Real-Time Maude and Its Applications

Peter Csaba Ölveczky(✉)

University of Oslo, Oslo, Norway
peterol@ifi.uio.no

Abstract. Real-Time Maude extends the rewriting-logic-based Maude system to support the executable formal modeling and analysis of real-time systems. Real-Time Maude is characterized by its general and expressive, yet intuitive, specification formalism, and offers a spectrum of formal analysis methods, including: rewriting for simulation purposes, search for reachability analysis, and both untimed and metric temporal logic model checking. Real-Time Maude is particularly suitable for specifying real-time systems in an object-oriented style, and its flexible formalism makes it easy to model different forms of communication.

This modeling flexibility, and the usefulness of both Real-Time Maude simulation and model checking, has been demonstrated in many advanced state-of-the-art applications, including both distributed protocols of different kinds and industrial embedded systems. Furthermore, Real-Time Maude's expressiveness has also been exploited to define the formal semantics of a number of modeling languages for real-time/embedded systems. Real-Time Maude thereby provides formal analysis for these languages for free, and such analysis has been integrated into the tool environment of a number of modeling languages.

This paper gives an informal overview of Real-Time Maude and some of its applications.

1 Introduction

Real-Time Maude is a rewriting-logic-based formal specification language and simulation and model checking tool that extends Maude to support the formal modeling and analysis of real-time systems. Being an extension of Maude, Real-Time Maude inherits Maude's key features:

- a simple and intuitive formalism;
- expressiveness and generality; and
- providing a natural model of object-based real-time systems.

In Real-Time Maude the data types are user-defined as an algebraic equational specification; that is, the user declares her sorts and functions on those sorts. Some functions are "constructors" that together define the "elements" of the sorts, and the other functions are defined by (first-order) conditional equations. Local transitions that are assumed to take zero time are specified by (possibly conditional) *rewrite rules* of the form `crl [`l`] :` t `=>` t' `if` *cond*, where t

S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 42–79, 2014.
DOI: 10.1007/978-3-319-12904-4_3

and t' are two terms, possibly containing variables, denoting (sets of) local state fragments. Finally, time elapse is modeled explicitly using *tick rewrite rules* of the form `crl [`l`]: {`t`} => {`t'`} in time` Δ `if` *cond*; this ensures uniform time elapse in all parts of the system, since the *entire state* should have the form `{`u`}`. The duration of the transition is given by the term Δ, which may contain variables, including variables not appearing in t. The time domain can be discrete or dense.

Some key features of Real-Time Maude, which distinguish it from other real-time formalisms and formal analysis tools, include:

- Expressiveness and flexibility. Any computable data type can be defined as an algebraic specification [18]. In particular, we may have "advanced" functions, unbounded data structures, etc. Likewise, the rewrite rules can define very sophisticated transition patterns.
- Object-based distributed real-time systems can be naturally modeled in Real-Time Maude, including features such as
  - dynamic object creation and deletion, and
  - *hierarchical* objects, which may contain entire dynamic subsystems.
- Real-Time Maude is not based on a fixed model of communication, into which other models of communication have to be encoded. Instead, the desired form of communication can be specified directly in the logic.
- Simple and intuitive formalism. Both static, dynamic, and real-time aspects are specified in a simple and intuitive framework (equations and rewrite rules), that should make Real-Time Maude a low-threshold tool for developers with limited formal methods experience.
- A range of automatic formal analyses, including simulation and different kinds of timed and untimed temporal logic model checking.
- The possibility of defining *parametrized* atomic propositions.

Real-Time Maude provides a number of automated explicit-state analyses, including:

- timed rewriting for simulation,
- timed reachability analysis,
- untimed—but possibly time-bounded—LTL model checking, and
- timed CTL (TCTL) model checking.

The price to pay for this modeling convenience is that key system properties are *undecidable*: the above analysis methods are *in general* not sound and complete.

Real-Time Maude should be seen as complementing the highly successful timed automaton formalism [6]—which is a fairly restricted formalism to ensure that key system properties are always decidable—and its equally successful tools such as Uppaal [14] (and RED [63]), by focussing on expressiveness and modeling convenience. Real-Time Maude can also be seen to complement languages such as IF [21] and BIP [13] by having (essentially) a single formalism instead of being composed of three fairly different formalisms for three different aspects, by supporting the specification of any data type in a logic instead of in Java, by not being based on a fixed communication model, and by supporting the dynamic creation and deletion of (possibly hierarchical) objects.

