$\mathcal{R}$ebeca2

Reference Manual
version 1.0.2

Hossein Hojjat
Formal Lab, University of Tehran
h.hojjat@ece.ut.ac.ir

December 18, 2006

Contents

Abstract

$\mathcal{R}$ebeca is a tool supported actor based modeling language that utilizes the compositional verification techniques. In this document we define the second version of $\mathcal{R}$ebeca. This version has extended the first version with seven new facilities: synchronous message passing, the capability of defining policies on queues, loops, enumerations, non-deterministic assignments, temporary variables and arrays. We also define SysReb, a new language based on Rebeca that is used for the verification of SystemC codes.

1 Introduction

The first version of $\mathcal{R}$ebeca as defined in [4, 7] was successfully applied to different case studies [3, 6]. Meanwhile, there were some demands made on the language to support new features. In various problems the $\mathcal{R}$ebeca users have noticed that if the language had some specific kind of features the modeling process would be easier and less cumbersome.
There were some attempts to introduce some parts of these new features to the language. For example, in [5] the authors have extended $\mathcal{R}$ebeca with components and synchronous message passing. Unfortunately none of the new suggestions for the language have been implemented completely in the tools. In order to unify the new requirements in a single language, we introduce $\mathcal{R}$ebeca2, a successor of the original $\mathcal{R}$ebeca language. The new tools that are being developed in the formal methods laboratory of the University of Tehran will all be based on $\mathcal{R}$ebeca2 definition.
This manual is divided into three sections. In the first section we describe what the original $\mathcal{R}$ebeca (hereafter $\mathcal{R}$ebeca1) used to be, and in the second section the $\mathcal{R}$ebeca2 additional abilities are defined. Section 3 is devoted to SysReb, an intermediate language for translation of SystemC codes.

2 $\mathcal{R}$ebeca

$\mathcal{R}$ebeca (Reactive Object Language) has been designed in an effort to facilitate the verification process for practitioners who are not experts in formal methods. From one point of view $\mathcal{R}$ebeca is a Java like language which is easy to use for software engineers, from another it is a modeling language with formal verification support and a background theory. A model in $\mathcal{R}$ebeca consists of concurrently executing reactive objects, rebecs. Computation takes place by asynchronous message passing between rebecs and execution of the corresponding methods of messages. Each message is put in the unbounded queue of the receiver rebec and specifies a unique method to be invoked when the message is serviced.

Figure 1 illustrates a simple $\mathcal{R}$ebeca class definition. Although in a pure actor model the queue length is unbounded, as a matter of model checking the modeler has to declare the maximum queue size in the class definition. This size shall be indicated in parenthesis next to the *reactiveclass* name. In a class definition there are two central declarations: the *knownrebecs* and *statevars*. The knownrebecs entry shows the rebecs that this rebec can communicate with them. The statevars are responsible for holding the rebec state. After these

```
reactiveclass Rebec1(2) {
 knownrebecs { Rebec2 d;}
 statevars{}
 msgsrv initial()
  { self.msg1();}
 msgsrv msg1()
  { d.askForService();}
 msgsrv msg2()
  { /* Handling message 2*/}
}
```

Figure 1: A typical class definition in $\mathcal{R}$ebeca

declarations, the message handling methods are defined in a Java like code. We call these methods the message servers of this reactiveclass, for the reason that their task is to serve the incoming messages.
In $\mathcal{R}$ebeca1, the execution of a message server can solely consists of three kinds of operations:

1. Executing an assignment: Changing the value of a state variable.
2. Evaluating a condition: The same as the if-conditional in Java.
3. Send a message: The dot notation is used to denote sending a message to a rebec.

The intra-object concurrency is different in actor and $\mathcal{R}$ebeca. In $\mathcal{R}$ebeca, objects have a unique thread of control. This brings simplicity and ease in model checking for $\mathcal{R}$ebeca. At each step the rebec takes a message from its queue and executes the corresponding message servers. Every reactive class definition has a message server named initial. In the initial state, each rebec has an initial message in its message queue, thus the first method executed by each rebec is the initial message server. After defining the reactive classes, there is a keyword *main* followed by the definition of the $\mathcal{R}$ebeca model which is a finite set of rebecs. In declaring a rebec, the bindings to its known rebecs are specified in the list of knownrebecs.

