebeca Verifier
Component generator
JGraph
JGraph is an open-source graph packge available for Java. Rebeca Verifier uses this component for displaying Rebeca model and components.
This component takes a parsed Rebeca model and after interacting with user to determine the component(Model viewer), generates the output component.
SMV code generator
SableCC-generated Model Parser
R2SMV is a component for generating SMV code from a Rebeca model.
Promela code generator
R2P is a component for generating Promela code from a Rebeca model.

Figure 5.3: Component Diagram of Rebeca Verifier

Each method of a rebec has to be executed in an atomic step, it can be done 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 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.

Execution of rebecs methods depends on the messages in the message queue, in each step the message on the top of the queue is taken and its corresponding method is executed atomically. 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

Table 5.1: Mapping Rebeca Constructs to SMV

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

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.

5.4 Translating Rebeca to Promela

Spin [4] 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 generater is used to produce Promela codes from Rebeca models. The mapping from Rebeca constructs to Promela is shown in Table 5.2. 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 the corresponding process in Promela.

Table 5.2: Mapping Rebeca Constructs to Promela

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

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. In extended Rebeca (which is presented in [52] and 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 problem in mapping an object-based code to Promela 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.

5.5 Creating Components and Module Checking

The compositional verification approach for Rebeca models is explained in Chapter 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 [62, 34]. 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. A simple example in Section 5.6 shows the approach.

5.6 An Example: Bridge Controller

Here, we explain a simple example to show our modeling and verification approach using Rebeca Verifier. 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.

Figure 5.4 shows the Rebeca code for bridge controller example. There are two classes, one for the bridge controller and one for the trains. 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.

```
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 5.4: Bridge Controller Example

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 *YouMay-Pass* 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 checks if the other train is waiting to pass and sends a *YouMayPass* message to it if it is waiting and sets the lights properly.

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 can include shared rebecs, and use deduction to prove the required property of the system. In the bridge controller example, the required properties are that at any moment only one train should be on the bridge (mutual exclusion), trains should finally pass the bridge (progress), and both trains finally pass the bridge (no starvation). So the system properties are specified in LTL (Linear Temporal Logic) [24] as follows:

- Mutual exclusion: $\Box !(train1.OnTheBridge \mathbin{\&\&} train2.OnTheBridge)$
- Progress: $\Box\Diamond(train1.OnTheBridge \mathbin{||} train2.OnTheBridge)$
- No starvation: $\Box(\Diamond(train1.OnTheBridge) \mathbin{\&\&} \Diamond(train2.OnTheBridge))$

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 5.5 shows a snapshot of the system, creating the required component. For the component in Figure 5.5, 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*:

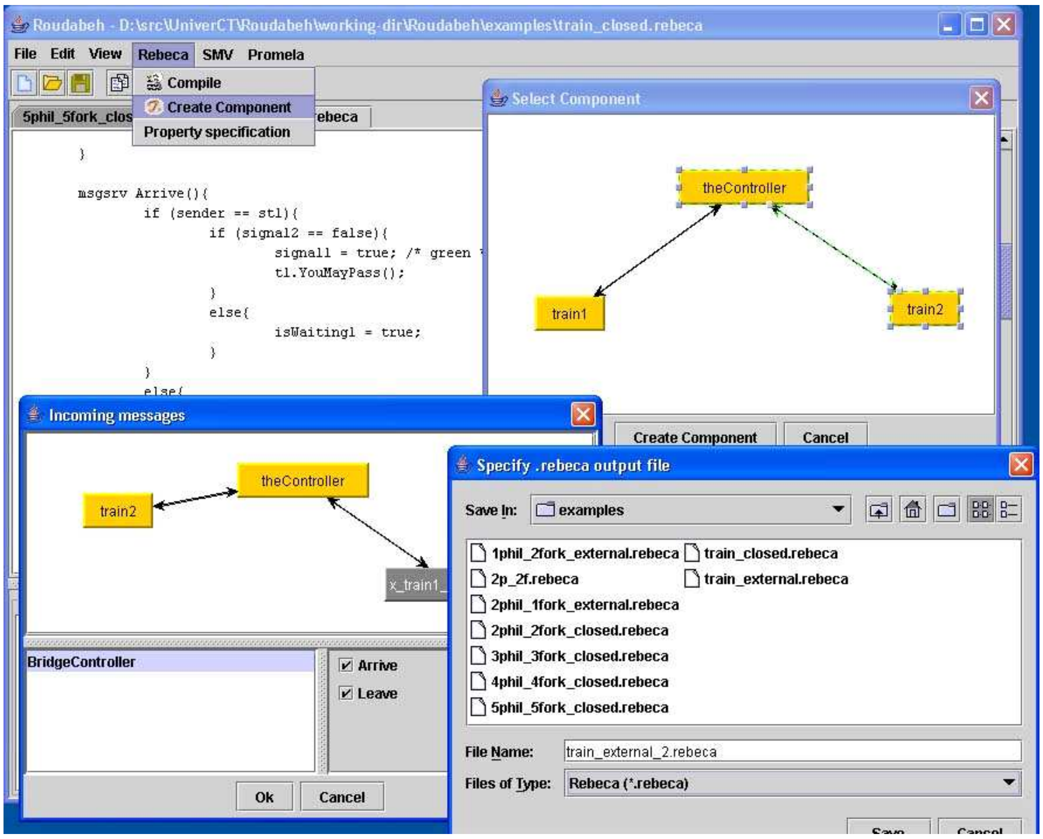
- Mutual exclusion: $\Box\ !(theController.signal1 \mathbin{\&\&} theController.signal2)$
- Progress: $\Box\Diamond(theController.signal1 \mathbin{||} theController.signal2)$
- No starvation: $\Box(\Diamond(theController.signal1) \mathbin{\&\&} \Diamond(theController.signal2))$

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.



```
        }

        msgsrv Arrive(){
                if (sender == stl){
                        if (signal2 == false){
                                signal1 = true; /* green 
                                tl.YouMayPass();
                        }
                        else{
                                isWaiting1 = true;
                        }
                }
                else{
```

Figure 5.5: A Snapshot of the Tool, Creating a Component from Bridge Controller Example.

Chapter 6

Case Studies

6.1 Introduction

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 [47, 19]). We selected typical case studies from [39] 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) [39], *the commit problem* [39], *trains and the bridge controller* [11], *dining philosophers* [29, 49, 39], *readers and writers*, and *gossiping girls*. These case studies are translated to SMV or Promela or both, and are included at Rebeca Home page [2]. 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. Although, comparing NuSMV and Spin, considering a number of criteria, is considered as our future work. In the following we shortly explain a number of case studies, for the first three examples compositional verification approach is applied and state space reduction is gained. Module checking by Rebeca Verifier is currently supported by SMV code generater, 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 the first three case studies. 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.

Table 6.1: Trains and the Controller: 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	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

6.2 Bridge Controller

This example is explained in Chapter 5. In this example there are two trains travelling opposite to each other. There is a bridge in the path, which is not wide enough to accommodate both trains. There is also a bridge controller which has to prevent collisions between the two trains. Model checking results are summarized in table 6.1. 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.

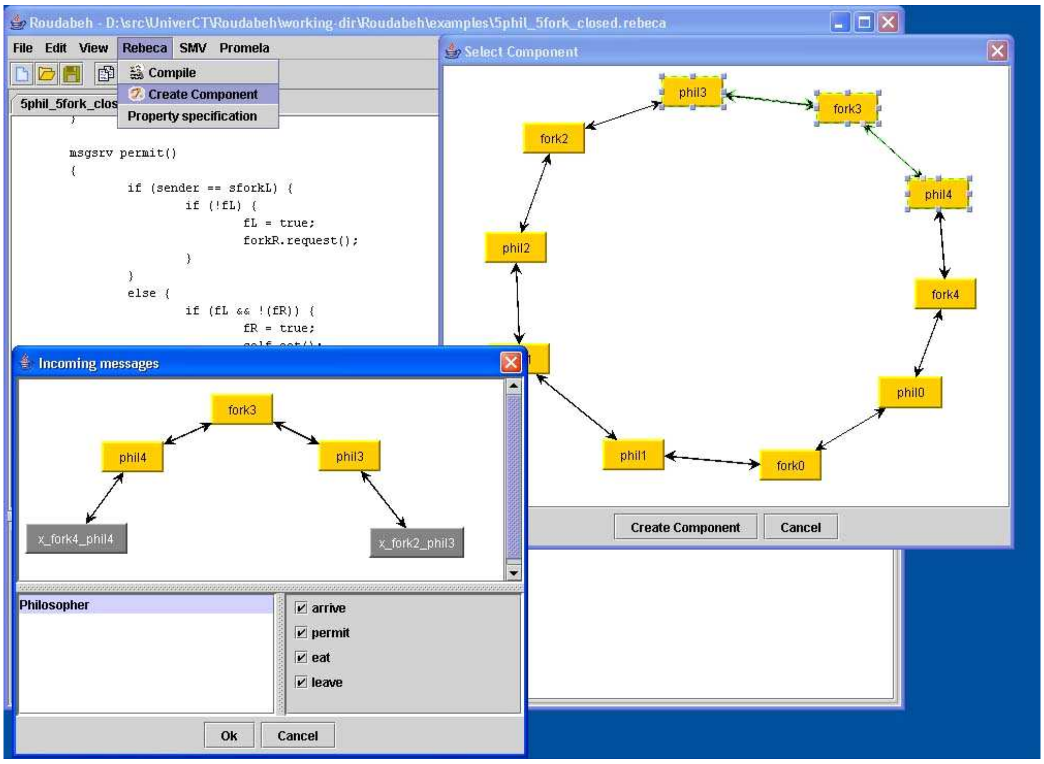
We checked the queue-overflow condition and found out that queue length of two for the trains and four for the bridge controller is enough for preventing overflow.

6.3 Dining Philosophers

This example is explained in Chapter 3. 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 6.2. 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



```
        }

        msgsrv permit()
        {
                if (sender == sforkL) {
                        if (!fL) {
                                fL = true;
                                forkR.request();
                        }
                }
                else {
                        if (fL && !(fR)) {
                                fR = true;
```

Figure 6.1: A Snapshot of the Tool, Creating a Component from Dining Philosophers Example.

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 6.1 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.

Table 6.2: 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	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 6.3: Readers and Writer: 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

6.4 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.

6.5 Leader Election

Leader election example is selecting a node as a leader in a ring of n nodes. In this ring, each node has a unique identifier which is supposed to be an integer number. The leader shall be the node with the least *id* among all of the *id*s. Each node knows its own *id* and can send messages to one of the nodes next to it or both of them, i.e., the ring can be uni- or bi-directional. The leader is selected by sending messages to other nodes.

At the beginning, each node introduces itself as the leader to its neighbor(s). Each node compares the *id* in the received message to its own $leader\ id$, and substitutes its $leader\ id$ with the new *id* received in the case that the received *id* is less than the current $leader\ id$. If a change is made to a node's $leader\ id$, it will declare this change to its neighbors by sending messages to them, containing its new $leader\ id$. Two possible algorithms for solving this problem are named LCR and HS. The time order in LCR algorithm is $O(n^2)$. This time order is decreased to $O(nlogn)$ in HS algorithm.

HS Algorithm Each node acts in a set of phases. Node i that is in phase 1, sends a message containing its ID in two directions. These messages pass through a 2^1 length way and then return to the sender. If both of the send messages are returned to the sender, node i will continue acting in the next phase. The sent messages might not get back to the node.

When the message sent by node i moves outwards this node, every node located in its way compares its own leader ID to the ID in the message. If their own leader ID is less than the ID in the message, it will be substituted. If it is greater than it, the message will be ignored. In case they are equal, this means that the node has received its own ID, so the node selects itself as the leader. In the returning way, nothing is done to the message and it just passes through the nodes. The algorithm terminates when a node receives its sent messages from both sides with its own ID, and each message has passed through half of the ring.

LCR Algorithm This algorithm is declared in a directed ring in which the nodes are unaware of the number of the other nodes in the ring. First, each node sends its leader ID - which is equal to its own ID at the beginning - to its right neighbor, and receives a leader ID from its left neighbor. If the received lead ID is greater than its own leader ID, it will substitute its leader ID with the new ID and declares this change to its right neighbor. If the received leader ID is less than a node's leader ID, it will be ignored. In case the received leader ID is equal to the node's leader ID, the node will be considered as the real leader, and the algorithm terminates.

