$\mathcal{R}$ebeca

Reactive Object Language

Ali Jafari, Ehsan Khamespahanh
Hossein Hojjat, Zeynab Sabahi Kaviani,
and Marjan Sirjani
http://www.rebeca-lang.org

December 6, 2016

Contents

1 Introduction **1**

2 Structure of a $\mathcal{R}$ebeca Model **1**
2.1 Known Rebecs 3
2.2 Statevars 3
2.3 Message Servers 3
2.4 Methods 5
2.5 $\mathcal{R}$ebeca Statements 5
2.5.1 Loops 5
2.5.2 Conditional Statements 6
2.5.3 Non-deterministic Expressions 6
2.5.4 Local Variables 6
2.5.5 Arrays 7

3 A Typical Example **7**

Abstract

$\mathcal{R}$ebeca is an actor based modeling language that is supported by a model checking tool suite. In this document we explain the $\mathcal{R}$ebeca language. The syntax of Rebeca is defined formally and is explained through different examples.

1 Introduction

$\mathcal{R}$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, and from another point of view it is a modeling language

```
reactiveclass Rebec1(2) {
 knownrebecs { Rebec2 d;}
 statevars{}
 Rebec1()
  { self.msg1();
  /* the constructor */}
 msgsrv msg1()
  { d.askForService();}
 msgsrv msg2()
  { /* Handling message 2*/}
 void method1(int param1)
  { /* method definition */}
 int method2()
  { /* method definition */
   return intValue;}
}
```

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

with formal semantics and formal verification support. 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 message servers of messages. Each message is put in the queue of the receiver rebec and specifies a unique method to be invoked when the message is serviced.

2 Structure of a $\mathcal{R}$ebeca Model

Figure 1 illustrates a simple $\mathcal{R}$ebeca class definition. Although in a pure actor model the queue length is unbounded, for 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*. Known rebecs are the rebecs that messages are sent to. The statevars are responsible for holding the rebec state. After these declarations, the message handling methods are defined in a Java like code. We call these methods the message servers of this reactive class, and their task is to serve the incoming messages.

In addition to message servers which are defined using the keyword *msgsrv*, local methods can also be defined. Methods are local, and can only be called from within the message servers and methods of the same rebec. A method call can result in a return value to the caller.

Execution of a message server consists of the following operations:

1. Executing statements: statements are defined in Figure 2.

2. Sending a message: a message can be sent to a rebec (to this rebec or known rebecs).

3. Calling a method: a method is called by its name.

Execution of a method includes the three above mentioned operations except that messages can only be sent to this rebec, not to known rebecs. Sending a message to this rebec is done using the keyword *self*.

In $\mathcal{R}$ebeca, rebecs have a unique thread of control. This brings simplicity and ease in modeling. At each step the rebec takes a message from its queue and executes the corresponding message server.

Every reactive class definition has a constructor. In the initial state, each rebec has an constructor message in its message queue, thus the first message executed by each rebec is the constructor. The state variables can be initialized in the constructor.

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. The rebecs are instantiated from reactive classes. In declaring a rebec, the bindings to its known rebecs are specified in the list of known rebecs. The abstract syntax of Rebeca is shown in Figure 2.