2.1 Known Rebecs

The communication links between different rebecs are unilateral, strict and unchangeable during the execution. By unilateral we mean that if a rebec can send a message to another one, the other rebec cannot necessarily send a message to the sender except that it has directly define it as a known rebec. By strict and unchangeable we mean that (at least in $\mathcal{R}$ebeca1) a rebec cannot modify its known rebecs during the execution.

```
<knownrebecs> ::= "{" <rebec_decl>* "};"
<rebec_decl> ::= <reactiveclass> <identifier> ";"

<statevars> ::= "{" <var_decl>* "};"
<var_decl> ::= <type> <identifier> ";"
<type> ::= boolean | int | shortint | byte
```

Figure 2: Declaring known rebecs and state variables

Table 1: The variable ranges in $\mathcal{R}$ebeca

Type Name	Range
boolean	Can only have the value true or false
int	-32768 to 32767
shortint	-128 to 127
byte	0 to 255

The rebecs which are included in the *knownrebecs* part of a reactive class definition are those rebecs whose message servers may be called by instances of this reactive class. The name for this declaration was a bit misleading in $\mathcal{R}$ebeca1 (knownobjects). As there are many differences between a typical programming object and a rebec, in $\mathcal{R}$ebeca2 we prefer to use *knownrebecs* instead of knownobjects. The syntax of declaring known rebecs is shown in Figure 2.

2.2 Statevars

There are two primary factors determining the state space of a $\mathcal{R}$ebeca program: the contents of the queues and the values of the state variables. Each rebec implicitly has an internal queue, and there is no need to declare it explicitly. The state variables are declared after the knownrebecs definition in a statevars block. In Figure 2 the declaration style and in Table 1 the variables ranges are specified.

2.3 Message servers

The execution of rebecs in a $\mathcal{R}$ebeca program takes place in a coarse grained interleaving scheme. In this manner each rebec dequeues a message from the top of its queue and executes it corresponding message server. During execution no other rebecs is allowed to be executed. Although this type of interleaving reduces the model expressiveness, but it $\mathcal{R}$ebeca2 we decide to keep this way of execution because of its clarity and ease of model checking. The declaration of a message server is very similar to a method declaration in Java with the difference that there is no returning value associated with a message server.

```
<rebeca statement> ::= <assignment> | <send message> | <conditional>
<assignment> ::= <identifier> "=" <expression> ";"
<conditional> ::= <if then statement> | <if then else statement>
<if then statement>::= if "(" <boolean expression> ")" <rebeca compound>
<if then else statement>::= if "(" <boolean expression> ")"
      <rebeca compound> "else" <rebeca compound>
<send message> ::= <rebec name> "." ( <argument list>* )
<argument list> ::= <expression> | <argument list> "," <expression>
```

Figure 3: $\mathcal{R}$ebeca statements

Table 2: Operators in $\mathcal{R}$ebeca

Operation Type	Operation	Definition
Arithmetic	+ - * /	
	%	mod
	=	assignment
Logic	&&	and
	||	or
	!	not
Comparative	< > >= <=	
	==	equality
	!=	inequality

2.4 $\mathcal{R}$ebeca Statements

A message server contains one or more $\mathcal{R}$ebeca statements. There are three classes of statements in $\mathcal{R}$ebeca1: conditional statements, assignments and sending message statements. The syntax is described in Figure 3. The logical and arithmetic expressions in $\mathcal{R}$ebeca are similar to Java, and the syntax is not included in this manual. One can refer to Java documents for the syntax. However, not all of the Java expressions are valid in $\mathcal{R}$ebeca, and only a set of essential set of operators are included. The acceptable operators are given in Table 2.

2.5 A typical example

Figure 4 shows a producer consumer problem which is modeled by $\mathcal{R}$ebeca.