```
activeclass Node(8) {
  knownobjects {
    Node nodeL;
    Node nodeR;
  }
  statevars {
    boolean monitor;
    int myId;
    int phase;
    int monitorId;
    boolean receivedLeft;
    boolean receivedRight;
  }
  msgsrv initial(int id) {
    myId = id;
    monitor = false;
    monitorId = id;
    phase = 1;
    receivedLeft = false;
    receivedRight = false;
    self.arrive();
  }
  msgsrv arrive() {
    nodeL.receive(myId, true, phase);
    nodeR.receive(myId, true, phase);
  }
  msgsrv receive(int msgId, boolean inOut, int
hopCount) {
    if ((sender==nodeL) &&  (inOut)){
      if (((msgId <monitorId)||(msgId==monitorId)) &&
(hopCount >1)){
        monitorId = msgId;
        //temp= hopCount-1;
        nodeR.receive (msgId, true, hopCount-1);
      }
      else {
        if (((msgId <monitorId)||(msgId==monitorId)) &&
(hopCount ==1)){
          monitorId=msgId;
          nodeL.receive (msgId, false,1);
        }
        else{
          if (msgId == myId) {
            monitor = true;
            monitorId = myId;
          }
        }
      }
    }
    if ((sender==nodeR) &&  (inOut)) {
      if (((msgId <monitorId)||(msgId==monitorId)) &&
(hopCount >1)){
        monitorId=msgId;
        //temp= hopCount-1;
        nodeL.receive (msgId, true, hopCount-1);
      }
      else{
        if (((msgId <monitorId)||(msgId==monitorId)) &&
(hopCount ==1)) {
          monitorId=msgId;
          nodeR.receive (msgId, false,1);
        }
        else {
          if (msgId == myId) {
            monitor = true;
            monitorId = myId;
          }
        }
      }
    }
    if ((sender==nodeL) &&  !(inOut) &&
!(msgId==myId)){
      nodeR.receive(msgId, false, 1);
    }
    if ((sender==nodeR) &&  !(inOut) &&
!(msgId==myId)){
      nodeL.receive(msgId, false, 1);
    }
    if ((sender==nodeL) && (msgId == myId) && !(inOut)
&& (hopCount==1)){
      receivedLeft = true;
    }
    if ((sender==nodeR) && (msgId == myId) && !(inOut)
&& (hopCount==1)){
      receivedRight = true;
    }
    if (receivedLeft && receivedRight&& (phase<3)){
      if(phase==2) {
        monitor=true;
      }
      else{
        phase = phase * 2;
        receivedLeft=false;
        receivedRight=false;
        nodeL.receive(myId, true, phase);
        nodeR.receive(myId, true, phase);
      }
    }

  }
}
main {
  Node node1(node4,node2):(1);
  Node node2(node1,node3):(2);
  Node node3(node2,node4):(3);
  Node node4(node3,node1):(4);
}
```

Figure 6.2: Leader Election Example: HS Algorithm

```
activeclass Node(4) {
      knownobjects {
            Node rightNode;
      }
      statevars {
            int id; // my id
            int leaderId;
      }
      msgsrv initial(int myid) {
            id = myid;
            leaderId = myid;
            rightNode.receive(myid);
      }
      msgsrv receive(int lId) {
            if (lId > leaderId) {
                  leaderId = lId;
                  rightNode.receive(lId);
            }
            if (lId == leaderId) {
                  // I am the leader

            }
      }
      msgsrv arrive() {
            leaderId = id;
            rightNode.receive(id);
      }
}

main {
      Node node00(node01):(0);
      Node node01(node02):(1);
      Node node02(node03):(2);
      Node node03(node04):(3);
      Node node04(node05):(4);
      Node node05(node06):(5);
      Node node06(node07):(6);
      Node node07(node00):(7);
}
```

Figure 6.3: Leader Election Example: LCR Algorithm

6.6 CSMA/CD Protocol

In this section, we briefly describe the Media Access Control (MAC) sub layer of the Carrier Sense, Multiple Access with Collision Detection (IEEE 802.3 CSMA/CD) communication protocol. This protocol is used in multiple access shared media environments such as Ethernet LANs, which use a shared bus for connecting a number of independent computers. The protocol specification consists of MAC entities interconnected by a bi-directional Medium. Each MAC is representative of a computer in the data link layer. The MAC entities are identical for all computers and can both transmit and receive messages over the shared Medium. This means that collisions may occur on the Medium (if two MAC's transmit simultaneously). It is assumed that collisions will be detected in the Medium and signaled to every MAC. Each MAC after transmitting a packet over the Medium, waits to make sure that no collision has occurred; but if collision occurs, it tries to retransmit its last packet, until it gets the chance to send the packet successfully without any collision.

As shown in Figure 6.4, a MAC may receive *send* messages from its higher level, indicating a new packet to be sent over the Medium. The MAC cannot process the next packet before it has transmitted the previous packet successfully over the Medium. In the simplified model of the protocol shown in Figure 6.4, the target of a packet is clearly the other MAC present in the composition. Each MAC, similarly, signals a *rec* message to its

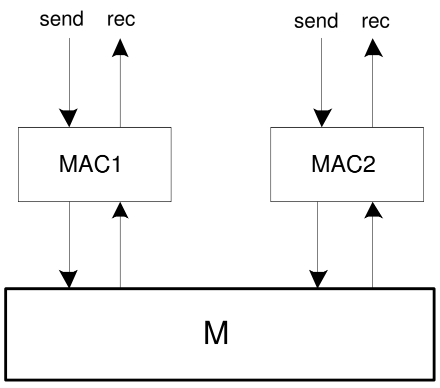


Figure 6.4: The MAC Sublayer of CSMA/CD Protocol

higher level upon successful receipt of a packet from the Medium.

Modeling in Rebeca For modeling this protocol in Rebeca, we defined two active classes: one for the MAC class and another for the Medium class, as shown in Figures 6.6 and 6.7. The role of the components in the higher level is abstracted in our model using a nondeterministic choice in the MAC for deciding when a new packet is available for sending. The other role of this layer, which is receiving the packets, does not change any thing in the model and can easily be ignored.

The composition of our model consists of two instances of the MAC class and one instance of the Medium class. In order to send a packet, each MAC goes through the following scenario, as shown in Figure 6.5. After it has decided to send a packet in the 'start' state, the MAC sends a *b* message to the Medium and enters the 'transfer' state. In

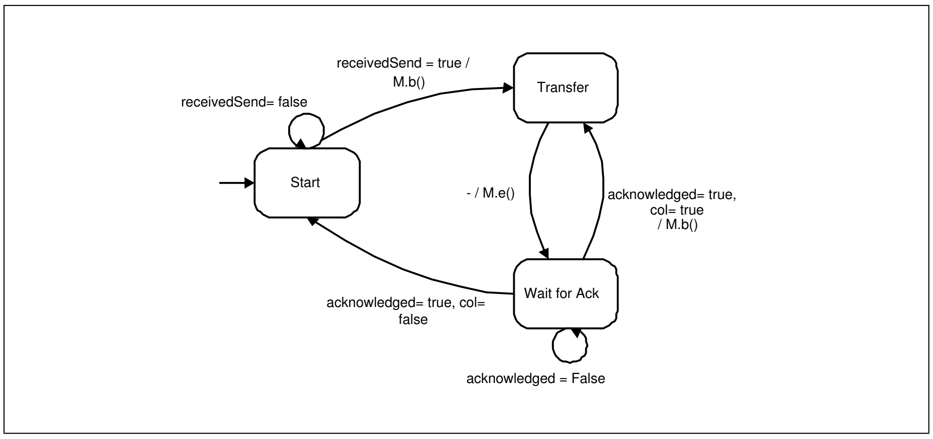

receivedSend = true / M.b()
receivedSend= false
Transfer
Start
- / M.e()
acknowledged= true, col= true / M.b()
Wait for Ack
acknowledged= true, col= false
acknowledged = False

Figure 6.5: State Chart of a MAC Showing the *Send Cycle*

the ’transfer’ state, it sends an *e* message to the Medium, indicating the end of the packet. Then if no collision has occurred, packet transmission is finished and the MAC can get back to the ’start’ state; otherwise, it should retransmit the last packet by sending a new *b* to the Medium and going back to the ’transfer’ state. We name the above cycle, the *Send cycle* of the rebec MAC.

Collision is detected by the Medium if both MACs try to send packets at the same time. However, since we are using asynchronous message passing, collision in our model is defined as the Medium receiving two *b* messages from both MACs before it has received their corresponding *e* messages. This way of modeling collision (the coincidence of the time that two MACs try to send packets) shows how we can model the concept of time using asynchronous message passing.

The important point here is that although the MACs work independently from the Medium, they need to wait for the Medium's response after sending *b* and *e* to make sure whether collision has happened. This is achieved by repeatedly sending the *wait4ack* message to *self* until the acknowledgment from the Medium is received. The Medium on the other hand, needs to wait for the MAC's both *b* and *e* messages to make sure whether collision has happened or not. Therefore, the Medium only after receiving *e* from a MAC can determine if its transmission has been collision-free, and give corresponding acknowledgment.

In order to simplify the model, the receipt of a packet is represented by only one message from the Medium to the receiving MAC, after which the Medium is assumed to be empty and ready for the next packet transmission. This has no effect on the generality of the model; because we can assume that the MAC starts receiving sometime in between receiving *b* and *e* messages from the other MAC, and ends receiving upon receipt of *rec* message, which is sent by Medium immediately after processing the *e* message from the sending MAC. It should be noted that after receiving message *ackRec* from both of the MACs, any *b* from either of them no longer collides with this finished transmission.

When the Medium is processing an *e* message, if no collision has happened, a *collision-false* message can be sent to the sender of the *e* message. On the other hand, which is the case of a collision, the *collisiontrue* message needs to be broadcast to both MACs. In such a case, the Medium surely will receive two *e* messages, because it already has received two *b*s. If we do the broadcast just at the first *e*, we may lose track the *b*s and the next *e* (which should be ignored) may conflict with the next transmission from the MAC that had sent the first *e*.

Verification Results The CSMA/CD protocol (shown in Figures 6.6 and 6.7) is verified using Rebeca Verifier. We used Rebeca Verifier to generate codes in both SMV and Promela. The results of verification of the last version of our model by NuSMV is 1438 reachable states out of 2.2378e+21 total states. In Spin, the max depth is 6603, and the number of stored states is 9184.

In the preliminary versions of our Rebeca model, the number of reachable states in equivalent SMV model exceeds 8 million. Version 6 in Table 6.4 represents one of these versions. The number of reachable states, the CPU time for computing these states, and also the memory used in this computation are shown. Table 6.4 shows the results of executing NuSMV on a Pentium IV 2.00 GHz (full cache) system with 1.0 GB RAM.

Existence of redundant message servers in the MACs, although correct, results in an excessive increase in the number of states. This is caused by the fact that a rebec needs to