2.1 Known Rebecs

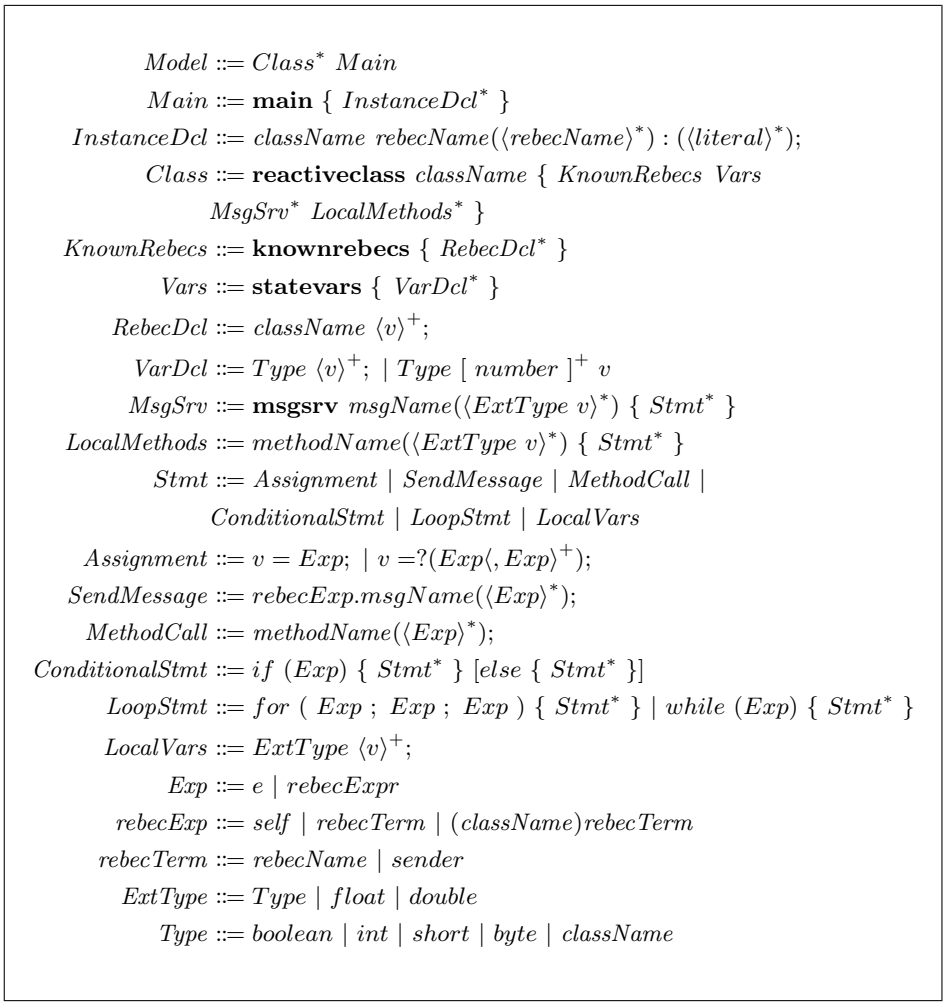
The rebecs which are included in the *knownrebecs* part of a reactive class definition are those rebecs whose message servers may be called by instances of this reactive class. The syntax of declaring known rebecs is shown in Figure 2.

There is another important feature in $\mathcal{R}$ebeca. A rebec can also send messages to variables of type *className* if their values are not equal to null. When sending the message, the value of these variables should be associated to a reference to a rebec.

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.

The state variables can be defined of types *int*, *byte*, *short*, *boolean*, or *className*. The type of *className* is similar to Java, and refers to the name of a defined reactive class in the $\mathcal{R}$ebeca program. In the constructor, state variables can be initialized. Specifically, variables of type *className* have to be initialized to *null*.



$$
\begin{aligned}
Model &::= Class^{*}\ Main \\
Main &::= \textbf{main}\ \{\ InstanceDcl^{*}\ \} \\
InstanceDcl &::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*}); \\
Class &::= \textbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars \\
&\quad MsgSrv^{*}\ LocalMethods^{*}\ \} \\
KnownRebecs &::= \textbf{knownrebecs}\ \{\ RebecDcl^{*}\ \} \\
Vars &::= \textbf{statevars}\ \{\ VarDcl^{*}\ \} \\
RebecDcl &::= className\ \langle v\rangle^{+}; \\
VarDcl &::= Type\ \langle v\rangle^{+}; \mid Type\ [\ number\ ]^{+}\ v \\
MsgSrv &::= \textbf{msgsrv}\ msgName(\langle ExtType\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
LocalMethods &::= methodName(\langle ExtType\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
Stmt &::= Assignment \mid SendMessage \mid MethodCall \mid \\
&\quad ConditionalStmt \mid LoopStmt \mid LocalVars \\
Assignment &::= v = Exp; \mid v = ?(Exp\langle, Exp\rangle^{+}); \\
SendMessage &::= rebecExp.msgName(\langle Exp\rangle^{*}); \\
MethodCall &::= methodName(\langle Exp\rangle^{*}); \\
ConditionalStmt &::= if\ (Exp)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}] \\
LoopStmt &::= for\ (\ Exp\ ;\ Exp\ ;\ Exp\ )\ \{\ Stmt^{*}\ \} \mid while\ (Exp)\ \{\ Stmt^{*}\ \} \\
LocalVars &::= ExtType\ \langle v\rangle^{+}; \\
Exp &::= e \mid rebecExpr \\
rebecExp &::= self \mid rebecTerm \mid (className)rebecTerm \\
rebecTerm &::= rebecName \mid sender \\
ExtType &::= Type \mid float \mid double \\
Type &::= boolean \mid int \mid short \mid byte \mid className
\end{aligned}
$$

Figure 2: (a) Abstract syntax of Rebeca. Angle brackets $\langle ... \rangle$ are used as meta parentheses, superscript $+$ for repetition at least once, superscript $*$ for repetition zero or more times, whereas using $\langle ... \rangle$ with repetition denotes a comma separated list. Brackets [...] indicate that the text within the brackets is optional. The symbol ? shows nondet choice. Identifiers $className$, $rebecName$, $methodName$, v, and $literal$ denote class name, rebec name, method name, variable, and literal, respectively; and e denotes an (arithmetic, boolean, nondet choice, or a rebec name in case of a variable of type reactive class) expression.

