Formal Semantics and Analysis of Timed Rebeca in Real-Time Maude

Zeynab Sabahi-Kaviani[1], Ramtin Khosravi[1], Marjan Sirjani[2,1], Peter Csaba Ölveczky[3], and Ehsan Khamespanah[1]

[1] School of Electrical and Computer Engineering, University of Tehran
[2] School of Computer Science, Reykjavik University
[3] Department of Informatics, University of Oslo

Abstract. The actor model is one of the main models for asynchronous and distributed computation. Timed Rebeca is a timed extension of the actor-based modeling language Rebeca. Although Rebeca is supported by a rich verification toolset, Timed Rebeca has not had an executable formal semantics, and has therefore had limited support for formal analysis. In this paper, we provide a formal semantics of Timed Rebeca in Real-Time Maude. We have automated the translation from Timed Rebeca to Real-Time Maude, allowing Timed Rebeca models to be automatically analyzed using Real-Time Maude's reachability analysis tool and timed CTL model checker. This enables a formal model-based methodology which combines the convenience of intuitive modeling in Timed Rebeca with formal verification in Real-Time Maude. We illustrate this methodology with a collision avoidance protocol for wireless networks.

1 Introduction

The importance of formal modeling and analysis for ensuring the dependability and correctness of safety-critical systems has long been acknowledged. However, the lack of formal modeling languages close to programming and modeling languages used by practitioners has limited the use of formal methods. Timed Rebeca [1] is an actor-based [2] modeling language that extends the Rebeca language [17] to support the modeling of distributed real-time systems. Because of its Java-like syntax and its simple and intuitive message-driven and object-based computational model, Timed Rebeca is an easy-to-learn language for system developers, thereby bridging the gap between formal methods and practical software engineering.

Although Rebeca is supported by a rich model checking toolset [15], model checking of Timed Rebeca models has not been supported until now. Even though Timed Rebeca has an SOS semantics, it lacks an executable formal semantics that would enable automated analysis methods such as simulation and temporal logic model checking.

However, providing an executable formal semantics for Timed Rebeca is quite challenging. For example, since Timed Rebeca has a rich expression/statement language that allows the values of state variables to grow beyond any bound, and since the message queues can become arbitrarily long, Timed Rebeca cannot be translated into popular real-time formalisms such as, e.g., timed automata.

In this paper, we provide a formal Real-Time Maude semantics for Timed Rebeca. Real-Time Maude [12] is a specification formalism and analysis tool for real-time systems based on rewriting logic [11]. With its natural time model and expressive formalism, which is particularly suitable for formally specifying distributed real-time systems in an object-oriented way, Real-Time Maude should be ideally suited for this challenging task. Real-Time Maude is supported by a high-performance toolset providing a spectrum of analysis methods, including simulation through timed rewriting, reachability analysis, and (untimed) linear temporal logic model checking as well as timed CTL model checking.

We have automated the translation from Timed Rebeca to Real-Time Maude, so that the user gets Real-Time Maude simulation and model checking of his/her Timed Rebeca model for free. Furthermore, such formal analysis is being integrated into the Rebeca toolset. This would of course not be very useful if the user would need to understand the Real-Time Maude representation of his/her Timed Rebeca model, and/or would need to define state properties in Real-Time Maude, in order to model check his/her Timed Rebeca model. We have therefore taken advantage of Real-Time Maude's support for *parametric* state propositions to predefine useful generic state propositions, so that the user can define his/her (possibly timed) temporal logic properties *without* having to know Real-Time Maude or understand how the mapping from Timed Rebeca works.

Altogether, this enables a formal model-engineering methodology that combines the convenience of modeling in an intuitive actor language with Java-like syntax with formal verification in Real-Time Maude. We illustrate this methodology with a collision avoidance protocol case study.

The rest of the paper is structured as follows. Section 2 briefly introduces Timed Rebeca and Real-Time Maude. Section 3 explains the Real-Time Maude formalization of the Timed Rebeca semantics. Section 4 defines some useful generic atomic state propositions that allows the user to easily define his/her temporal logic formulas without knowing Real-Time Maude. Section 5 illustrates our methodology on a collision avoidance protocol. Finally, Section 6 discusses related work and Section 7 gives some concluding remarks.

2 Preliminaries

2.1 Timed Rebeca

Since Timed Rebeca is an extension of the Rebeca modeling language, we first introduce Rebeca and then explain Timed Rebeca in more detail.

Rebeca [17] is a pure actor-based modeling language suitable for specifying distributed systems. Rebeca is supported by a rich model checking toolset [15].