3 $\mathcal{R}$ebeca2

$\mathcal{R}$ebeca2 modeling language is a slight movement towards the ultimate goal of $\mathcal{R}$ebeca, making the model checking process easier for practitioners. After collecting and reviewing the viewpoints of several students who have applied

```
reactiveclass Producer(2) {
knownrebecs { Consumer knownconsumer; }
statevars { boolean productsent; }
msgsrv initial() {
  productsent = false;
  self.produce(); }
msgsrv produce() {
  knownconsumer.consume();
  productsent = true; }
}
reactiveclass Consumer(2) {
knownrebecs { Producer knownproducer; }
statevars { boolean productreceived; }
msgsrv initial() {
  productreceived = false;
  self.consume(); }
msgsrv consume() {
  knownproducer.produce();
  productreceived = false; }
}
main {
  Producer producer1( consumer1);
  Consumer consumer1( producer1); }
```

Figure 4: A typical class definition in $\mathcal{R}$ebeca

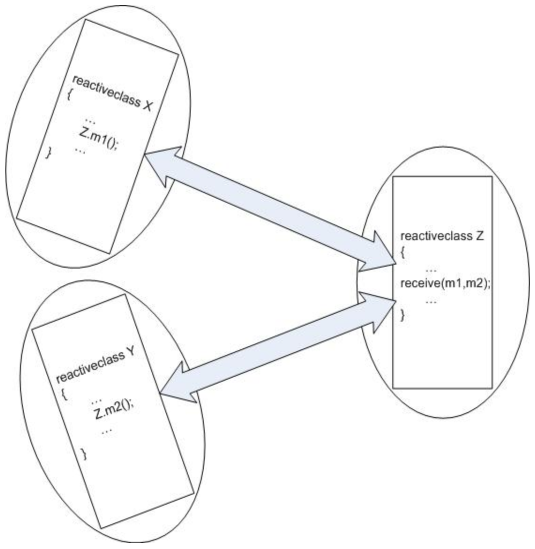

Figure 5: Rendezvous in $\mathcal{R}$ebeca2. The receiver can accept the sender non-deterministically.

the language to different case studies and projects, we decided to enrich the language with some new facilities to make the model checking easier. This section is dedicated to introduce theses new features.

3.1 Rendezvous

The rendezvous mechanism in $\mathcal{R}$ebeca2 is essentially based on Ada programming language. The minor distinction is that the receiver in the meeting point can non-deterministically choose the sender from a list. The syntax for a synchronous message is like sending a normal message, with the difference that there is not a message server matching the message in the receiver side. A sender which has sent the synchronous message *m1* to rebec *Z* is blocked until *Z* executes a *receive(m1)* statement (Figure 5).

3.2 Overflow policies

The actor model assumes that the internal queues of actors are infinite [1]. But in practice the model checkers cannot maintain this condition, and as a matter of fact they should set a finite length for queues. This value should be specified in the reactiveclass header (Section 2). This limitation causes some problems: the behavior of a full rebec which has received a new message from the outside world. In this case an overflow policy should be considered for the rebec. In $\mathcal{R}$ebeca2, there are two kinds of policies: drophead and droptail. In the first one the head of the queue is replaced with the new message, and in the second

```
<rebeca loop> ::= "for" <identifier> "=" <constant> "to" <constant> "do"
                  <rebeca compound>
```

Figure 6: $\mathcal{R}$ebeca2 for-statement

```
<EnumDeclaration> ::= "enum" <identifier> <EnumBody>
<EnumBody> ::= "{" <EnumConstants list> "};"
<EnumConstants list> ::= <EnumConstants> | <EnumConstants list> ","
<EnumConstants>

enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY };
```

Figure 7: Enumeration in $\mathcal{R}$ebeca2

one the tail is replaced. The policy is added to the head of a reactiveclass. For example, one can specify a droptail reactiveclass as below:
`reactiveclass y(qlength) :  droptail {···}`

3.3 Loops

In $\mathcal{R}$ebeca1 there is not any facility for *loop*, so, when a user wants to iterate over a set of statements s/he should explicitly mimic the loop by sending a message to the rebec itself. In reality this may not reflects the user's requirement, because the iterations of a single loop can be interleaved by the execution of some other message servers.
In $\mathcal{R}$ebeca2, the loop construct is introduced by including the Pascal like for-statements (Figure 6). The start and end points of the loops should be explicitly specified. This constrained version of the iteration does not have influences on the $\mathcal{R}$ebeca semantics, as it is solely a syntactic sugar for representing a group of sequentially running statements.

3.4 Enumerated types

In many cases, the ranges of the standard variables do not fit the problem. Many examples contain variables that won't become more than ten. If these variables are modeled with byte or a similar type, it may cause some troubles, especially for model checking engines that produce the whole state space at the beginning. For this purpose in $\mathcal{R}$ebeca2 we add the enumerated type. Figure 7 shows the BNF of enumerate types and also an example. These types are normally written at the top of a $\mathcal{R}$ebeca program.

