A Front-End Tool for Automated Abstraction and Modular Verification of Actor-Based Models

Marjan Sirjani
Department of Computer Engineering, Sharif University of Technology
Azadi Ave., Tehran, Iran
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands
msirjani@cwi.nl

Amin Shali, Mohammad Mahdi Jaghoori, Hamed Iravanchi
Department of Electrical and Computer Engineering, Tehran University
Karegar Ave., Tehran, Iran
shali,jaghoori,iravanchi@ece.ut.ac.ir

Ali Movaghar
Department of Computer Engineering, Sharif University of Technology
Azadi Ave., Tehran, Iran
movaghar@sharif.edu

Abstract

Actor-based modeling is known to be an appropriate approach for representing concurrent and distributed systems. Rebeca is an actor-based language with a formal foundation, based on an operational interpretation of the actor model. We develop a front-end tool for translating a subset of Rebeca to SMV in order to model check Rebeca models. Automated modular verification and abstraction techniques are supported by the tool.

Keywords*: formal verification, model checking, compositional verification, actor model, reactive systems*

1 Introduction

Rebeca (*Reactive Objects Language*) is an actor-based language with a formal foundation, presented in [10, 12]. It can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model [7, 3]. It is also a platform for developing object-based concurrent systems in practice. Besides having an appropriate and efficient way for modeling concurrent and distributed systems, one needs a formal verification approach to ensure their correctness. Rebeca is supported by Rebeca Verifier tool, as a front-end, to translate Rebeca codes into languages of existing model-checkers and thus, be able to verify their properties. Modular verification and abstraction techniques are used to reduce the state space and make it possible to verify complicated reactive systems [5, 4, 8]. Interesting work has been done on formalizing the actor model, and there are other concurrent models supported by tools for modeling and verifying reactive systems, but to the best of our knowledge little is done about the formal verification of the actor model.

Rebeca Verifier is an environment to create Rebeca models, edit them, and translate them to SMV [1]. Also, modeler can enter the properties to be verified. The output code can be model checked by NuSMV. The main contribution is supporting modular verification and automatic abstraction. 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.

In the following section, we briefly explain Rebeca and

NuSMV. In Section 3, components, abstraction, and modular verification are explained. Section 4, shows the architecture and implementation of Rebeca Verifier. Section 5 is a short conclusion and an overview of our future work.

2 Rebeca and NuSMV

A model in Rebeca consists of a set of *rebec*s (*reactive object*) which are concurrently executed. Rebecs are encapsulated active objects, with no shared variables, which communicate via asynchronous message passing. Each rebec is instantiated from an *active 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 removed from the queue. Note that reading messages, thus, drives the computation of a rebec. Rebecs do not provide an explicit control over the message queue. Because of this asynchronous communication mechanism, with only an asynchronous send operation and no explicit receive operation, methods can be executed atomically.

NuSMV 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. We have developed a Rebeca Parser and a compiler of Rebeca into SMV [11]. The only data types in the SMV 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 nondeterministically choosing a *process*, then executing all of the assignment statements 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. The mapping from Rebeca constructs to SMV is shown in Table 1. Each active class in Rebeca is translated to a module in SMV and for each rebec a process is defined. 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.

Rebeca construct	SMV construct
activeclass	module
rebec	process
message queue	array
message server	distributed in the code of a process
state variables of a rebec	local variables of a process

Table 1. Mapping Rebeca constructs to SMV

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.

Instead of defining fixed length arrays for all rebecs, we let the modeler to define the length of the queue. A queue-overflow variable (corresponding to each rebec) is maintained in SMV code and can be checked as a property. Often, in our case studies, we had to increase the length of the queues to allow proper executions. Modeling the message queue as a structured variable increases the number of state variables considerably and it may cause state explosion quickly. A solution is using modular verification approach, for which we need to model components.

3 Components in Rebeca

In Rebeca, for verification purposes we may *decompose* a closed model and think of one part as the open system and the remainder as the environment that makes the overall system closed. This decomposition, determines which rebecs in the model have to be modeled with state and behavior and which rebecs may be abstracted away such that they only *send* messages. Since environment rebecs never execute their own methods, there is no need to model their message queues, state, or behaviors. They can be modeled as abstracted external rebecs, sending messages to internal rebecs of a component, in an arbitrary manner. By modeling environment in this way, certain properties still hold (namely, LTL-X,ACTL) [9].

Considering the unbounded queue, sending messages in an arbitrary manner will cause an infinite system. We use another abstraction technique to overcome this problem. Instead of putting incoming external messages in the queue, we enforce a fair interleaving process of them and messages

in the queue. It means that in the internal rebecs which could receive messages from outside, a fair nondeterministic choice is made between the internal message on top of the queue, and the external messages coming to it. In a number of conducted case studies, it has been shown that these techniques result in a huge reduction of the state space. The detailed explanation of these case studies, together with the theoretical background and the proof of correctness of our approach can be found in [11] and [10].

The state space may still be infinite, due to the unboundedness of message queues in Rebeca. Thus, we need to impose a user-specified, finite upper bound on the size of the message queue of each rebec. This may change the behavior of the model , so, we incorporate a user-defined queue length in our tool. Queue overflow can be checked as a property and queue length can be increased. While there is no queue overflow, the queue can be considered as an unbounded queue. This condition can be gained in many cases.