The key question concerning Real-Time Maude is of course whether all these features are needed or useful. That is:

> **Question 1.** Are there interesting systems where the above features of Real-Time Maude are needed/useful, and where meaningful Real-Time Maude analysis is still possible?

Since Real-Time Maude analyses are in general not sound and complete, an important part of answering **Question 1** is to answer the following question:

> **Question 2.** Are there interesting classes of systems for which Real-Time Maude analyses are guaranteed to be sound and complete?

The goal of this paper is to briefly and informally introduce Real-Time Maude and its applications. In particular, Sect. 2 introduces modeling in Real-Time Maude. Section 3 explains how Real-Time Maude specifications are executed, and gives an overview of the tool's analysis features. Section 4 addresses **Question 2** by presenting classes of systems for which LTL and TCTL model checking is indeed sound and complete. The main part of the paper is Sect. 5, which summarizes some applications of Real-Time Maude. Section 6 discusses extensions of Real-Time Maude, and Sect. 7 gives some concluding remarks.

The formal treatment of Real-Time Maude and its semantics is given in [49]; the underlying real-time rewrite theory model is presented in [47]; and early summaries of some of the uses of Real-Time Maude to define the semantics of modeling languages were presented in [43,44].

Finally, Real-Time Maude is a Maude program that is available free of charge at http://ifi.uio.no/RealTimeMaude.

2 Specification in Real-Time Maude

A Real-Time Maude module `tmod` M `is` (Σ, E, IR, TR) `endtm` specifies a real-time rewrite theory [47], where Σ is an algebraic signature declaring sorts (keywords `sort` and `sorts`) subsorts (`subsort`), and function symbols. A function declaration has the form `op` f `:` s_1 `...` s_n `->` s `[`*atts*`]`, which declares a function f with n arguments of sorts $s_1, \ldots, s_n$, respectively, that gives an element of sort s. The optional set *atts* of function attributes could declare f to be a *constructor* symbol that constructs the elements of the sort s, or could declare the function—in case it is a binary function—to be *associative* (`assoc`), *commutative* (`comm`), and/or to have an identity element, or declare that f is a *frozen* operator, so that rewrites cannot take place in its subterms. E is a set of (possibly conditional) equations of the form `eq` t `=` t' and `ceq` t `=` t' `if` *cond*; the terms t and t' could contain mathematical variables, which are declared with the keyword `var` or `vars`. IR is a set of declarations of *instantaneous* rewrite rules `rl [`l`] :` t `=>` t' and `crl [`l`] :` t `=>` t' `if` *cond*, where l is a label; such rules define local transitions that are assumed to take zero time. Finally, TR is a set of *tick* *(rewrite) rules* of the form `rl` `[`l`]` `:` `{`t`}` `=>` `{`t'`}` `in time` u and `crl` `[`l`]` `:` `{`t`}` `=>` `{`t'`}` `in time` Δ `if` *cond* that are used to model time advance in the system.

The equational specification (Σ, E) must contain a specification of a time domain, which may be dense or discrete. Real-Time Maude has predefined useful time domains such as `NAT-TIME-DOMAIN` (unbounded natural numbers) and `POSRAT-TIME-DOMAIN` (unbounded nonnegative rational numbers), and their extensions `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` that add a supersort `TimeInf` of `Time` with an “infinity” element `INF`.

The global state of the system must always have the form `{`u`}`, where u is a term of sort `System`. The form of the tick rule then ensures that time advances uniformly in all parts of the system.

We illustrate specification in Real-Time Maude with a small example borrowed from [49].

Example 1. The following module models a *“retrograde” clock* with a dense time domain. The clock may be running (in which case the system is in state `{clock(`r`)}` for r the time shown by the clock) or may have stopped (in which case the system is in state `{stopped-clock(`r`)}` for r the clock value when it stopped). When the clock shows `24` it must be reset to `0` immediately:

```
(tmod DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
  crl [tickWhenRunning] :
      {clock(R)} => {clock(R + R')} in time R' if R' <= 24 - R [nonexec] .
  rl [tickWhenStopped] :
     {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
  rl [reset] :   clock(24) => clock(0) .
  rl [batteryDies] :   clock(R) => stopped-clock(R) .
endtm)
```

The two tick rules model the effect of time elapse on a system by increasing the clock value of a running clock according to the time elapsed, and by leaving a stopped clock unchanged. The rules `reset` and `batteryDies` are instantaneous rules modeling events that take zero time.