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

Type Name	Range
boolean	true, false
int	-2147483648 to 2147483647
short	-32768 to 32767
byte	-128 to 127

2.3 Message Servers

Execution of rebecs in a $\mathcal{R}$ebeca program takes place in an interleaving scheme. In this manner each rebec dequeues a message from the top of its queue and executes its corresponding message server. During execution no other message servers of this rebec is allowed to be executed. 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. The message servers can have input parameters which can be of different types as shown in Figure 2.

2.4 Methods

Methods can be defined in a reactive class. Methods are local to the container rebec, and can only be called by the message servers and other methods of this rebec. A method can return a value to its caller, and can include input parameters of types *ExpType*, as defined in Figure 2. When a variable of type *className* is passed as an argument to message servers or methods, its value has to be binded to the reference to a rebec (i.e. an instance of a reactive class which is defined in the main part). Note that a rebec can send a message to a variable of type *className* if its value is not equal to null.

It's worth mentioning that a method can only send a message to the rebec containing it. This rebec is accessible by using keyword $self$, which is a reference to a rebec (an instance of this reactive class).

2.5 $\mathcal{R}$ebeca Statements

A message server contains one or more $\mathcal{R}$ebeca statements. There are different types of statements in $\mathcal{R}$ebeca: conditional statements, assignments, non-deterministic expressions, local variables declaration, for-loop and while-loop statements, method calls, and sending message statements. The syntax is described in Figure 2.

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. The set of acceptable operators are given in Table 2. We should emphasize that casting in $\mathcal{R}$ebeca is the same as in Java. There is a predefined variable named *sender* in $\mathcal{R}$ebeca. The receiver of a message can get a reference to the sender of the message through accessing the value of variable *sender*. The

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

Operation Type	Operation	Definition
Arithmetic	+ - * /	
	%	mod
	++ - -	
Assignment	=	
	+= -= *= /= %=	
	&= |=	
Conditional	?:	ternary conditional
Logic	&&	logical and
	||	logical or
	!	
	&	bitwise and
	|	bitwise or
Comparative	< > >= <=	
	==	equality
	!=	inequality
Cast		like in Java

following example shows the casting of variable *sender*.

```
senderDevice = (CommunicationDevice)sender;
```

In this example, the value of *sender* is assigned to the variable `senderDevice` which is of type `CommunicationDevice` (a defined reactive class in the program).

2.5.1 Loops

In $\mathcal{R}$ebeca, for-loop and while-loop constructs are introduced with the same syntax as in Java. To facilitate the loop iteration, *break* and *continue* are included too. The syntax is shown in Figure 2.

2.5.2 Conditional Statements

The *switch* conditional statement has been added to $\mathcal{R}$ebeca and its syntax is like Java. So, the keywords *case*, *break* and $default$ have been added too. The case expression will be valid if its value is determined at compile time. The if conditional statement can also be used in $\mathcal{R}$ebeca.

2.5.3 Non-deterministic Expressions

The non-deterministic expressions are valuable in many models. The assignment $x = ?(e_1, e_2, \cdots, e_n)$ assigns non-deterministically a value from the e_i's valuation to x. It means that e_i can be an expression. The expression will be valid

```
msgserver AdaptFrequency() {
  int noise, frequency;
  bool isNoisy;
  noise = ?( -1, 1) + 0;
  isNoisy = ?(true,false);
  if ( isNoisy )
    frequency = ?(18 , 2 + noise);
  else
    frequency = 18;
}
```

Figure 3: An example of nondeterministic expressions

if its value is determined at compile time. An example of non-deterministic expressions is shown in Figure 3. This Figure shows a *msgserver* that sets the frequency by non-deterministically choosing occurrence of a noise and the amount of it.

2.5.4 Local Variables

In many situations a rebec needs to work with a variable that is not included in the state space. We call these variables local variables. These variables are declared locally in a message server and the value is accessible only in the message server context. Their value are not considered in state space. The possible types of local variables are shown in Figure 2 (as *ExtType*).

2.5.5 Arrays

Arrays can be defined in a $\mathcal{R}$ebeca program. The type of arrays can be *int*, *byte*, *short*, *boolean*, or *className*, as defined in Figure 2. The length of an array should be specified in its declaration. As an example, x is an array of bytes with length 3 in the following declaration:

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

An 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 $\mathcal{R}$ebeca, 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.

3 A Typical Example

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

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

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