```
activeclass Mac(3) {
    knownobjects {
     Medium medium; }
    statevars {
        boolean receivedSend;
        boolean col;
        boolean acknowledged; }
    msgsrv initial() {
        acknowledged = false;
        receivedSend = ?{true, false};
        col=false;
        self.start();
    }
    msgsrv rec(){
        medium.ackRec();
    }
    msgsrv start (){
        if (receivedSend){
            receivedSend=false;
            medium.b();
            self.transfer();
        }
        else {
            receivedSend = ?{true, false};
            self.start();
        }
    }
    msgsrv transfer(){
        acknowledged = false;
        medium.e();
        self.wait4ack();
    }
    msgsrv wait4ack (){
        if (acknowledged) {
            acknowledged = false;
            if (col){
                medium.b(); /* retransmit */
                self.transfer();
            }
            else{
                receivedSend = ?{true, false};
                self.start();
            }
        }
        else {
            self.wait4ack();
        }
    }
    msgsrv collisiontrue(){
        col = true;
        acknowledged = true;
    }
    msgsrv collisionfalse(){
        col = false;
        acknowledged = true;
    }
 }
```

Figure 6.6: Rebeca Code for MAC

```
activeclass Medium(5) {
   knownobjects  {
       Mac mac1; Mac mac2; }
   statevars {
       boolean bb1; boolean bb2;
       boolean r1;  boolean r2;
       boolean col; }
   msgsrv initial() {
       bb1=false; bb2=false;
       col = true;  }
   msgsrv b() {
       if (sender == mac1){
           bb1 = true;  }
       else{
           bb2 = true;
     } }
   msgsrv e() {
       if (sender == mac1) {
           if (!bb2 && bb1){
               mac2.rec();
               self.ackReceive1();
               bb1 = false;
               col = false;  }   }
       else {
           if (bb1){
               mac1.collisiontrue();
               mac2.collisiontrue();
               bb1 = false;
               col = true;   }
           else{
               mac1.rec();
               self.ackReceive2();
               col = false;
           }
           bb2=false;
   }   }
   msgsrv ackReceive1(){
       if (!r2){
           self.ackReceive1(); }
       else{
           mac1.collisionfalse();
           r2 = false;
   }   }
   msgsrv ackReceive2(){
       if (!r1){
           self.ackReceive2(); }
       else{
           mac2.collisionfalse();
           r1 = false;
  }    }
   msgsrv ackRec(){
       if (sender == mac1){
           r1 = true; }
       else{
           r2 = true;
   }   }
}
```

Figure 6.7: Rebeca Code for Medium

Table 6.4: CSMA/CD Versions Compared using NuSMV

Version	States	Compute time	Memory (KB)
6	8×10^6	$00:23:10$	$972,413$
8	2×10^6	$00:05:23$	$118,016$
9.5	1438	$00:00:00$	$14,292$
9.6	951	$00:00:00$	$13,384$

send a message to itself in order to make a transition from one state to another. Therefore, arrival of a message between each two state transitions can cause a virtual new state. It increases the state space proportional to the number of steps in the life cycle of the rebec. Removing redundant message servers results in version 8 in Table 6.4.

In these versions we also have queue overflow. This is due to the logical unfairness in the execution of MAC instances. One MAC may infinitely send packets. Consequently the Medium puts *rec* messages in the queue of the other MAC. As long as the sender MAC gets more turns than the receiver MAC, the number of messages in the queue increases. In order to handle this problem, some kind of logical fairness is introduced in versions 9.5 and 9.6. To ensure that MACs receive incoming packets, acknowledgements are sent, declaring that a MAC has received the last packet; i.e., it finds the chance for execution in the situation explained above.

The safety property, which is verified and proved to be true in the model, is that no collision occurs when one of the MACs receives a packet. For that, we defined a *col* variable in the Medium indicating the collision. The LTL (Linear Temporal Logic) specification of this property is as follows:

$$G((medium.r1 \vee medium.r2) \rightarrow !(medium.col))$$

Version 9.6 is developed in order to check the property that there is a possible computation where although collision happens, the packet is finally received. For this purpose, we simplified the model in the way that only one packet is sent. If collision occurs, the MAC retransmits the packet. The LTL specification of this property is as follows:

$$\begin{aligned}&(mac1.col \wedge mac1.acknowledged)\\ &\qquad\qquad \rightarrow F(medium.r2)\end{aligned}$$

and its symmetric counterpart:

$$\begin{aligned}&(mac2.col \wedge mac2.acknowledged)\\ &\qquad\qquad \rightarrow F(medium.r1)\end{aligned}$$

In global, the other MAC may never receive the packet, as collision may happen forever. So, the following specifications are false:

$$\begin{aligned}&G((mac1.col \wedge mac1.acknowledged)\\ &\qquad\qquad \rightarrow F(medium.r2))\end{aligned}$$

$$G((mac2.col \wedge mac2.acknowledged) \\ \rightarrow F(medium.r1))$$

Chapter 7

Rebecs as Components in a Coordination Language

7.1 Introduction

The Rebeca semantics, as explained in Chapter 2 is not compositional. We cannot construct the semantics of the total model by composing the semantics of each rebec which constructs the model. The compositional verification approach which is discussed in Chapter 3 is based on decomposing a Rebeca model as a closed model and not composing the rebecs as the components of a model.

The possibility of mapping Rebeca models into a coordination language, Reo [13, 15], is investigated and a natural mapping which provides us a compositional semantics of Rebeca is found. As reactive objects (rebecs) are encapsulated and loosely coupled modules in Rebeca, we consider them as components in a coordination language. Modeling the coordination and communication mechanisms between rebecs can be done by Reo circuits, and the behavior of each rebec is specified by constraint automata [14] as a black-box component within the Reo circuit.

7.2 Reo: a Coordination Language

Components-based software development has been proposed by several authors as a solution to the increasing complexity of software development.Components are assumed to be individual and independent units of functionality and deployment and thus to turn them into an application, a mechanism for component composition is needed.

As an important part of component composition mechanism, a piece of connecting code has to be devised in order to match different requirements in a component composition. This piece of code is often referred to as *glue code*. The glue code can range from a simple synchronization and ordering primitive to a complicated distributed coordination protocol. It is often necessary to be able to specify and design these connecting devices and analyze and reason about their behavior individually, as well as in orchestration with (abstract) behavioral models of components.

Reo is a model for building component connectors in a compositional manner. It allows for modeling the behavior of such connectors, formally reasoning about them, and once proven correct, automatically generating the so-called glue code from the specification. Reo's notion of components and connectors is depicted in Figure 7.1, where component instances are represented as boxes, channels as straight lines, and connectors are delineated by dashed lines. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of primitive channels.

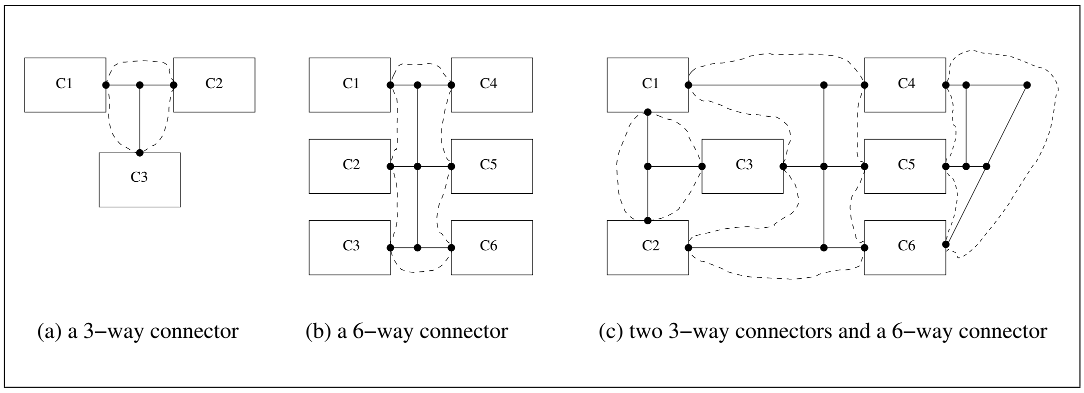


Figure 7.1: Components and Connectors

Reo is a compositional approach to defining component connectors. Reo *connectors* (also called *circuits*) are constructed in the same spirit as logic and electronics circuits: take basic elements (e.g., wires, diodes and transistors) and connect them. Basic connectors in Reo are *channels*. Each channel has exactly two ends, which can be a *sink* end or a *source* end. A *sink* end is where data flows out of a channel, and a *source* end is where data flows in a channel. It is possible that the channel ends of a channel are both sink or both source. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc. But for our purpose to model Rebeca models, we need a small set of basic channels.

Channels are connected to make a circuit. Connecting channels is putting channel ends together in a *node*. So, a *node* is a set of channel ends. A node in Reo has a certain semantics: for all the source channel ends on a node, a fork operation takes place which is copying the outgoing data to all the channel ends; for all the sink channel ends on a node, a merge operation takes place which is a nondeterministic choice between incoming data. A node is called a sink node if it consists of only sink channel ends, it is called a source node if it consists of only source channel ends, and it is called a mixed node if it consists of both sink and source channel ends. Figures 7.2.a and b show sink nodes, Figures 7.2.c and d show source nodes, Figure 7.2.e shows a mixed node. Components can only be connected to sink or source nodes, mixed nodes are hidden from outside world.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data

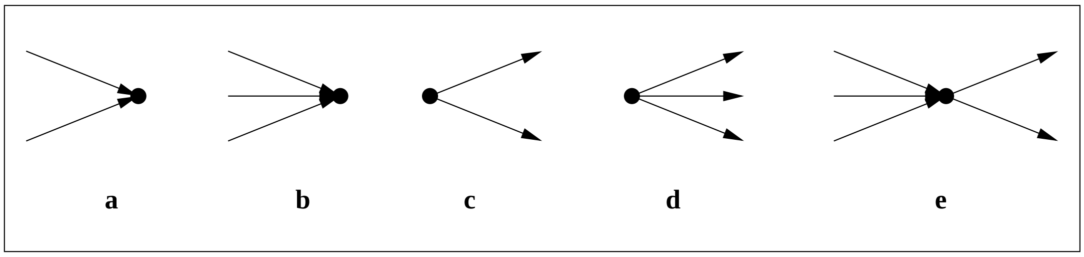


Figure 7.2: Nodes in Reo

item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node is a self-contained "pumping station" that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. A data item is suitable for selection in an iteration only if it can be accepted by all source channel ends that coincide on the mixed node.

Figure 7.3 shows a Reo connector, exclusive router which we call it as *Xrouter*. Here,

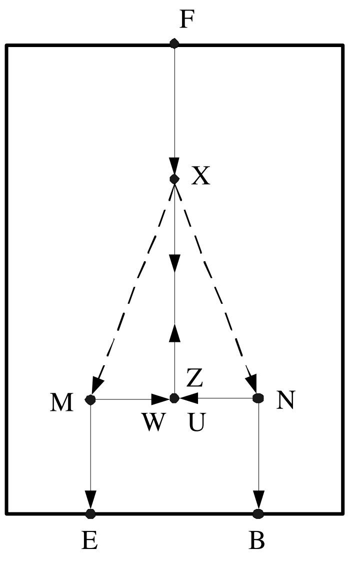


Figure 7.3: Exclusive Router

we use it to show the visual syntax for presenting Reo connector graphs and some frequently useful channel types. This circuit is used in modeling Rebeca by Reo. The enclosing thick box in this figure represents *hiding*: the topologies of the nodes (and their edges) inside the box are hidden and cannot be modified. It yields a connector with a number of input/output *ports*, represented as nodes on the border of the bounding box, which can be used by other entities outside the box to interact with and through the connector.