4 Tool Architecture and Implementation

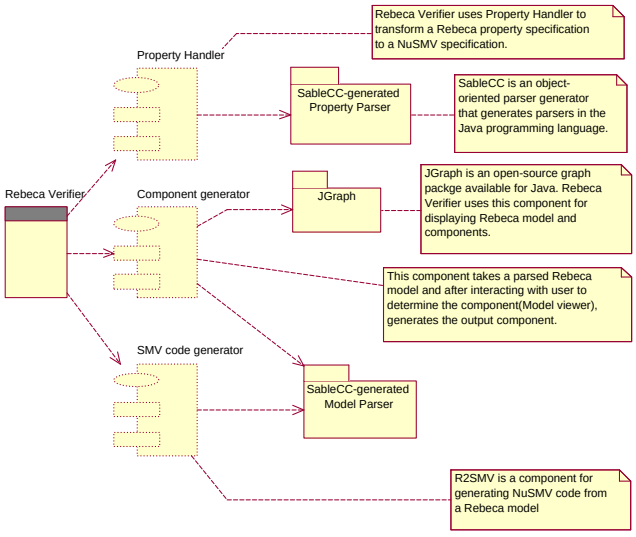

Figure 1. The Rebeca Verifier Component Diagram

The component diagram of Rebeca Verifier is shown in Figure 1. The components include Property handler, Component generator, and Code generator which uses Property parser, Model parser and JGraph packages. The code generator is written in Java. The modeler can also specify a property in the temporal logic supported by NuSMV, based on

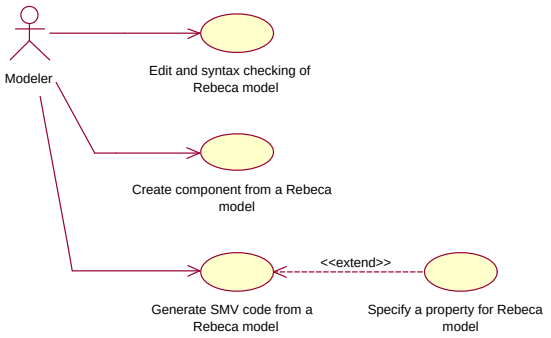

Figure 2. The Rebeca Verifier Use Case Diagram

rebec's variables at the source code level. The property handler, changes this property to the suitable form to be used by NuSMV.

Modular verification is supported by the component generator using the model viewer. Using the model viewer, modeler 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 which is composed by its environment makes up a close system, called a component model. Component models can also be translated into SMV, as described in Section 3. Figure 2, shows the use case diagram of the system. Figure 3, shows a snapshot of the tool, creating a component from well-known dining philosophers example, consisting of two forks and one philosopher.

5 Conclusion and Future Work

We generate a front-end tool for translating Rebeca models to SMV. Our tool supports modular verification, enabling the modeler to model check components derived from decomposing Rebeca models. Abstraction techniques are used to overcome state explosion problem.

Our research group in Tehran and Sharif universities has already implemented a translator of Rebeca to Promela [2]. It will be soon added to Rebeca Verifier. Rebeca group is also working on medium-sized case studies to be modeled and verified, for example IEEE CSMA/CD protocol is modeled in Rebeca (more information can be obtained from our home page at $http://khorshid.ut.ac.ir/\sim rebeca/$ or at $http://mehr.sharif.edu/\sim msirjani/rebeca$). Currently we are investigating the formal relationship between a subset of UML developed in European IST Project Omega [6] and an extended version of Rebeca. In this con-

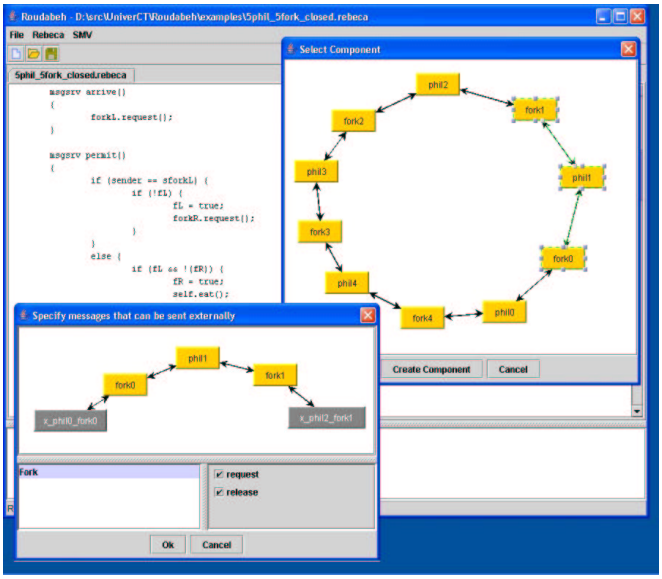

Figure 3. A snapshot of the tool, creating a component

tent, we add components as additional modeling structures. Synchronous message passing is allowed within a component, giving us the ability to model globally asynchronous and locally synchronous systems. By adding the required features, Rebeca can be used as a verifier in Omega project.

Special algorithms can be used to reduce the state space generated by executing Rebeca models, but this requires implementing a model checker algorithm from scratch. We are also working on this project. Other abstraction techniques, specially for abstracting the message queues, are under consideration.

References

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

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

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

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

[5] R. Alur, T. A. Henzinger, F. Y. C. Mang, and S. Qadeer. MOCHA: Modularity in model checking. In *Lecture Notes in Computer Science*, volume 1427, pages 521–525. Springer-Verlag, Berlin, 1998.

[6] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Proceedings of Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98, Leiden, The Netherlands. Springer-Verlag, Berlin, Germany, 2003.

[7] 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, Massachusettes Institute of Technology, April 1972.

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

[9] M. Sirjani and A. Movaghar. Simulation in Rebeca. In *Proceedings of PDPTA 2002*, pages 923–926. CSREA Press, USA, 2002.

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

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

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