3.5 Non-deterministic assignments

The nondeterministic assignments are valuable in many models. The $\mathcal{R}$ebeca2 statement $x =?(v_1, v_2, \cdots, v_n)$ assigns nondeterministically a value from the v_i's list to x.

3.6 Temporary variables

In many situations a rebec needs to work with a variable that is not included in the state space. We call these variables a temporary variable. These variables are declared locally in a message server and the value is accessible only in the message server context.

3.7 Arrays

Arrays are some syntactic sugars which have been added to $\mathcal{R}$ebeca2. The length of an array should be specified in declaration, as an example x is an array of bytes with length 3 in the following example:

```
byte [3] x;
```

A $\mathcal{R}$ebeca2 array is indexed from 0. So there are three indexes for the above declaration, namely 0, 1 and 2. The elements of an array can be accessed as in Java, for example `x[2]` denotes the last element of the array. Arrays have strict type checking in $\mathcal{R}$ebeca2, i.e. if a message server is willing to accept a variable of type `byte[2]`, a variable of type `byte[3]` cannot be passed to it.

4 SysReb

Recently, the researchers of the Formal methods laboratory of the University of Tehran have initiated a project on formal verification of the SystemC [2] codes. SystemC is a system description language having semantical similarities to some other hardware description language. In order to make $\mathcal{R}$ebeca a straightforward language for mapping the SystemC codes, some extra features are added to the $\mathcal{R}$ebeca. Although some of these newly added constructs does not necessarily conform to the design goals of $\mathcal{R}$ebeca (such as the object based paradigm) but we believe that they are a lot helpful in SystemC verification.

4.1 Global variables

A global variable is a variable that does not belong to any rebec, so it is accessible from anywhere in a $\mathcal{R}$ebeca program. The global variables are usually considered a bad practice in the programming community, and a programmer is advised not to use them. However, in some cases during the model checking process using a global variable instead of a set of local variables can reduce the state space. The most outstanding example is the case in which some rebecs are working on a shared variable. If all rebecs take a local copy of the variable the state space will increase enormously.

```
<while statement> ::= "while" "(" <boolean expression> ")"
                      <rebeca compound>
```

Figure 8: While loops in SysReb

Using global variables is normally not allowed for the end users. The feature is only used in some particular cases, essentially when $\mathcal{R}$ebeca is used as an intermediate language for model checking hardware designs.

4.1.1 Non-observable Global Variables

The declaration of non-observable global variables is the same as global variables, with the difference that the declaration is prefixed with the keyword hidden. These variables are not included in the state space.

4.2 Bitwise operations

Working with bits is not primarily supported in $\mathcal{R}$ebeca. We introduce four bitwise operations for this purpose: AND (&), OR (|), NOT(∼) and XOR(^) .

4.3 While loops

The $\mathcal{R}$ebeca2 loops (3.3) have a deterministic behavior, i.e., the number of iterations is known during the compilation process. The compiler can safely replace the loop with some statements running sequentially. In SysReb, we decide to include a more general, and perhaps a more model checking dangerous iteration, namely while loops. Although these loops can have unsafe behaviors such as infinite executions we consider them to facilitate converting SystemC codes to Rebeca. The syntax is similar to Java, and is shown in Figure 8.

References

[1] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[2] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[3] Hossein Hojjat, Hootan Nakhost, and Marjan Sirjani. Formal verification of the IEEE 802.1d spanning tree protocol using Extended Rebeca. In *Proceedings of the First IPM International Workshop on Foundations of Software Engineering*. Electronic Notes in Theoretical Computer Science, Volume 159, 2005.

[4] Marjan Sirjani. *Formal Specification and Verification of Concurrent and Reactive Systems*. PhD thesis, December 2004.

[5] Marjan Sirjani, Frank S. de Boer, Ali Movaghar, and Amin Shali. Extended Rebeca: a component-based actor language with synchronous message passing. In *Fifth International Conference on Application of Concurrency to System Design*, 2005.

[6] Marjan Sirjani, Mohammad Mahdi Jaghoori, Sara Forghanizadeh, Mona Mojdeh, and Ali Movaghar. Model checking CSMA/CD protocol using an actor-based language. *WSEAS Transactions on Circuit and Systems*, 4(6):1052– 1057, 2004.

[7] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.