The simplest channels used in these connectors are synchronous (*Sync*) channels, represented as simple solid arrows. A Sync channel has a source and a sink end, and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A lossy synchronous (*LossySync*) channel is similar to a Sync channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g., there is a take operation pending on its sink) the channel transfers the data item; otherwise the data item is lost. LossySync channels are depicted as dashed arrows, e.g., in Figure 7.3. Another channel is the synchronous drain channel (*SyncDrain*), whose visual symbol appears as the edge XZ in Figure 7.3. A *SyncDrain* channel has two source ends. Because it has no sink end, no data value can ever be obtained from this channel. It accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost.

Two channels which are used in modeling Rebeca but are not included in *Xrouter* circuit, are *Filter* and a special kind of *FIFO* channels. We define *FIFO* as an unbounded asynchronous channel where data can flow in unboundedly from the input port of it (sink node) and flow out of the output port (source node) if it is not empty; input and output operations cannot take place simultaneously. Figure 7.5 in Section 7.4, shows the Reo notation (and the constraint automaton) of an 1-bounded *FIFO* channel. We call the special kind of *FIFO* channel which is used to model Rebeca communications *Queue*, and it is defined in Section 7.5. *Filter* is a channel with a corresponding data pattern, it lets the data matched with the pattern to pass, and lose the other data. *Filter* channel (and its constraint automaton) is shown in Figure 7.6.

7.3 Rebecs as Components in Reo

For modeling Rebeca using Reo, we can consider each rebec as a component, and model the coordination and communication by Reo circuits. For modeling the coordination, an *Xrouter* is used which passes the control to each rebec nondeterministically. Communication takes place by asynchronous message passing which is modeled by queue and filter channels in Reo.

We model each rebec as a black-box component which starts its execution by receiving a *start* signal, and sends an *end* signal upon its end. The behavior of a rebec as a component is to take a message from its message queue upon receiving the *start* signal through its start port, execute the corresponding message server, and send an *end* signal through its end port. The coordination, which is modeled by interleaved execution of rebecs, is handled by *Xrouter* which passes the *start* signal to one and only one rebec, waits until receiving an *end* signal, and passes the *start* signal again. This loop is repeated by *Xrouter*, and sending the signals is done by a nondeterministic choice, which guarantees the execution to be exactly according to the semantics of Rebeca. The Reo circuit in Figure 7.4 shows the *Xrouter*

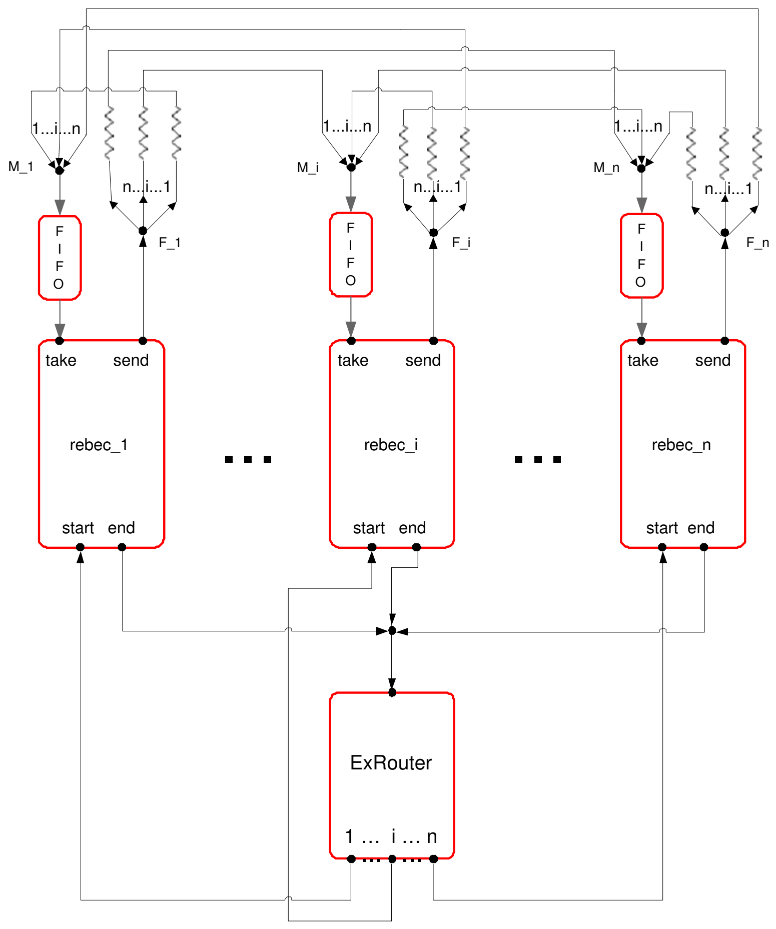


Figure 7.4: Modeling Rebeca by Reo

and other channels which are used to manage the coordination and also communication between rebecs.

For the communication between rebecs, we need queue and filter channels. The message queues of rebecs are modeled by queue channels, each queue models a message queue.

We need to design a circuit which allows only the messages which are sent to the corresponding rebec to get into its queue, and filter out the other messages. In Figure 7.4, there are fork nodes named F_i, and merge nodes named M_i. All the messages that are sent by a rebec $rebec_i$ get out of its port *send*, then pass a *Sync* channel and enters the corresponding fork node F_i. Here, a message is copied into all the source channel ends of the outgoing filter channels. For a model with n rebecs, there are n filter channels, which filter all the messages except those whose receiver is the one matched to that filter channel. The filter pattern for all the channels toward a rebec is the id of that rebec. So, all the filter channels which are merged in the node M_i filter out the messages whose receivers are not $rebec_i$, and only the proper message can pass through the filters and get into the merger node and hence to the message queue (the queue).

Upon receiving a *start* signal, a rebec takes a message from its queue by enabling the *take* port, and then execute the corresponding message server. During this execution, the messages which are sent, flow out of the rebec component through *send* port, and arrive to the message queue of the destination rebec properly, passing the fork node, filter channels, and merge node.

Now, we have a Reo circuit which models a Rebeca model. But, to be able to construct the compositional semantics of a model and verify the properties we need to have a proper semantics for this Reo circuit and also for the rebecs. Constraint automata [14] is presented as a compositional semantics for Reo circuits and can be used to model components and the glue code circuit in a consistent way, and provide us also with verification facilities.

7.4 Constraint Automata: Compositional semantics of Reo

Constraint automata are presented in [14] to model Reo connectors. We also use constraint automata to model the components, then we have a Rebeca model fully modeled by constraint automata. In this section, we explain the definition of constraint automata and how the constraint automata of a Reo circuit is compositionally constructed.

Using constraint automata as an operational model for Reo connectors, the automata-states stand for the possible configurations (e.g., the contents of the FIFO-channels of a Reo-connector) while the automata-transitions represent the possible data flow and its effect on these configurations. The operational semantics for Reo presented in [13] can be reformulated in terms of constraint automata. Constraint automaton of a given Reo connector can also be defined in a *compositional* way. For this, composition operators for constraint automata corresponding to the Reo connector primitives are presented.

Constraint automata use a finite set $\mathcal{N}$ of *names*, e.g., $\mathcal{N} = \{A_1, \ldots, A_n\}$ where A_i stands for the i-th input/output port of a connector or component. The transitions of constraint automata are labeled with pairs consisting of a nonempty subset of $\mathcal{N}$, denoted by N, and a data constraint g. Data constraints can be viewed as a symbolic representation of *sets* of data-assignments. Formally, data constraints are propositional formulae built from the atoms "$d_A = d$" which means that data item d is assigned to port A. Data constraints are given by the following grammar:

$$g \ ::= \ true \ \Big| \ d_A = d \ \Big| \ g_1 \vee g_2 \ \Big| \ \neg g$$

where A is a name and $d \in Data$. In the sequel, $DC(N, Data)$ shows a nonempty subset N of $\mathcal{N}$, and denotes the set of data constraints using only atoms "$d_A = d$" for $A \in N$. As an abbreviation for $DC(\mathcal{N}, Data)$, we can use DC. The boolean connectors $\wedge$ (conjunction), $\oplus$ (exclusive or), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), and so on, can be derived as usual. We often use derived data constraints such as $d_A \neq d$ or $d_A = d_B$ which stand for the data constraints

$$\bigvee_{d' \in Data \setminus \{d\}} (d_A = d') \quad \text{and} \quad \bigvee_{d \in Data} \big((d_A = d) \wedge (d_B = d) \big),$$

respectively.

We assume a global data domain $Data$ for all names. Alternatively, we could assign a data domain $Data_A$ to every name A and require type-consistency in the definition of data constraints.

The assumption that $Data$ is finite allows us to derive data constraints as "$d_A = d_B$" or "$d_A \in D$" or "$(d_A, d_B) \in E$" for $D \subseteq Data$ and $E \subseteq Data \times Data$.

The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting data constraints over data-assignments. For instance,

$$\begin{array}{lcl} [A \mapsto d_1, B \mapsto d_2, C \mapsto d_1] & \models & d_A = d_C, \\ [A \mapsto d_1, B \mapsto d_2, C \mapsto d_1] & \not\models & d_A = d_B \end{array}$$

if $d_1 \neq d_2$. With this satisfaction relation, we may identify any data constraint g with the set δ of all data-assignments where $\delta \models g$ holds.

Satisfiability and validity, logical equivalence $\equiv$ and logical implication $\leq$ of data constraints are defined as usual; e.g.:

$$\begin{array}{lll} g_1 \equiv g_2 & \text{iff} & \text{for all data-assignments } \delta\text{:} \quad \delta \models g_1 \iff \delta \models g_2 \\ g_1 \leq g_2 & \text{iff} & \text{for all data-assignments } \delta\text{:} \quad \delta \models g_1 \implies \delta \models g_2 \end{array}$$

Definition of constraint automata We now present the definition of constraint automata which can serve as operational model for channel-based coordination language, Reo.

Definition 22 *[Constraint automata] A constraint automaton (over the data domain Data) is a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ where*

- *Q is a set of states,*
- *$\mathcal{N}ames$ is a finite set of names,*
- *$\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}ames} \times DC \times Q$, called the transition relation of $\mathcal{A}$,*

- *$Q_0 \subseteq Q$ is the set of initial states.*

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. We call N the name-set and g the guard of the transition. For every transition

$$q \xrightarrow{N,g} p$$

we require that (1) $N \neq \emptyset$ and (2) $g \in DC(N, Data)$. $\mathcal{A}$ is called finite iff Q, $\longrightarrow$ and the underlying data domain $Data$ are finite.

□

The intuitive meaning of a constraint automaton as an operational model for connectors of a coordination language is similar to the interpretation of labelled transition systems as formal models for reactive systems. The states represent the configurations of the connector, the transitions the possible one-step behavior where the meaning of

$$q \xrightarrow{N,g} p$$

is that in configuration q the ports $A_i \in N$ have the possibility to perform I/O-operations that meet the guard g and that lead from configuration q to p, while the other ports $A_j \in \mathcal{N}ames \setminus N$ do not perform any I/O-operation.

Example 9 (*1-bounded FIFO channel*) *Figure 7.5 shows a constraint automaton for a 1-bounded FIFO channel with input port A and output port B. Here, we assume that the data domain consists of two data items* 0 *and* 1.

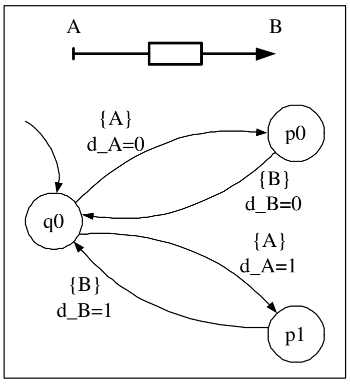


Figure 7.5: Constraint Automaton for a 1-Bounded FIFO Channel

Intuitively, the initial state q_0 stands for the configuration where the buffer is empty, while the states p_0 and p_1 represent the configurations where the buffer is filled with one of the data items. $\Box$

The intuitive behavior of a constraint automaton is that $\mathcal{A}$ starts in one of its initial states q_0. If the current state is q, then $\mathcal{A}$ waits until data items occur at some of the input/output ports $A_i \in \mathcal{N}ames$. Suppose data item d_1 occurs at A_1 and data item d_2 at A_2 while (at this moment) no data is observed at the other ports $A_3,\ldots,A_n$. This triggers the automaton to check the data constraints of the outgoing $\{A_1,A_2\}$-transitions of state q to choose a transition