A Rebeca model consists of a set of actors (called *rebecs*) that communicate asynchronously by message passing. Each actor maintains a queue of messages that it has received but not yet processed. An actor repeatedly takes a message

$$\begin{array}{rcl}
\mathit{Model} & ::= & \mathit{Class}^{*}\ \mathit{Main} \\
\mathit{Main} & ::= & \textbf{main}\ \{\ \mathit{InstanceDcl}^{*}\ \} \\
\mathit{InstanceDcl} & ::= & \mathit{className}\ \mathit{rebecName}(\langle \mathit{rebecName}\rangle^{*}) : (\langle \mathit{literal}\rangle^{*}); \\
\mathit{Class} & ::= & \textbf{reactiveclass}\ \mathit{className}\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{Constr}\ \mathit{MsgSrv}^{*}\ \} \\
\mathit{KnownRebecs} & ::= & \textbf{knownrebecs}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{Vars} & ::= & \textbf{statevars}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{VarDcl} & ::= & \mathit{type}\ \langle v\rangle^{+}; \\
\mathit{Constr} & ::= & \mathit{className}\ \mathit{methodName}(\langle \mathit{type}\ v\rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{MsgSrv} & ::= & \textbf{msgsrv}\ \mathit{methodName}(\langle \mathit{type}\ v\rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{Stmt} & ::= & v = e;\ \mid\ v =?(e_1, \ldots, e_n)\ \mid\ \mathit{Send};\ \mid\ \textbf{if}\ (e)\ \{\ \mathit{Stmt}^{*}\ \}\ [\textbf{else}\ \{\ \mathit{Stmt}^{*}\ \}]\ \mid \\
 & & \textbf{delay}(e);\ \mid\ \textbf{for}\ (\mathit{Stmt}_1;\ e;\ \mathit{Stmt}_2)\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{Send} & ::= & \mathit{rebecName}.\mathit{methodName}(\langle e\rangle^{*})\ [\textbf{after}(e)]\ [\textbf{deadline}(e)]
\end{array}$$

Fig. 1. Abstract syntax of Timed Rebeca. Angle brackets ⟨...⟩ are used as meta parenthesis. Identifiers *className*, *rebecName*, *methodName*, v, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and e denotes an (arithmetic or boolean) expression. In *for* loops, Stmt_1 is the initialization statement, e_2 is a boolean expression (the loop execution condition), and Stmt_2 is the update statement (executed after each iteration).

from the beginning of its queue and executes the corresponding *message server*, which may involve sending messages to other actors and changing the actor's local state. Execution is non-preemptive: the actor does not take the next message from its queue before the running message server is finished.

A Rebeca specification defines a number of *reactive classes* and a *main* block. A reactive class defines an actor type and its behavior as well as its relationship to other actors. The main block is used to instantiate *rebecs* as the objects of reactive classes. The body of a reactive class definition has three sections: *known rebecs*, *state variables*, and *message servers*. A rebec can only send messages to its known rebecs. The local state of a rebec is given by the values of its state variables. The type of state variables can be integer types, Boolean, and arrays.

The message servers specify how the rebecs respond to incoming messages. They may have parameters and may define local variables. The body of a message server consists of a number of statements, including assignments, conditionals, loops, and sending messages. The expressions contains common arithmetic and logical operators. The nondeterministic assignment $v =?(e_1, \ldots, e_k)$ nondeterministically assigns (the current evaluation of) one of the expressions e_i to the variable v. Each class has a constructor (with the same name as the class) which initializes the state variables of its instances.

Timed Rebeca [1] is a timed extension of Rebeca whose abstract syntax is given in Fig. 1. The following timed features have been added for specifying distributed real-time systems:

- *delay* is a statement used to model *computation times*. Since we assume that the execution times of the other statements to be zero, the computation time must be specified by the modeler using the delay statement.
- *after* is a time tag attached to a message and defines the earliest time the message can be served, *relative* to the time when the message was sent.
- *deadline* is a time tag attached to a message which determines the expiration time of the messages, *relative* to the time when the message was sent.

When a message with tag *after* t is sent, it is added to the set of *undelivered messages* and resides there until t time units have elapsed. Then, it is *delivered*, i.e., appended to the receiving rebec's message queue. The messages in a rebec's queue are therefore ordered according to their delivery time (if the delivery time of two messages are the same, the order in which they are delivered is selected nondeterministically). If the deadline of a message is reached, regardless of whether it is delivered or not, the message is purged. A rebec takes a message from its queue as soon as it can (i.e., when it has finished processing the previous message, and there are some messages in the queue).

Figure 2 shows a Timed Rebeca model of a simple thermostat system composed of two actors `t` and `h` of reactive classes `Thermostat` and `Heater`, respectively. The goal of the system is to keep the temperature between 25 and 30 degrees. The `Thermostat` actor checks the temperature every 5 time units, by sending a `checkTemp` message to itself (line 19). If the temperature is not in the acceptable range, it sends the `Heater` actor `h` the proper `on` or `off` message, which expires after 20 time units (lines 16 and 18). It takes two time units for the heater to turn itself on or off. The heater also models the change in the environment by nondeterministically changing the temperature by 1 to 3 degrees every 10 time units (lines 47-49), and sending the delta to the heater (line 50).