$$q \xrightarrow{\{A_1,A_2\},g} p$$

where $\left[A_1 \mapsto d_1, A_2 \mapsto d_2\right] \models g$ and move to state p. If there is no $\{A_1,A_2\}$-transition from q whose data constraint is fulfilled then $\mathcal{A}$ rejects. In general, if data occur exactly at the input/output ports $A_i \in N$ then only N-transitions (but no N'-transitions where N' is a subset or superset of N) where the data constraint is fulfilled can fire.

Having this behavior in mind, the intuitive meaning of conditions (1) and (2) in Definition 22 is as follows. Condition (1) stands for the requirement that automata-transitions can fire only if some data occurs at one or more of the ports $A_1, \ldots, A_n$, while condition (2) formalizes that the behavior of an automaton may depend only on its observed data (and not on data that will occur sometime in the future).

We now explain how constraint automata can be used to model the possible data flow of a given Reo circuit. The nodes of a Reo-circuit play the role of the ports in the constraint automata. We provide a *compositional* semantics for Reo circuits. Thus, we need constraint automata for each of the basic channel connectors and automata-operations to mimick the behavior of the Reo-operations for join and hiding.

Constraint automata for the basic channels Figure 7.6 shows the constraint automata for some of the standard basic channel types: synchronous channels with source A and sink B, synchronous drain with the sources A, B, lossy synchronous channels with source A and sink B, and filter with source A and sink B and pattern P. In every case, one single state is sufficient. Moreover, the automata are deterministic.

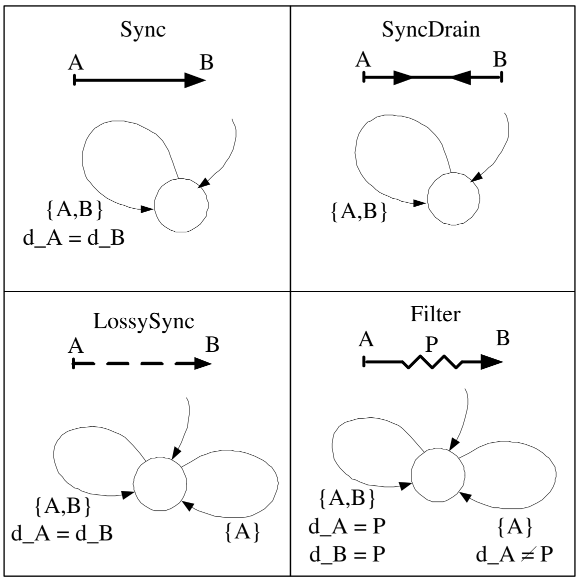


Figure 7.6: Constraint Automata for Basic Connectors

A constraint automaton for the FIFO1 channel was shown in Example 9. For FIFO channels with capacity ≥ 2, similar constraint automata can be used. However, the number of states grows exponentially with the capacity. For instance, for a FIFO2 channel with the data domain $\{0, 1\}$ we need 7 states representing the configurations where the buffer is empty or the buffer contains one element (0 or 1) or is full (00, 01, 10 or 11). For unbounded FIFO channels we even get constraint automata with an infinite state space.

Join: merge and product

Definition 23 *[Product-automaton] The product-automaton of the two constraint automata*

$\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \longrightarrow_2, Q_{0,2})$, is:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 \;=\; (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \longrightarrow, Q_{0,1} \times Q_{0,2})$$

where $\longrightarrow$ is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1,g_1}_1 p_1,\;\; q_2 \xrightarrow{N_2,g_2}_2 p_2,\;\; N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1}{\langle q_1, q_2\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2\rangle}$$

and

$$\frac{q_1 \xrightarrow{N,g}_1 p_1,\;\; N \cap \mathcal{N}ames_2 = \emptyset}{\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, q_2\rangle}$$

and latter's symmetric rule. $\Box$

It remains to explain how the join of two sink nodes, say A and B, is realized with constraint automata. To capture the merge semantics of the resulting (new) node C, we use a *merger* as shown in Figure 7.7[1] which we then join (via the product-operator $\bowtie$) with the constraint automata that contain A and B respectively. We then can again apply the product-construction to join the resulting constraint automaton (that contains C in its name-set) with another constraint automaton that contains C as a source node.

Parameterized Constraint Automata In the previous examples, we concentrated on data-abstract coordination mechanism. In many applications, the data-abstract view is too coarse, e.g., for reasoning about the functionality of the components that are glued together. Because data-dependencies often lead to rather complex constraint automata, we propose

[1]In a similar way, a merger can be defined as a connector with three or more "input" nodes.

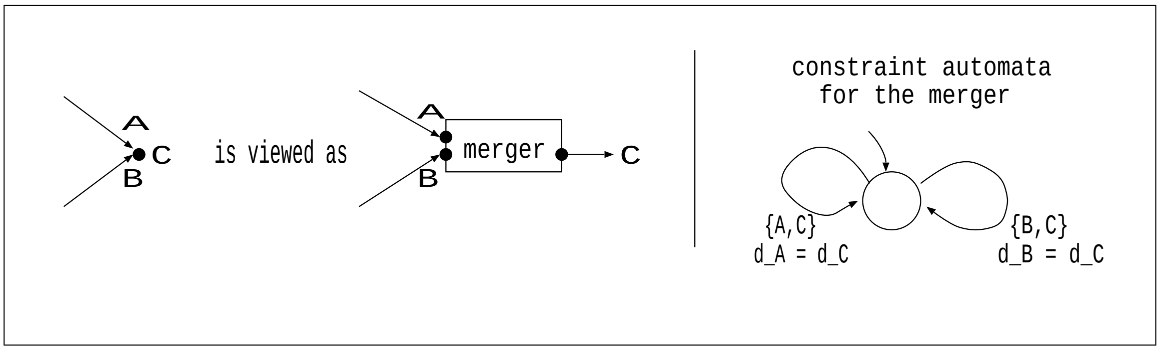


Figure 7.7: The Merger

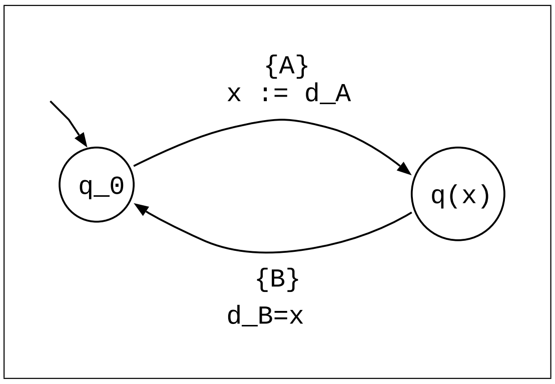


Figure 7.8: Parameterized Constraint Automaton for a 1-Bounded FIFO Channel

a parameterized notation which can simplify the picture of constraint automata with non-trivial guards. For instance, the 1-bounded FIFO channel with arbitrary data domain can be depicted as in Figure 7.8.

The automaton in Figure 7.8 is *not* a constraint automaton, but an intuitive symbolic representation of the constraint automaton with state-space $Q = \{q_0\} \cup \{q(d) : d \in Data\}$,

$Q_0 = \{q_0\}$, $\mathcal{N}ames = \{A, B\}$ and the transitions

$$q_0 \xrightarrow{\{A\}, d_A = d} q(d), \quad q(d) \xrightarrow{\{B\}, d_B = d} q_0$$

for any data item $d \in Data$. Formally, to reason about data-dependent coordination mechanisms, we define a *parameterized constraint automaton* as a tuple

$$\mathcal{P} = (Loc, Var, v, \mathcal{N}ames, \rightsquigarrow, Loc_0, init)$$

where

- Loc is a set of locations,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- Var a set of variables,
- $v : Loc \rightarrow 2^{Var}$ assigns to any location ℓ a (possibly empty) set of variables,
- $init$ is a function that assigns to any initial location $\ell \in Loc_0$ a condition for the variables.

$v(\ell)$ can be viewed as the parameter list of location ℓ. For instance, in Figure 7.8 we use $q(x)$ to denote that q is a location with parameter list $v(q) = \{x\}$, while q_0 is a location with an empty parameter list. The initial condition for q_0 is omitted which denotes that $init(q_0) = true$.

The transition relation $\rightsquigarrow$ of a parameterized constraint automaton is a (finite) set of tuples (ℓ, N, h, X, ℓ'), written in the form

$$\ell \overset{N,h}{\rightsquigarrow}_X \overline{\ell}.$$

Here,

- ℓ and $\overline{\ell}$ are locations,
- N is a non-empty name-set,
- h a (parameterized) data constraint for N, built out of atoms of the form "$d_A = expr$". The expression $expr$ is built from constants $d \in Data$, the symbols d_B for $B \in N$, variables $x \in v(\ell)$ and operators for the chosen data domain, e.g., boolean operator $\vee$, $\wedge$, etc. for $Data = \{0,1\}$ and arithmetic operators $+$, $*$, etc. for $Data = \mathbb{N}$.
- The subscript X of the above transition stands for a function that assigns a name $A \in N$ to each variable $\overline{x} \in v(\overline{\ell}) \setminus v(\ell)$ and possibly to some of the variables in $v(\overline{\ell}) \cap v(\ell)$. The intuitive meaning of $X(\overline{x}) = A$ is the assignment "$\overline{x} := d_A$".

We use parameterized constraint automata as a symbolic representation of (non-parameterized) constraint automata. The states of the latter are obtained by augmenting the locations with values for the variables of their parameter list. Formally, given $\mathcal{P}$ as above, the induced constraint automaton $\mathcal{A}_{\mathcal{P}} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ is defined as follows. The state-space Q of$\mathcal{A}_{\mathcal{P}}$ consists of the pairs $\langle \ell, \eta \rangle$ where $\ell \in Loc$ is a location and η a variable evaluation for the variables $x \in v(\ell)$, i.e., η is a function from $v(\ell)$ to $Data$. The states $\langle \ell, \eta \rangle$ with $\ell \in Loc_0$ and $\eta \models init(\ell)$ are the initial states of $\mathcal{A}_{\mathcal{P}}$. The transition relation $\longrightarrow$ is derived from $\rightsquigarrow$ by the following rule:

$$\frac{\ell \overset{N,h}{\rightsquigarrow}_X \overline{\ell}, \;\; \overline{\eta} = \eta[X,\delta]|_{v(\overline{\ell})}, \;\; g = h[x/\eta(x) : x \in v(\ell)] \wedge g[\delta]}{\langle \ell, \eta \rangle \xrightarrow{N,g} \langle \overline{\ell}, \overline{\eta} \rangle}$$

where $\delta = [A \mapsto \delta_A : A \in N_X]$ is an arbitrary data assignment for N_X, the set of names $A \in N$ where X contains an assignment "$\overline{x} := d_A$" and $g[\delta]$ is the data constraint

$$g[\delta] \; = \; \bigwedge_{A \in N_X} (d_A = \delta_A).$$

The construct $h[x/\eta(x)]$ stands for the data constraint obtained from h by syntactically replacing variable x with the value $\eta(x) \in Data$. The construct $\eta[X,\delta]$ denotes the evaluation for the variables in $v(\ell) \cup v(\overline{\ell})$ that is obtained from η by executing the assignments of X. For instance,

$$\eta[\underbrace{\overline{x} := d_A}_{X}, \underbrace{A \mapsto d}_{\delta}](y) \; = \; \begin{cases} \eta(y) & : \text{ if } y \in v(\ell) \setminus \{\overline{x}\} \\ d & : \text{ if } y = \overline{x}. \end{cases}$$

The construct $\eta[X,\delta]|_{v(\overline{\ell})}$ denotes the restriction of $\eta[X,\delta]$ to the variables in $v(\overline{\ell})$.

Note that constraint automata are special instances of their parameterized version with empty parameter lists for all their locations. (In this case, there is no difference between locations and states, and we have $\mathcal{A}_{\mathcal{A}} = \mathcal{A}$.)

The product construction (Definition 23) can easily be modified for parameterized constraint automata $\mathcal{P}_1$ and $\mathcal{P}_2$ with disjoint variable sets such that the unfolding of the product $\mathcal{P}_1 \bowtie \mathcal{P}_2$ into a (non-parameterized) constraint automaton $\mathcal{A}_{\mathcal{P}_1 \bowtie \mathcal{P}_2}$ generates the same TDS-language as the product $\mathcal{A}_{\mathcal{P}_1} \bowtie \mathcal{A}_{\mathcal{P}_2}$ of the constraint automata for $\mathcal{P}_1$ and $\mathcal{P}_2$.

7.5 Compositional Semantics of Rebeca using Constraint Automata

To obtain the constraint automata of the coordination and communication parts of the Rebeca model, which is modeled in Reo, we use the algorithm in Section 7.4. For specifying the semantics of rebecs we need parameterized constraint automata. To obtain the parameterized constraint automaton (PCA) of each rebec, we use an algorithm, shown in Figure 7.9, to extract the PCA directly from the Rebeca code.

In the parameterized constraint automaton for each rebec,

$$\mathcal{P}_i = (Loc_i, Var_i, v_i, \mathcal{N}ames_i, \rightsquigarrow_i, Loc_{0i}, init_i)$$

for all the rebecs, we have $\mathcal{N}ames_i = \{start, end, send, take\}$, and $Loc_{0i} = \{idle\}$. For each rebec Var_i includes state variables of the rebec, local variables of each method, and *sender*

```
Variables: sender, state variables of the rebec, local variables
of each method

addState('Idle') addState('Take Message') addTransition('Idle' to
'Take Message', N={'start'}) addTransition('Take Message' to
'Idle', N={'take', 'end'}, d_take(2)='empty')

for each message server do begin
   addTransition('Take Message' to addState(i), N={'take'}, d_take(2)= MessageServerId)
   sender := d_take(1)
   Determine different flows of control in the message server, according to different conditions
   Flag each statement with its correspondent condition
   devide each flow of control into fragments:
          each fragment ends with a 'send' statement (or the end of the message server)
          for each fragment do
          begin
             addstate(i)
             addTransition(previousstateInFragment to addState(i), N={'send'},
                      d_send= (receiver, requestedmessageName),
                      other data constraints: Condition of the fragment,
                      data assignments: assignment statements in that fragment)

             (for the last fragment:
              addTransition(previousstateInFragment to 'Idle', N={'send','end'}
                      d_send= (receiver, requestedmessageName),
                      other data constraints: Condition of the fragment,
                      data assignments: assignment statements in that fragment)
              )
           end
end
```

Figure 7.9: The Algorithm for Constructing the Parameterized Constraint Automaton from a Rebec Code

variable which gets the value of the sender of each message.

The initial state of the PCA (Parameterized Constraint Automaton) of each rebec is denoted as *idle* state. At the beginning all the rebecs are in their *idle* state. By getting the *start* signal as input from the *Xrouter*, a rebec moves to its *Take Message* state, where a message is taken from top of the queue. The data item of the port *take* is assumed to be a tuple consisting of *sender* of the message and the *message server name*. According to the *d_take*, the next state is chosen. If the message queue is empty the transition goes back to the *idle* state. If not, the transition goes to the state which is the beginning of the execution of a message server. In fact, the second item of *d_take* which is the *message server name* specifies the next state. The rest of the work depends on the statements of the message servers; at the end of each message server there shall be a transition back to the *idle* state which has the *end* signal (and maybe a *send* signal) on it. We use an example, the bridge controller, to explain the rest of the algorithm in more details in the next section.

As we mentioned before, we use a special kind of FIFO channel for modeling the message queue. The main point is that we want to be able to realize the situation when the queue is empty, this cannot be done with the conventional definitions of FIFO in Reo [15, 13], so we assume that there is a special data denoted by *empty* which shows that the queue is empty. We define the behavior of message queue channel according to the constraint

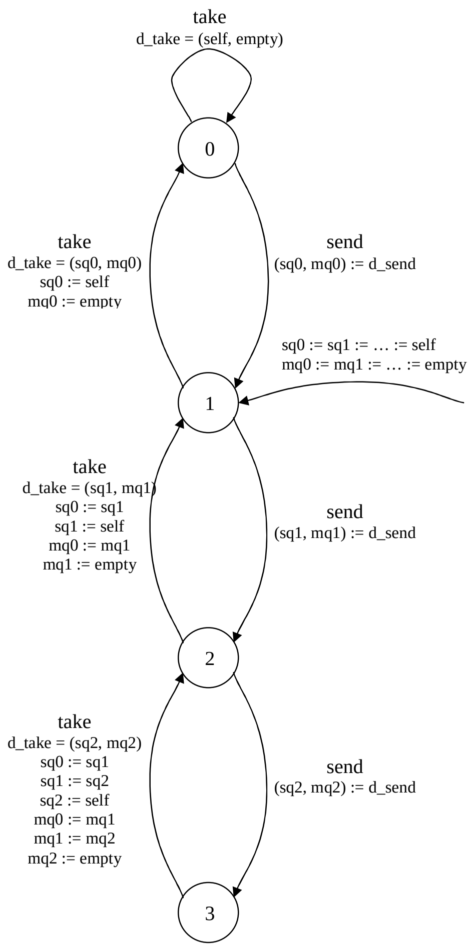


Figure 7.10: Constraint Automaton for the 4-Bounded Message Queue Channel

automaton shown in Figure 7.10. In this figure we show the behavior of a bounded queue.

7.6 An Example: Bridge Controller

The bridge controller is chosen as an example to be modeled by constraint automata. This example is described in Section 5.6, and the Rebeca code is shown in Figure 5.4. There is 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.

Figure 7.11 shows the constraint automata for each train and Figure 7.12 shows the constraint automata for the bridge controller. The initial state for a train is *idle* state. We move to the *take message* state by receiving the *start* signal. A train has four message servers: *initial*, *YouMayPass*, *Passed*, and *ReachBridge*. For each one of these message servers there is an outgoing transition from the *take message* state. There is also another transition which is fired when the message queue is empty. The four first transitions, each goes to a state showing the beginning of a message server. The last one goes back to the *idle* state outputting the *end* signal.

As described in the algorithm of Figure 7.9, we have to consider the different flows of control in each message server. In the message servers of the trains we only have one flow of control. We partition each flow by the *send* statements. For example in the message server *Passed* we have two fragments. You can see two transitions corresponding to the *send* statements in Figure 7.11. The *end* signal is added to the last transitions, which can be considered as an optimization issue. Considering the controller, we have conditional statements in message servers *Arrive* and *Leave*, and hence more than one possible flow of control. The transitions generated for different flows of controls can be seen in Figure 7.12.

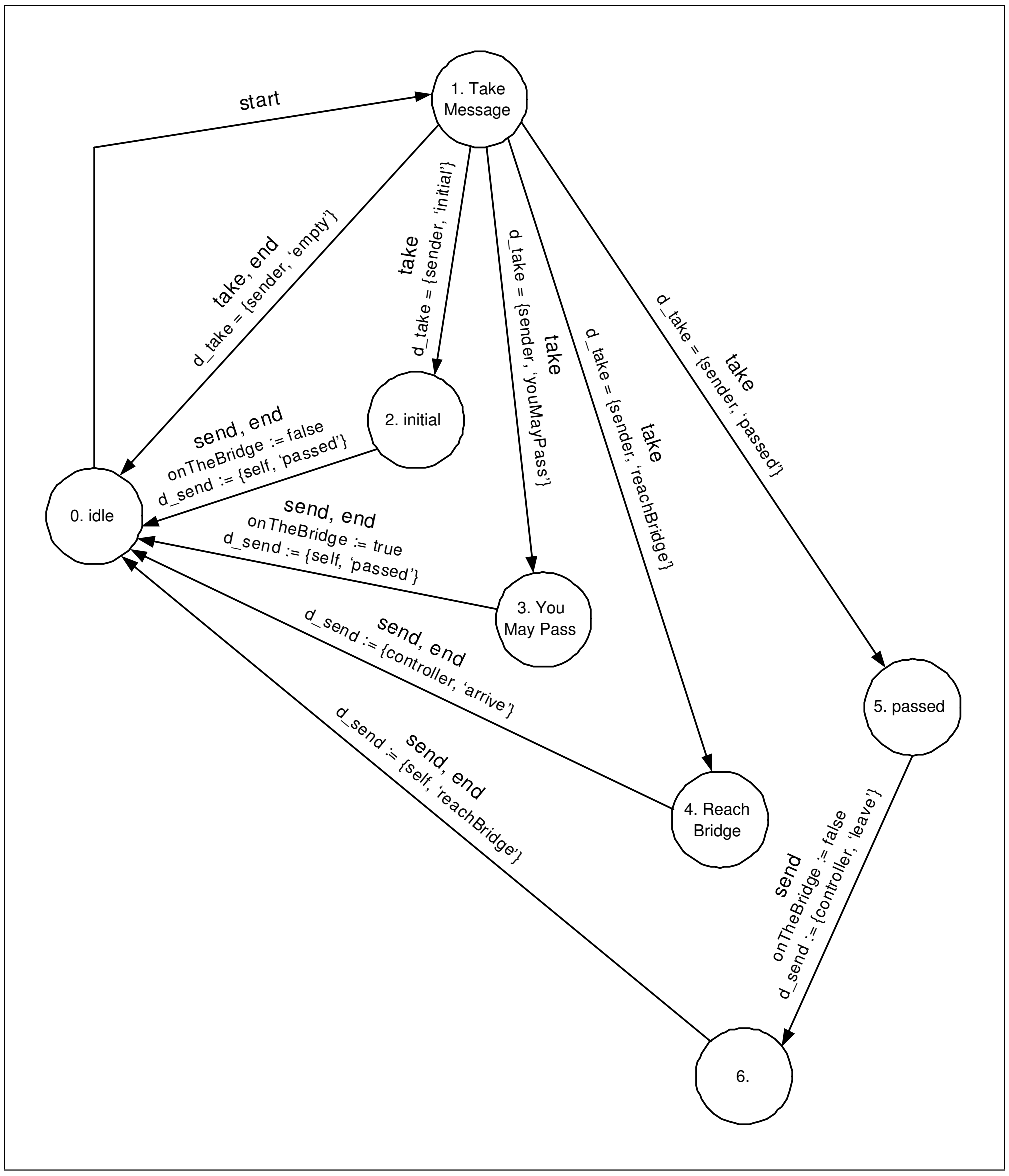

start
1. Take Message
take, end
d_take = {sender, 'empty'}
take
d_take = {sender, 'initial'}
take
d_take = {sender, 'youMayPass'}
take
d_take = {sender, 'reachBridge'}
take
d_take = {sender, 'passed'}
2. initial
send, end
onTheBridge := false
d_send := {self, 'passed'}
0. idle
send, end
onTheBridge := true
d_send := {self, 'passed'}
3. You May Pass
send, end
d_send := {controller, 'arrive'}
5. passed
4. Reach Bridge
send, end
d_send := {self, 'reachBridge'}
send
onTheBridge := false
d_send := {controller, 'leave'}
6.

Figure 7.11: Modeling Train by Constraint Automata

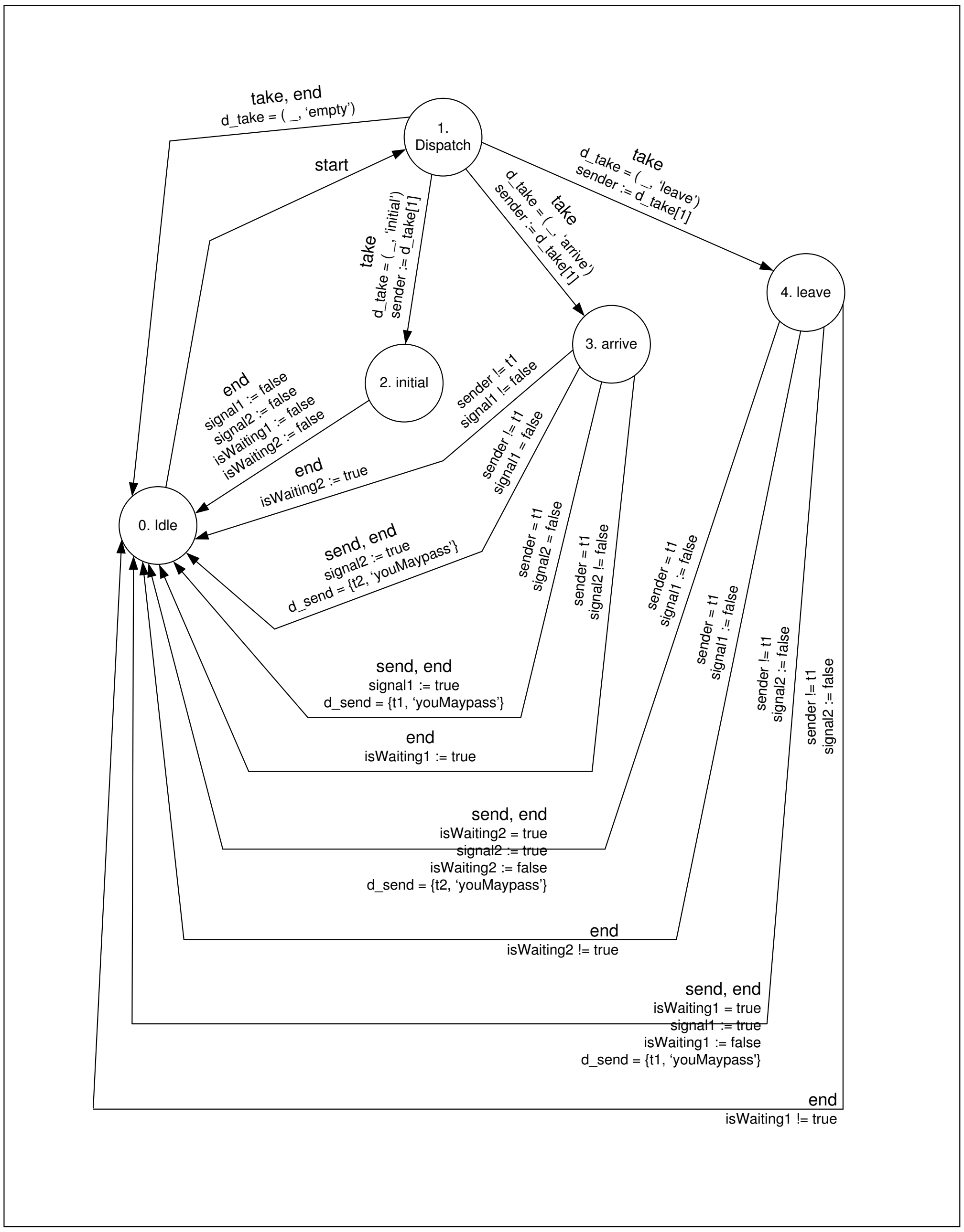


Figure 7.12: Modeling Controller by Constraint Automata

Chapter 8

Conclusion and Future Work

Actor-based modeling can help the modeler via its encapsulated constructs, and formal verification can be used to design more dependable systems. Compositional verification seems to be a sound way to make formal verification practical, but it can help best when the model is modular and the modules are encapsulated and loosely coupled. That is where the modular nature of actor-based modeling may help in formal verification.

In Rebeca, we have independent reactive objects called *rebec*s which run concurrently and communicate by asynchronous message passing. There is an unbounded message queue for each rebec. We have classes for declaring the rebecs in the model. Therefore it is possible to reuse the code and simplify the verification process. A nonempty set of rebecs, referred to as a component, may be used to represent a reactive system.

A system can be decomposed into components that are executed concurrently. We can first verify properties of these components, specified in LTL-X or ACTL by model checking, and then conclude the overall system property using these latter results. Composition of two components is a simple operation due to independence of rebecs in Rebeca. The result is another component, while no conditions on composing components are required.