2.2 Real-Time Maude

Real-Time Maude [13, 12] extends the rewriting-logic-based Maude language and tool [6] to support the formal specification and analysis of real-time systems. A Real-Time Maude timed module is a tuple *(Σ,E,IR,TR)*, where:

- *(Σ,E)* is a membership equational logic [6] theory where Σ is an algebraic signature, declaring the sorts, subsorts, and functions of the system, and E a set of confluent and terminating conditional equations. *(Σ,E)* specifies the system's states as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- *IR* is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e, zero-time) local transitions, written with syntax `rl [`l`] : `u` => `v, where l is a label. Such a rule specifies a *one-step transition* from an instance of the term u to the corresponding instance of the term v. The rules are applied *modulo* the equations E.
- *TR* is a set of (usually conditional) *tick rules*, written with syntax `crl [`l`] : {`t`} => {`t'`} in Time `τ` if cond`, that model time elapse. `{_}` is a built-in

```
reactiveclass Thermostat {
  knownrebecs {
      Heater heater;
  }
  statevars {
    int period;
    int temp;
  }
  Thermostat() {
    period = 5;
    temp = 25;
    self.checkTemp();
  }
  msgsrv checkTemp() {
    if (temp >= 30)
      heater.off() deadline(20);
    if (temp <= 25)
      heater.on() deadline(20);
    self.checkTemp()
         after(period);
  }
  msgsrv changeTemp(int delta) {
    temp = temp + delta;
  }
}

reactiveclass Heater {
  knownrebecs {
    Thermostat thermostat;
  }
  statevars {
    boolean on;
    int delta;
  }
  Heater() {
    on = false;
    self.run();
  }
  msgsrv on() {
    delay(2);
    on = true;
  }
  msgsrv off() {
    delay(2);
    on = false;
  }
  msgsrv run(){
    delta = ?(1,2,3);
    if (on == false)
      delta = -1 * delta;
    thermostat.changeTemp(delta);
    self.run() after(10);
  }
}

main {
  Thermostat t(h):();
  Heater h(t):();
}
```

Fig. 2. The Timed Rebeca model for a simple thermostat/heater system.

constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form `{`u`}` using the equations in the specification.

The Real-Time Maude syntax is fairly intuitive. A function symbol f in Σ is declared with the syntax `op` f `:` s_1 `...` s_n `->` s, where $s_1 \dots s_n$ are the sorts of its arguments, and s is its result *sort*. Equations are written with syntax `eq` u `=` v, and `ceq` u `=` v `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. An equation $f(t_i, \dots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a subterm $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.

A *class* declaration `class` C `|` att_1 `:` s_1`,` `...` `,` att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n, respectively. An *object* of class C is represented as a term `<` $O : C$ `|` $att_1 : val_1$ `, ... ,` $att_n : val_n$ `>` where O, of sort `Oid` is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. The state is a term of sort

`Configuration`, and has the structure of a *multiset* of objects and *messages*, with multiset union denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is *multiset rewriting.*

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :
   m(O,w)
   < O : C | a1 : x, a2 : O', a3 : Z >
  =>
   < O : C | a1 : x+w, a2 : O', a3 : Z >
   dly(m'(O'),x) .
```

defines a parameterized family of transition in which a message `m`, with parameter `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of the object `O` and send a new message `m'(O')` with delay `x`.

Formal Analysis. The Real-Time Maude tool provides a spectrum of analysis methods, including:

– *timed rewriting* that simulates one behavior of the system *up to certain duration* from an initial state;
– *timed search* analyzes whether a state matching a state pattern is reachable from the initial state within a certain time interval;
– *model checking* to check whether each possible behavior from the initial state satisfies a temporal logic formula. Real-Time Maude extends Maude's *linear temporal logic model checker. State proposition* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form `{`*statePattern*`} |=` *prop* `=` *b*, for a *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in a state `{`*t*`}` if `{`*t*`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, $\sim$ (negation), /\, \/, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has the syntax `mc {`*t*`} |=t` φ `in time <=` τ . for initial state `{`*t*`}` and temporal logic formula φ. Real-Time Maude has also recently been equipped with a model checker for *timed* computation tree logic (TCTL) properties [10].

3 Real-Time Maude Semantics of Timed Rebeca

This section explains how we have formalized the semantics of Timed Rebeca in Real-Time Maude in an object-oriented style.

Specifying the Static Parts. In the Real-Time Maude semantics of a Timed Rebeca model we need to keep track of (i) the *declarations* of the (message servers of the) reactive classes; (ii) the rebecs in their current states; and (iii) the set of as-yet undelivered messages.

Since the message servers do not change dynamically, we do not need to carry them around in the state. Instead, the message servers are modeled by a function

```
op msgServer : ClassName MsgHeader -> Statements .
```

where `msgServer(`c, m`)` defines the code to be executed by a rebec of reactive class c when it treats a message with header m. The sort `Statements` is a straight-forward representation of the body of a message server. For example, in our thermostat example, `msgServer(Thermostat, Thermostat)` equals