We use abstraction and symmetry to tackle state explosion problem. The asynchronous nature of message passing in Rebeca, let us to use coarse-grained transitions which reduce the state space and make the model simpler. Abstracting from message queues in specifying system properties introduces some kind of abstraction. In our compositional approach, we model the environment only by external messages, and the abstraction is to have these messages in a set and not to put them in the queues. In Rebeca we do not have any restrictions for components in parallel composition. Also, we do not need any assumptions about the environment, in the properties that we prove. We also use symmetry to simplify our verification process when there are replicated components in the system.

We have enriched the modeling power of the basic message-driven, asynchronous computational model of the actor-based language Rebeca by introducing a formal concept of components for structuring a model in Rebeca and to integrate asynchrony by synchronous message passing. We exploited the additional structuring mechanisms, provided by components, in a compositional verification approach based on model-checking. Formal semantics of extended Rebeca is used to establish the verification theory corresponding this approach.

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.

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.

Future Work We defined our compositional verification approach on the simplified version of Rebeca ignoring dynamic behavior. With some restrictions on defining components we can use our compositional approach even in presence of dynamic creation and topology. In presence of dynamic changing topology, we need to talk about variables of known rebecs of a rebec. In presence of dynamic creation, sometimes it is needed to check the value of a specific variable for all instances of a class. Therefore, we need quantification over rebecs and state formulas are predicates instead of propositions. Determining this subject more precisely is one of our future works.

Our research group in Tehran and Sharif universities is working on the Rebeca Verifier tool. Currently we are working on extending our tool to support model checking and compositional verification of extended Rebeca. Another team is working on translating Rebeca to Java programs. This will give us a refinement tool which can be another step towards building a formal methodology for reliable software development.

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. Data abstraction in model checking Rebeca codes is now based on the back-end model checker approaches. We provide the same data types as in SMV and Promela. In our future work, for direct model checking of Rebeca codes we also need to consider the abstract interpretation of supported data types.

Furthermore, we used Rebeca for modeling security protocols, using dynamic data structures to describe the behavior of intruders [32]. For model checking these applications, we therefore need appropriate abstraction techniques. Mitnick attack is also modeled in Rebeca to show how an attacker may chain simple attacks to construct a complex distributed attack.

The additional synchronous communication of messages increases the modeling power and also serves as a formal semantic basis for modeling languages like UML. UML integrates an asynchronous event driven model of computation, like that of the actor languages, with a synchronous model of computation described by state charts. However, a comprehensive formal account of the intricacies involved in the interplay between synchrony and asynchrony in UML is still missing. Currently we are investigating the formal relationship between a subset of UML developed in European IST Project Omega [20] and our extended Rebeca. This line of research can be seen as a first step to a formal account of the integration of synchrony and asynchrony in UML.

Bibliography

[1] *NuSMV user manual,* availabe through http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html.

[2] *Rebeca*, http://khorshid.ut.ac.ir/∼rebeca.

[3] *Slam project*, available through http://research.microsoft.com/slam/.

[4] *Spin user manual,* available through http://netlib.bell-labs.com/netlib/spin/whatisspin.html.

[5] M. Abadi and L. Lamport, *Conjoining Specifications,* ACM Transactions on Programming Languages and Systems **17** (1995), no. 3, 507–534.

[6] G. Agha, *Actors: A model of concurrent computation in distributed systems,* MIT Press, Cambridge, MA, USA, 1990.

[7] G. Agha, *The structure and semantics of actor languages,* Foundations of Object-Oriented Languages (J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds.), Springer-Verlag, Berlin, Germany, 1990, pp. 1–59.

[8] G. Agha, I. Mason, S. Smith, and C. Talcott, *A foundation for actor computation*, Journal of Functional Programming **7** (1997), 1–72.

[9] R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang, *Automating modular verification*, CONCUR: 10th International Conference on Concurrency Theory, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1999, pp. 82–97.

[10] R. Alur, T. A. Henzinger, F. Y. C. Mang, and S. Qadeer, *MOCHA: Modularity in model checking*, Proceedings of CAV'98, vol. 1427, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 521–525.

[11] R. Alur and T.A. Henzinger, *Computer aided verification*, Tech. Report Draft, 1999.

[12] R. Alur and T.A. Henzinger, *Reactive Modules*, Formal Methods in System Design: An International Journal **15** (1999), no. 1, 7–48.

[13] F. Arbab, *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science (2004), To appear in February 2004.

[14] Farhad Arbab, Christel Baier, Jan J.M.M. Rutten, and Marjan Sirjani, *Modeling component connectors in Reo by constraint automata*, Proceedings of Second International Workshop on Foundations of Coordination Languages and Software Architectures (FLOCASA'03), to appear, 2003.

[15] Farhad Arbab and Jan J.M.M. Rutten, *A coinductive calculus of component connectors*, Tech. Report SEN-R0216, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 2002.

[16] J. Bakker, J. Kok, and J. Rutten, *Operational semantics of a parallel object-oriented language*, Conference Record of the 13th Symposium on Principles of Programming Languages.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*, The MIT Press, Cambridge, Massachusetts, 1999.

[18] E. M. Clarke, D. E. Long, and K. L. McMillan, *Compositional model checking*, Proceedings, Fourth Annual Symposium on Logic in Computer Science (Asilomar Conference Center, Pacific Grove, California), IEEE Computer Society Press, 5–8 June 1989, pp. 353–362.

[19] Beyer D., Lewerentz C., and Noack A., *Rabbit: A tool for BDD-based verification of real-time systems*, Proceedings of CAV 2003 (Hunt W.A., Jr. Somenzi, and F. Somenzi, eds.), Lecture Notes in Computer Science, vol. 2725, Springer-Verlag, Berlin, Germany, 2003, pp. 122–125.