```
(period := 5) ; (temp := 25) ;
(sendSelf checkTemp with noArg deadline INF after 0)
```

and `msgServer(Thermostat, checkTemp)` equals

```
(if( temp >= 30 ) then (send off with noArg to "heater" deadline 20 after 0)) ;
(if( temp <= 25 ) then (send on  with noArg to "heater" deadline 20 after 0)) ;
(sendSelf checkTemp with noArg deadline INF after 5)
```

We also have a function `formalParams` such that `formalParams(`c, m`)` returns the list of the formal parameters of the message server for m in reactive class c.

We mostly omit the details of how basic Rebeca statements (e.g., assignments and evaluation of expressions) are formalized in Real-Time Maude, and refer to [3] for a thorough treatment of the Real-Time Maude formalization of the evaluation of expressions in a fairly sophisticated language. The only expression we mention is due to the possibility of having *nondeterministic assignments*. We formalize the expression list $?\,(e_1, e_2, \ldots, e_n)$ in a nondeterministic assignment as a list e_1 `?` e_2 `?` $\ldots$ `?` e_n using the following list data type:

```
sort NDExpr .    subsort Expr < NDExpr .
op nil : -> NDExpr .
op _?_ : NDExpr NDExpr -> NDExpr [assoc id: nil] .
```

Since `nil` is the identity element for lists, Maude considers l and `nil ?` l and l `? nil` to be identical lists. In particular, a single expression e is considered by Maude to be identical to the lists `nil ?` e and e `? nil` and `nil ?` e `? nil`.

The state of the Real-Time Maude representation of a Timed Rebeca model is a multiset consisting of one `Rebec` object for each rebec in the system and one message for each message in the set of undelivered messages.

A rebec is modeled by an object instance of the following class `Rebec`:

```
class Rebec | stateVars : Valuation,    queue : MsgList,
              classId : ClassName,      toExecute : Statements,
              knownRebecs : KnownList .
```

where `stateVars` represents the state variables of the rebec and the formal parameters of the message server being treated, together with their current values, as a set of terms of the form *var-name* `|->` *value*; `queue` is a '`::`'-separated list of messages representing the message queue of the rebec; `classId` is the name of the reactive class of the rebec; `toExecute` denotes the remaining statements the rebec has to execute (and is `noStatements` if the rebec is not executing a message server); and `knownRebecs` denotes the "known rebecs" of the rebec.

For example, the following term models the rebec `"t"` of class `Thermostat` right after completing its constructor. Its state variables have the values 5 and 25, there is only one message in its queue (sent by itself), and the rebec is not executing any message server.

```
< "t" : Rebec | stateVars : ('period |-> 5) ('temp |-> 25),
                queue : (checkTemp with noArg from "t" to "t" deadline INF),
                classId : Thermostat,
                toExecute : noStatements,
                knownRebecs : (Heater heater --> "h") >
```

Communication between rebecs takes place when a rebec sends a message to another rebec (or to itself). The message is put into the multiset of undelivered messages until its message delay ends. It is then delivered to the receiver's message queue. *Delivered messages* are modeled using the constructor

```
msg _with_from_to_deadline_ : MsgHeader Valuation Oid Oid TimeInf -> Msg .
```

A delivered message therefore contains a header (the message name), its arguments, the id of the sender rebec, the id of the receiver, and the time remaining until the expiration (deadline) of the message. *Delayed messages* have the form `dly`(m, t), where m is a message as above and t is the remaining delay of the message, and where `dly`$(m, 0)$ is considered to be identical to m [12].

Instantaneous Transitions. We next formalize the instantaneous actions of a Timed Rebeca rebec using rewrite rules. We show 9 of the 16 rewrite rules that define our semantics of the Timed Rebeca.

In the following rule, an idle rebec takes the first message from its queue and starts executing the statements in the corresponding message server by putting those statements into its `toExecute` attribute. Some additional bookkeeping is also required: the formal parameters of the message server must be initialized to the values in the message and added to the state variables; to clean up at the end of the execution, we add a new statement `removeVars` to execute *after* the statements in the message server have been executed:[4]

```
rl [takeMessage] :
   < O : Rebec | stateVars : SVARS,
                 queue : (M with VAL from O' deadline DL) :: MSGLIST,
                 classId : C, toExecute : noStatements >
=>
   < O : Rebec | stateVars : SVARS  VAL ('sender |-> O'),
                 queue : MSGLIST,
                 toExecute : msgServer(C,M) ; removeVars(VAL ('sender |-> O')) > .
```

[4] In this paper we follow the Maude convention that variables are written with (only) capital letters, and do not show the variable declarations.

Because of the possibility of having nondeterministic assignments, the rewrite rule modeling (both deterministic and nondeterministic) assignment is interesting. The following rule uses pattern matching and the fact that the list concatenation operator `?` is declared to be associate and to have identity `nil` to nondeterministically select *any* possibly expression `EX` from a list of expressions. This rule also covers *deterministic* assignment, since the list variables `LIST1` and `LIST2` may both match the empty list `nil`. In addition, the rebec updates its `toExecute` attribute to only execute the remaining statements:

```
  rl [detAndNondetAssignment]  :
     < O : Rebec | stateVars : (VAR |-> VAL) SVARS,
                   toExecute : (VAR := LIST1 ? EX ? LIST2) ; STMTLIST >
    =>
     < O : Rebec | stateVars : (VAR |-> evalExp(EX, (VAR |-> VAL) SVARS)) SVARS,
                   toExecute : STMTLIST > .
```

We next describe the semantics of loops `for (`*init*`;` *cond*`;` *update*`)`{*body*}, where *init* is a statement executed once in the beginning, *cond* is a Boolean expression that must be true to continue the iterations, *update* is a statement executed after each iteration, and *body* is a statement list executed in each iteration. The semantics of loops is formalized in a standard "unfolding" style:

```
rl [forLoop] :
   < O : Rebec | toExecute : for(INIT, COND, UPDATE, BODY) ; STMTLIST >
  =>
   < O : Rebec | toExecute : INIT ; iterate(COND, UPDATE, BODY) ; STMTLIST > .

rl [iterate] :
   < O : Rebec | stateVars : SVARS,
                 toExecute : iterate(COND, UPDATE, BODY) ; STMTLIST >
  =>
   < O : Rebec | toExecute : if evalBoolExp(COND, SVARS) then
                             BODY ; UPDATE ; iterate(COND, UPDATE, BODY) ; STMTLIST
                             else STMTLIST fi > .
```

If the first statement is a send statement, the rebec creates a delayed message which is added to the undelivered message soup.

```
  rl [sendMessage] :
     < O : Rebec | stateVars : SVARS,
                   toExecute : (send M with ARGS to REC deadline DL after AFT)
                               ; STMTLIST , knownRebecs : (CN NK --> RCVR) NL >
    =>
     < O : Rebec | toExecute :  STMTLIST >
     dly(M with getVals(ARGS, SVARS, formalParams(CN,M)) from O to RCVR
           deadline evalIntExp(DL,SVARS),
         evalIntExp(AFT,SVARS)) .
```

Both `DL` and `AFT` are expressions evaluated using `evalIntExp` in the context of the current variable assignment `SVARS`. The created message is added to the system configuration; when its remaining delay becomes 0, the message becomes "undelayed" as explained above, and can be received by the intended recipient, which puts the message into its message queue:

```
  rl [readMessage] :
     (M with ARGS from O to O' deadline DL )
     < O' : Rebec | queue : MSGLIST >
    =>
     < O' : Rebec | queue : MSGLIST :: (M with ARGS from O deadline DL) > .
```

Another interesting case is the execution of a delay statement, which is treated as follows: When the rebec encounters the delay statement, it evaluates the delay expression using the current values of the variables. Once it has done that, it leaves the delay statement in the beginning of its `toExecute` attribute *until* the remaining delay becomes 0, when the rebec just continues with the next statement. Decreasing the remaining delay is done by the tick rule below. The following rules then, respectively, evaluate the delay expression at the beginning of the delay, and finish the delay when the remaining delay is 0:

```
  crl [evaluateDelayExpression] :
      < O : Rebec | stateVars : SVARS, toExecute : delay(EX) ; STMTLIST >
     =>
      < O : Rebec | toExecute :  delay(evalIntExp(EX, SVARS)) ; STMTLIST  >
     if not (EX :: Int) .

  rl [endDelay] :
     < O : Rebec | toExecute : delay(0) ; STMTLIST >
    =>
     < O : Rebec | toExecute :  STMTLIST  > .
```

Timed Behavior. The following "standard" object-oriented tick rule [12] is used to model time advance until the next time when something must "happen":

```
var SYSTEM : Configuration .
crl [tick] : {SYSTEM}  =>  {elapsedTime(SYSTEM, mte(SYSTEM))} in time mte(SYSTEM)
             if mte(SYSTEM) > 0 .
```

The variable `SYSTEM` matches the entire state of the system. The function `mte` (*m*aximal *t*ime *e*lapse) determines how much time can advance in a given state. If an instantaneous rule is enabled, it must be executed immediately; therefore, `mte` of a state must be zero when an instantaneous rule is enabled in that state.

The function `mte` is the minimum of the `mte` of each rebec and each message in the soup. As mentioned above, the `mte` must be 0 when the rebec has a statement to execute which does not have the form `delay`(i), for an integer i; in the latter case, the `mte` equals i. If there are no statements to be executed, the `mte` equals 0 if the rebec has a message in its queue, and equals the infinity value `INF` if the message queue is empty:

```
op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
eq mte(dly(M,T)  CONF) = min(T, mte(CONF)) .
ceq mte(OBJECT CONF) = min(mte(OBJECT), mte(CONF)) if CONF =/= none .
eq mte(< O : Rebec | toExecute : noStatements, queue : empty >) = INF .
eq mte(< O : Rebec | toExecute : delay(T) ; STMTLIST >) = T .
eq mte(< O : Rebec | >) = 0 [owise] .
```

The function `elapsedTime` models the effect of time elapse on a state as follows: The effect of time elapse on a rebec is that the remaining time until the message deadline is decreased according to the elapsed time for each message in the queue. Furthermore, the remaining delay of a delay statement being executed is also decreased according to the elapsed time. For messages traveling between rebecs, their remaining delays and deadline are decreased according to the elapsed time. In both cases, if the deadline expires before the message is treated, the message is purged (i.e., becomes the empty configuration `none`):

```
op elapsedTime : Configuration Time -> Configuration [frozen (1)] .
eq elapsedTime(none, T) = none .
eq elapsedTime(dly(M with ARGS from O to O' deadline T1, T2)  CONF, T)
 = (if T2 <= T1 then dly(M with ARGS from O to O' deadline (T1 - T), T2 - T)
    else none fi)   elapsedTime(CONF, T) .
eq elapsedTime(< O : Rebec | toExecute : STMTLIST, queue : MSGLIST >  CONF, T)
 = < O : Rebec | toExecute : decreaseDelay(STMTLIST, T),
                 queue : decreaseDeadline(MSGLIST, T) >   elapsedTime(CONF, T) .

op decreaseDelay : StatementList Time -> StatementList .
eq decreaseDelay(delay(T1) ; STMTLIST, T) = delay(T1 - T) ; STMTLIST .
eq decreaseDelay(STMTLIST, T) = STMTLIST [owise] .
op decreaseDeadlines : MsgList Time -> MsgList .
eq decreaseDeadlines(nil, T) = nil .
eq decreaseDeadlines((M with ARGS from O to O' deadline T1) :: MSGLIST, T)
 = (if T <= T1 then (M with ARGS from O to O' deadline T1 - T) else none fi)
   decreaseDeadlines(MSGLIST, T) .
```

4 Formal Analysis of Timed Rebeca Models

We have automated the translation of Timed Rebeca models to Real-Time Maude. The translator is currently being integrated into RMC (Rebeca Model Checker) [15] to support Real-Time Maude simulation, reachability analysis, and untimed LTL and timed CTL model checking of Timed Rebeca models from within the Rebeca toolset. To allow the Timed Rebeca modeler to define his/her LTL and TCTL formulas without having to know anything about the Real-Time Maude representation of his/her model, and without having to know how to define atomic state propositions in Real-Time Maude, we have predefined a number of useful generic atomic propositions. LTL and TCTL formulas can then be defined using these propositions and the usual logical operators such as `~` (not), `/\` (conjunction), etc., linear temporal logic operators such as `[]` (always), `<>` (eventually), etc., and timed CTL operators such as `AG` (always), `AF[<= than` *t*`]` (always reachable within time *t*), etc.

We have defined atomic propositions on the *state variables* of the rebecs. The value of the state variables can be compared to the constants of the same type using common relational operators `is` (equality), `<=`, etc. For example, the proposition *variable* `of` *rebec* `<=` *value* holds if the current value of the state variable *variable* in the rebec *rebec* is less than or equal to *value*:

```
ops _of_is_  _of_<=_  _of_<_ ... : IntVar Oid Int -> Prop .
```

```
eq {CONF < O : Rebec | stateVars : (V |-> I) VAL >} |= V of O is J = I == J .
eq {CONF < O : Rebec | stateVars : (V |-> I) VAL >} |= V of O <= J = I <= J .
```

As an example, `temp of "h" <= 30` is true if the `temp` state variable of the rebec `h` is less than or equal to 30.

Likewise, we have defined generic propositions o `hasSent` m `to` o', denoting that rebec o has sent a message with header m to the rebec o' and that the message is still in the network; and o `hasReceived` m `from` o' (the message with header m is already in o's queue), and the more generic o `hasReceived` m:

```
ops _hasSent_to_  _hasReceived_from_ : Oid MsgHeader Oid -> Prop .
eq {CONF dly((MN with VAL from O to O' deadline T), T')}
    |= O hasSent MN to O' = true .
eq {CONF  < O : Rebec | queue : ML1 :: (MN with VAL from O' to O deadline T) :: ML2 >}
    |= O hasReceived MN from O' = true .
op _hasReceived_ : Oid MsgHeader -> Prop [ctor] .
eq {CONF  < O : Rebec | queue : ML1 :: (MN with VAL from O' to O deadline T) :: ML2 >}
    |= O hasReceived MN = true .
```

We can now easily define temporal logic properties of our Timed Rebeca models:

```
[] ((temp of "t" >= 30) -> <> (on of "h" is false))
```

5 Case Study: A Collision Avoidance Protocol

This section illustrates our modeling and verification methodology on the IEEE 802.11 RTS/CTS protocol for collision avoidance in wireless networks [8]. When a node decides to send data to another node, it sends a *Request to Send* (RTS) message to the destination node, which is expected to reply with a *Clear to Send* (CTS) message. Other nodes in the network which receive RTS or CTS messages wait for a certain amount of time, making the medium free for the two communicating nodes. This mechanism also solves the *hidden node problem*, which occurs when two nodes want to send data to the same node. The destination node is in the range of both senders, but the senders are out of the range of each other (hence, unaware of each other's decision to send a message). In the protocol, the destination node sends a CTS message to only one of the senders. The other sender waits for a random amount of time, and then sends an RTS message to the destination node. Furthermore, this protocol solves the *exposed node problem* as well, where two adjacent nodes send data to two different destination nodes, so that the interference of data transfer of adjacent senders results in message collision. The problem is solved by preventing the senders from sending data after receiving the CTS message from other sender nodes.

We have analyzed the following properties of our Timed Rebeca model:

– *Collision freedom:* there are not data messages from two different senders at the same time.
– *Starvation avoidance:* A node that wants to send data to any destination will eventually be able to do so.

– *Delivery time bound:* The must be an upper time bound on the data transfer to a node that is not in the radio transmission range of the sender; this time bound depends on the network topology and delays.

Our model uses the two reactive classes `Node` and `RadioTransfer`. Each `Node` knows a `RadioTransfer` rebec, which is responsible for broadcasting its messages to all nodes in the node's transmission range. To transmit data, the sender sends an RTS message to the receiver (through its `RadioTransfer` rebec) and waits for its response. When an RTS message is delivered, the receiver checks wether the network is busy. In that case, it sends an RTS message to itself after a random `backOff` (modeled by a non-deterministic choice among the values $\{2, 3, 4\}$). If the receiver is not the target of the message, it mark the status of the network as busy. Otherwise, it sends a CTS message to the sender. Receiving an RTS message is handled by the following message server:

```
msgsrv rcvRTS(byte sndr, byte rcvr) {
   if (rcvr == id)
       if (channleIdle)  radioTransfer.passCTS(id, sndr);
       else  self.rcvRTS(sndr,rcvr) after(backOff);
   else
       channleIdle = false;
}
```

When a node receives a CTS message, it checks whether it is the target of the message. If so, it sends its data. If not, it sets the network status to idle:

```
msgsrv rcvCTS(byte sndr,byte rcvr) {
   if (rcvr == id)  self.sendData();
   else  channelIdle = true;
}
```

We have performed the analysis on a 2.53 GHz Intel processor with 2GB RAM, running Ubuntu 10.10. The case examined has four nodes in a ring topology (each node has two adjacent nodes in its communication range). We have analyzed different transmission topologies to also analyze the hidden node and the exposed node problems.

To verify collision freedom, we must ensure that no two messages with different senders exist in radio transfer range, which can be verified for all behaviors up to time 1000 using the following model checking command:

```
(mc initState |=t
  [] ~ (  (("node1" hasSent passData) /\ ("node2" hasSent passData))
       \/ (("node2" hasSent passData) /\ ("node3" hasSent passData))
       \/ (("node3" hasSent passData) /\ ("node4" hasSent passData))
       \/ (("node4" hasSent passData) /\ ("node1" hasSent passData)) )
  in time <= 1000 .)
```

The model checking result reported by Real-Time Maude in 5 minutes was `true`.

To analyze starvation freedom we use the following command, which states that each node will eventually (within time 1000) be able to send a data message:

```
(mc initState |=t (<> ("node1" hasSent passData to "radioTransfer1")) /\
    (<> ("node2" hasSent passData to "radioTransfer2")) /\
    (<> ("node3" hasSent passData to "radioTransfer3")) /\
    (<> ("node4" hasSent passData to "radioTransfer4")) in time <= 1000 .)
```

This model checking command returns a counterexample, since the protocol suffers from starvation.

To analyze whether the upper time bound for a transmission from node 1 to node 3 via node 2 is less than t, we can use the TCTL formula $\forall\Box(s_{12} \rightarrow \forall\Diamond^{\leq t} r_{32})$, where s_{12} is true if node 1 has just sent a message to node 2, and r_{32} is true if node 3 has just received the message from node 2. This can be verified using the following command for $t = 6$:

```
(mc-tctl initState |= AG ( ("node1" hasSent passData to "node2") implies
         (AF[<= than 6] ("node3" hasRcv rcvData from "node2")) ) .)
```

The protocol fails to satisfy this property, because of the starvation. But changing `AF` to `EF` makes the property hold; i.e., for all possible behaviors from the initial state, there exists a path where the transmission can take place in less than 6 time units. Model checking this property took about 20 hours.

6 Related Work

Timed Actor Models. Although there are some actor-based modeling languages for real-time systems, their lack of effective analysis tools is a significant obstacle to applying formal verification to real systems. In some cases, assertion-based verification is suggested to analyze invariance and other safety properties. However, there is need for more general verification methods, such as model checking liveness properties and other (timed or untimed) temporal logic properties.

One real-time actor-based modeling language is RTSynchronizer [16]. The formalism specifies the model in terms of a number of actors and a global synchronizer which simulates the timed behavior of the actors. Each actor is extended with timing assumptions which are used by the synchronizer to figure out the ready-to-execute messages of the actor. In contrast with the "pure" actor language Timed Rebeca, the computation in RTSynchronizer takes place through interactions between the synchronizer and the actors. RTSynchronizer provides limited verification by placing the desired invariant properties in the body of the actors, but this approach does not support the model checking of more general temporal logic properties. (See also below.)

Creol is an actor-based language for modeling concurrent objects enriched by synchronization patterns and type system [4]. Jaghouri et al. add timing features to Creol in [5], where they also develop a schedulability analysis technique, but, again, there is no support for temporal logic verification of such models.

Work on Timed Rebeca. Aceto et al. in [1] suggested a mapping from Timed Rebeca models to Erlang for simulation (but not further formal analysis) purposes. A semantics based on *floating-time transition system* was recently proposed for

Timed Rebeca [9]. Schedulability and deadlock-freedom can be checked efficiently using this semantics, but no state-based property can be verified.

Real-Time Maude as a Semantic Framework. Because of its expressiveness and natural model for object-oriented distributed real-time systems, Real-Time Maude has proved to be a suitable semantic framework in which a number of formal modeling languages have been given a formal semantics. Examples of such modeling languages include Ptolemy II discrete-event models, the Orc web orchestration language, subsets and synchronous versions of the avionics modeling standard AADL, timed model transformation frameworks, and so on (see [14] for an overview). However, the only work on Real-Time Maude semantics for timed actor languages is the work by Ding et al. [7] on the above-mentioned quite different RTSynchronizer model. Unfortunately, no details about the Real-Time Maude semantics are given in [7], and it seems that their work does not define the semantics for the entire language, but only for the case study of a Simplex architecture modeled using RTSynchronizer. Furthermore, no attempts at temporal logic model checking was performed in [7].

7 Conclusion

Using Real-Time Maude, we have defined the first executable formal semantics of Timed Rebeca. This enables a wide range of formal analysis methods for Timed Rebeca models, including simulation, reachability analysis, and both timed and untimed temporal logic model checking. We have integrated such Real-Time Maude analysis of Timed Rebeca models into the Rebeca toolset, and have defined a number of useful atomic propositions, allowing the Timed Rebeca user to define her desired properties without knowing Real-Time Maude. We illustrated such verification of Timed Rebeca models on a collision avoidance protocol.

Since Timed Rebeca, with its Java-like syntax and simple and intuitive actor-based communication model, should be easy to learn and use for people unfamiliar with formal methods, our work bridges the gap between practitioners and formal methods, since it enables a model-engineering methodology that combines the convenience of Timed Rebeca modeling with powerful formal analysis in Real-Time Maude.

We have focused on providing a clean and intuitive semantics. If states encountered *during* the execution of a message server do not matter for the properties we are interested in, we could significantly optimize the semantics by executing together, in one step, all the statements in a message server. This would significantly reduce the number of interleavings and would drastically improve the model checking performance. Finally, although the counterexamples from the Real-Time Maude analyses should be fairly easy to understand, we should nevertheless provide them in terms of the Timed Rebeca model.

References

1. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: Proc. FOCLASA'11. EPTCS, vol. 58 (2011)
2. Agha, G.: ACTORS – a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
3. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. Science of Computer Programming 77(12), 1235–1271 (2012)
4. Bjørk, J., Johnsen, E.B., Owe, O., Schlatte, R.: Lightweight time modeling in Timed Creol. In: Proc. RTRTS'10. EPTCS, vol. 36 (2010)
5. de Boer, F.S., Chothia, T., Jaghoori, M.M.: Modular schedulability analysis of concurrent objects in Creol. In: Proc. FSEN'09. LNCS, vol. 5961. Springer (2009)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
7. Ding, H., Zheng, C., Agha, G., Sha, L.: Automated verification of the dependability of object-oriented real-time systems. In: Proc. WORDS Fall. IEEE (2003)
8. IEEE Standard for Information Technology - Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY). IEEE Std 802.11e-2005 (Amendment to IEEE Std 802.11, 1999 Edition (Reaff 2003)) (2005)
9. Khamespanah, E., Sabahi, Z., Khosravi, R., Sirjani, M., Izadi, M.: Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In: AGERE!'12, SPLASH Workshops. ACM (2012)
10. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Timed CTL model checking in real-time maude. In: Proc. WRLA'12. LNCS, vol. 7571. Springer (2012)
11. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
12. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
13. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Proc. TACAS'08. LNCS, vol. 4963. Springer (2008)
14. Ölveczky, P.C.: Semantics, simulation, and formal analysis of modeling languages for embedded systems in Real-Time Maude. In: Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000. Springer (2011)
15. Rebeca Language Home Page. `http://www.rebeca-lang.org`
16. Ren, S., Agha, G.: RTsynchronizer: Language support for real-time specifications in distributed systems. In: Proc. LCT-RTS'95. ACM (1995)
17. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. 63(4), 385–410 (2004)