[20] W. Damm, B. Josko, A. Pnueli, and A. Votintseva, *Understanding UML: A formal semantics of concurrency and communication in real-time UML*, Proceedings of Formal Methods for Components and Objects (Leiden, The Netherlands), Lecture Notes

in Computer Science, vol. 2852, Springer-Verlag, Berlin, Germany, pp. 71–98.

[21] F. S. de Boer, *A proof system for the language pool*, Foundations of Object-Oriented Languages (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), Springer-Verlag, Berlin, Heidelberg, 1991, pp. 124–150.

[22] W. P. de Roever, h. Langmaack, and A. Pnueli (eds.), *Compositionality: The significant difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures*, Lecture Notes in Computer Science, vol. 1536, Springer-Verlag, Berlin, Germany, 1998.

[23] 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*, Proceedings of the 23nd International Conference on Software Engineering, 2001, pp. 177–187.

[24] E.A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science (Amsterdam) (J. van Leeuwen, ed.), vol. B, Elsevier Science Publishers, 1990, pp. 996–1072.

[25] E. Gagnon and L. Hendren, *SableCC – an object-oriented compiler framework*, Proceedings of TOOLS 1998, Springer-Verlag, Berlin, 1998, pp. 140–154.

[26] M. Gaspari and G. Zavattaro, *An actor algebra for specifying distributed systems: The hurried philosophers case study*, Lecture Notes in Computer Science **2001** (2001), 216–246.

[27] K. Havelund and T. Pressburger, *Model checking Java programs using Java PathFinder*, International Journal on Software Tools for Technology Transfer **2** (2000), no. 4, 366–381.

[28] 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.

[29] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[30] M.R. Huth and M. Ryan, *Logic in computer science: Modelling and reasoning about systems*, Cambridge University Press, 2002.

[31] N. Ioustinova, N. Sidorova, and M. Steffen, *Closing open SDL-systems for model checking with DTSpin*, FME’2002, Lecture Notes in Computer Science, vol. 2391, Springer-Verlag, Berlin, Germany, 2002, pp. 531–548.

[32] H. Iravanchi, M. Sirjani, and F. de Boer, *Modeling and verifying security protocols using Rebeca*, Tech. Report to appear, CWI, Amsterdam, The Netherlands, 2003.

[33] Y. Kesten and A. Pnueli, *Modularization and abstraction: The keys to practical formal verification*, Proceedings of MFCS-98, vol. 1450, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1998, pp. 54–71.

[34] O. Kupferman, M. Y. Vardi, and P. Wolper, *Module checking*, Information and Computation **164** (2001), no. 2, 322–344.

[35] Abadi L. and Lamport L., *Composing specifications*, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada (H. V. Jagadish and Inderpal Singh Mumick, eds.), ACM Press, New York, USA, 1996, 1996, pp. 365–376.

[36] L. Lamport, *Composition: A way to make proofs harder*, Proceedings of COMPOS: International Symposium on Compositionality: The Significant Difference, Lecture Notes in Computer Science, vol. 1536, Springer-Verlag, Berlin, Germany, 1997, pp. 402–407.

[37] K. G. Larsen and R. Milner, *A compositional protocol verification using relativized bisimulation*, Information and Computation **99** (1992), no. 1, 80–108.

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

[39] N.A Lynch, *Distributed algorithms*, Morgan Kaufmann, San Francisco, CS, 1996 (English).

[40] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag, Berlin, Germany, 1992.

[41] Z. Manna and A. Pnueli, *Temporal verification of reactive systems (safety)*, Springer-Verlag, Berlin, Germany, 1995.

[42] I. A. Mason and C. L. Talcott, *Actor languages: Their syntax, semantics, translation, and equivalence*, Theoretical Computer Science **220** (1999), no. 2, 409–467.

[43] K. McMillan, *Verification of digital and hybrid systems*, Springer-Verlag, Berlin, Germany, 2000.

[44] K. L. McMillan, *A methodology for hardware verification using compositional model checking*, Science of Computer Programming **37** (2000), no. 1–3, 279–309.

[45] R. Milner, *A Calculus on Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlin, Germany, 1980.

[46] R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes*, Information and Computation **100** (1992), no. 1, 1–77.

[47] J. Parrow, *Verifying a CSMA/CD-protocol with CCS*, Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification (Atlantic City, New Jersey), North-Holland, 1988, pp. 373–387.

[48] S. Ren and G. Agha, *RTsynchronizer: language support for real-time specifications in distributed systems*, ACM SIGPLAN Notices **30** (1995), no. 11, 50–59.

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

[50] S. Schacht, *Formal reasoning about actor programs using temporal logic*, Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science, vol. 2001, Springer-Verlag, Berlin, Germany, 2001, pp. 445–460.

[51] N. Sidorova and M. Steffen, *Embedding chaos*, Proceedings of Static Analysis Symposium (SAS01), Lecture Notes in Computer Science, vol. 2126, Springer-Verlag, Berlin, Germany, 2001, pp. 319–334.

[52] M. Sirjani and F. de Boer, *Modular verification of components in an actor-based language extended with synchronous communication*, Tech. Report to appear, CWI, Amsterdam, The Netherlands, 2004.

[53] M. Sirjani and A. Movaghar, *An actor-based model for formal modelling of reactive systems: Rebeca*, Tech. Report CS-TR-80-01, Tehran, Iran, 2001.

[54] M. Sirjani, A. Movaghar, H. Iravanchi, M. Jaghoori, and A. Shali, *Model checking in Rebeca*, Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'03), CSREA Press, USA, 2003, June 2003, pp. 1819–1822.

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

[56] M. Sirjani, A. Movaghar, and M.R. Mousavi, *Compositional verification of an object-based reactive system*, Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01) (Oxford, UK), April 2001, pp. 114–118.

[57] 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*, Proceedings of ACSD 2004.

[58] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen, *Divide, abstract and model-check*, Proceedings of the 5th International SPIN Workshop on Theoretical Aspects of Model Checking, Lecture Notes in Computer Science, vol. 1680, Springer-Verlag, Berlin, Germany, 1999.

[59] C. Talcott, *Composable semantic models for actor theories*, Higher-Order and Symbolic Computation **11** (1998), no. 3, 281–343.

[60] C. Talcott, *Actor theories in rewriting logic*, Theoretical Computer Science **285** (2002), no. 2, 441–485.

[61] Y. Tsay, *Compositional verification in linear-time temporal logic*, Proceedings of FOSSACS 2000, Lecture Notes in Computer Science, vol. 1784, Springer-Verlag, Berlin, Germany, 2000, pp. 344–358.

[62] M. Y. Vardi, *Verification of open systems*, Lecture Notes in Computer Science **1346** (1997), 250–267.

[63] C. Varela and G. Agha, *Programming dynamically reconfigurable open systems with SALSA*, ACM SIGPLAN Notices **36** (2001), no. 12, 20–34.

[64] A. Yonezawa, *ABCL: An object-oriented concurrent system*, Series in Computer Systems, MIT Press, 1990.