To specify *all possible behaviors* in a dense time domain, the duration of the tick rules is given by a variable (`R'`) that is not present in the lefthand sides of the rules. This means that time may elapse by *any* amount less than $24 - r$ from a state `{clock(`r`)}`, and by any amount from a state `{stopped-clock(`r`)}`.

Tick rules—especially in dense time domains—typically have the forms

```
 rl [l] : {t} => {t'} in time x .
crl [l] : {t} => {t'} in time x  if cond .
crl [l] : {t} => {t'} in time x  if x <= u .
crl [l] : {t} => {t'} in time x  if x <= u /\ cond .
```

where x is a variable that does not appear in t and is not instantiated in the condition. Section 3 explains how such tick rules are executed.

2.1 Object-Oriented Specifications

In Real-Time Maude, we can declare *classes* in object-oriented timed modules (`(tomod` M `is` ... `endtom)`). A class declaration

class C | att_1 : s_1, ... , att_n : s_1 .

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n, respectively. An *object* of class C in a given state is represented as a term

< O : C | att_1 : val_1, ..., att_n : val_n >

of sort `Object`, 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. A *message* m with parameters $p_1, \ldots, p_n$ of sorts $s_1, \ldots, s_n$ can be represented as a term $m(p_1, \ldots, p_n)$ of sort `Msg`; such messages are declared

msg m : s_1 ... s_n -> Msg .

A *configuration* is term of the sort `Configuration`, and is a *multiset* of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. Since a class attribute may have sort `Configuration`, we can have *hierarchical* objects which contain a subconfiguration of other (possibly hierarchical) objects and messages.

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 >
          m'(O',x) .
```

defines a parameterized family of transitions (one for each substitution instance) in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated. The message `m(O,w)` is *removed* from the state by the rule, since it does *not* occur in the right-hand side of the rule. Likewise, the message `m'(O',x)` is *generated* by the rule, since it *only* occurs in the right-hand side of the rule. Attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3`, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as `a2`, can be omitted from right-hand sides of rules.

A *subclass* inherits all the attributes and rules of its superclasses.

Messaging delay can be modeled by sending a "delayed" message `dly(`m, d`)`, where d is the remaining delay of the message. When the remaining delay is 0, the message is "ripe" and becomes m. The `dly` wrapper can be declared as follows (the sort `NEConfiguration` denotes non-empty configurations):

```
sort DlyMsg .
subsorts Msg < DlyMsg < NEConfiguration .

op dly : Msg Time -> DlyMsg [ctor right id: 0] .
```

Most object-oriented Real-Time Maude specifications use the tick rule

```
var T : Time .   var SYSTEM : Configuration .

crl [tick] :
    {SYSTEM} => {timeEffect(SYSTEM,T)} in time T if T <= mte(SYSTEM) .
```

where

– the function `mte` defines the *m*aximum *t*ime that may *e*lapse before some (instantaneous) event must take place, and
– the function `timeEffect` defines how the passage of a certain amount of time affects the state of the system.

These functions typically distribute over the elements (objects and messages) in a configuration as follows:

```
vars T T1 T2 : Time .    vars C1 C2 : Configuration .   vars M : Msg .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .     --- infinity value
ceq mte(C1 C2) = min(mte(C1), mte(C2)) if C1 =/= none and C2 =/= none .

op timeEffect : Configuration Time -> Configuration [frozen (1)] .
eq timeEffect(none, T) = none .
ceq timeEffect(C1 C2, T) = timeEffect(C1, T)  timeEffect(C2, T)
    if C1 =/= none and C2 =/= none .

eq timeEffect(dly(M, T1), T2) = dly(M, T1 monus T2) .
```

where the built-in operator `monus` is defined by x `monus` $y = \max(x - y, 0)$. If a message must be read exactly when it becomes "ripe", i.e., when its remaining delay is 0, `mte` is defined as follows:

```
eq mte(dly(M, T)) = T .
```

However, if the message delay instead denotes the *minimum* delay, and the message can be read *at any time* after the messaging delay has expired, the following definition of `mte` on messages should be used instead:

```
eq mte(dly(M, T)) = INF .
```

To fully specify the timing behavior an object-oriented real-time system it is then enough to define `mte` and `timeEffect` on *single* objects, as illustrated in the following example, which is also borrowed from [49].

Example 2. We illustrate object-oriented specification with a simple protocol for computing *round trip times* (i.e., the time it takes for a message to travel from a given node to another node, and back) between pairs of nodes in a network.

The protocol is straightforward: A message `findRtt(`*initiator*, *neighbor*`)` kicks off the protocol and indicates that the node *initiator* wants to find the round trip time to the *neighbor* node. An initiator cannot participate in multiple runs of the protocol, as an initiator, at the same time.

The *initiator* node has a local clock and starts a run of the protocol by sending an `rttReq` message to its neighbor *neighbor* with its current time stamp r (rule `startSession`). When the neighbor receives the `rttReq` message, it replies with an `rttResp` message containing the received time stamp r (rule `rttResponse`). When the initiator reads the `rttResp` message with its original time stamp r, the rtt value is just its current clock value minus r (rule `treatRttResp`).

Since the transmission times might depend on factors such as network traffic, we assume that the messaging delay of a single message could be *any* value greater than or equal to `MIN-DELAY`. If the initiator does not receive a response in time less than `MAX-RTT`, it initiates another round of the protocol exactly time `MAX-RTT` after its first attempt (rule `tryAgain`). The process is repeated until an rtt value less than `MAX-RTT` is found. The rule `ignoreOldResp` ignores responses from earlier rounds of the protocol.

In the following specification, each `Node` object uses a `timer` attribute to ensure that a new attempt is initiated at every `MAX-RTT` time units, until an rtt value is found. If the timer has value r, it must "ring" in time r from the current time. The timer is turned off when its value is `INF`. The class `Node` has the attributes `nbr`, which denotes the node whose rtt value it is interested in (and is `noOid` otherwise), and a `clock` attribute denoting the value of its local clock. The `rtt` attribute stores the rtt to its neighbor:

```
(tomod RTT is protecting NAT-TIME-DOMAIN-WITH-INF .
  ops MIN-DELAY MAX-RTT : -> Time .
  eq MIN-DELAY = 1 .     eq MAX-RTT = 4 .

  op noOid : -> Oid [ctor] .     ---"null" object name

  class Node | clock : Time,  rtt : TimeInf,  nbr : Oid,  timer : TimeInf .

  msgs rttReq rttResp : Oid Oid Time -> Msg .
  msg  findRtt : Oid Oid -> Msg .                  --- start a run

  vars O O' : Oid .   vars R R' : Time .   var TI : TimeInf .

  --- start a session, and set timer:
  rl [startSession] :
     findRtt(O, O')
     < O : Node | clock : R >
    =>
     < O : Node | timer : MAX-RTT, nbr : O' >
     dly(rttReq(O', O, R), MIN-DELAY) .

  --- respond to request:
```

```
  rl [rttResponse] :
     rttReq(O, O', R)
     < O : Node | >
    =>
     < O : Node | >
     dly(rttResp(O', O, R), MIN-DELAY) .

  --- received resp within time MAX-RTT; record rtt value and turn off timer:
  crl [treatRttResp] :
      rttResp(O, O', R)
      < O : Node | clock : R' >
     =>
      < O : Node | rtt : (R' monus R), timer : INF >
    if (R' monus R) < MAX-RTT .

  --- ignore and discard too old message:
  crl [ignoreOldResp] :
      rttResp(O, O', R)
      < O : Node | clock : R' >
     =>
      < O : Node | >
    if (R' monus R) >= MAX-RTT .

  --- start new round and reset timer when timer expires:
  rl [tryAgain] :
     < O : Node | timer : 0, clock : R, nbr : O' >
    =>
     < O : Node | timer : MAX-RTT >
     dly(rttReq(O', O, R), MIN-DELAY) .

  ...    --- the tick rule, dly, mte, and timeEffect are defined as above
         --- and are not shown
  eq timeEffect(< O : Node | clock : R, timer : TI >, R') =
          < O : Node | clock : R + R', timer : TI monus R' > .

  eq mte(< O : Node | timer : TI >) = TI .
  eq mte(dly(M, T)) = INF .
endtom)
```

This use of timers, clocks, and the functions `mte` and `timeEffect` is fairly typical for object-oriented real-time specifications. The following timed module defines an initial state with three nodes `n1`, `n2`, and `n3`:

```
(tomod RTT-I is including RTT .
  ops n1 n2 n3 : -> Oid [ctor] .
  op initState : -> GlobalSystem .
  eq initState =
      {findRtt(n1, n2)   findRtt(n2, n3)   findRtt(n3, n1)
       < n1 : Node | clock : 0, timer : INF, nbr : noOid, rtt : INF >
       < n2 : Node | clock : 0, timer : INF, nbr : noOid, rtt : INF >
       < n3 : Node | clock : 0, timer : INF, nbr : noOid, rtt : INF >} .
endtom)
```

3 Formal Analysis

This section gives an overview of the formal analyses supported by the Real-Time Maude tool.

3.1 Time Sampling Strategies

Maude specifications are *executable* under reasonable conditions. However, as explained above, in dense time domains, tick rules will typically have the forms

```
crl [l] : {t} => {t'} in time x  if cond [nonexec] .
crl [l] : {t} => {t'} in time x  if x <= u /\ cond [nonexec] .
```

(where *cond* might be omitted), where x a variable not occurring in t and not initialized in *cond*, which allows any moment in time to be "visited." Such tick rules are not directly executable (`[nonexec]`). Timed automata "discretize" dense time by defining "clock regions," so that all states in the same clock region satisfy the same properties [6]. The clock region construction cannot be employed in the much more expressive Real-Time Maude formalism. The Real-Time Maude approach to such tick rules is to provide a set of *time sampling strategies* that define how to instantiate the variable x in the tick rules:

- The *default* time sampling strategy increases time by a user-given value.
- The *maximal* time sampling strategy advances time *as much as possible* (as given by u). If there is no bound on the time elapse, time is advanced by a user-given value.

The user selects her time sampling strategy by giving either the command `(set tick def` r `.)` or the command `(set tick max def` r `.)`.

All applications of time-nondeterministic tick rules—whether it is for rewriting, search, or model checking—are performed using the given time sampling strategy. This means that some behaviors in the system, namely those obtained by applying the tick rules differently, are not analyzed. The result of a Real-Time Maude analysis should be understood as being in general incomplete: counterexamples are true counterexamples, but (except for the case of discrete time when all states are visited) satisfaction of a property only shows that it holds for the states visited. Section 4 shows that Real-Time Maude analyses are, nevertheless, sound and complete for many interesting systems.

3.2 Analysis Commands

Simulation. The *timed rewrite* command

```
(tfrew t0 in time <= r .)
```

simulates *one* behavior of the system from initial state t_0 up to a total duration less than or equal to the `Time` value r. The time bound can also have the forms `in time <` r and `with no time limit`. Real-Time Maude's *tracing* facilities allow us to trace the steps in a timed rewrite sequence (see [42] for details).

Example 3. Before we can analyze our retrograde clock, we need to define a time sampling strategy. Since the clock may stop at any time, we use the time sampling strategy that increases time by one time unit in each tick rule application:

```
Maude> (set tick def 1 .)
```

We can then simulate one behavior of the clock system up to time 100:

```
Maude> (tfrew {clock(0)}  in time <= 100 .)

Result ClockedSystem :   {stopped-clock(0)} in time 100
```

Reachability Analysis. Explicit-state *timed search* can be used to analyze not just *one* behavior, but to analyze *all* behaviors from a given initial state, relative to the chosen time sampling strategy. The syntax variations of the timed search command—which is used to search for states which match a *search pattern* and which are reachable in a given time interval—are:

(tsearch t_0 *arrow* *pattern* with no time limit .)
(tsearch t_0 *arrow* *pattern* in time $\sim$ r .)
(tsearch t_0 *arrow* *pattern* in time-interval between $\sim'$ r and $\sim''$ r' .)

where t_0 is the initial state, *pattern* is either t or has the form t `such that` *cond*, for a term t and a semantic condition *cond*, $\sim$ is either `<`, `<=`, `>`, or `>=`, $\sim'$ is either `>=` or `>`, $\sim''$ is either `<=` or `<`, and r and r' are ground terms of sort `Time`. The *arrow* is either `=>1`, `=>*`, and `=>+`, which searches for states reachable from t_0 in, respectively, one, zero or more, and one or more rewrite steps. The arrow `=>!` is used to search for states which cannot be further rewritten. The search command can be parametrized by the number of solutions sought ((tsearch [n] ...)). As explained in Sect. 3.3, timed search maintains a "system clock" in the state. The set of reachable "timestamped states" will therefore be infinite even when the reachable state space is finite. Therefore, the *untimed search* command

(utsearch t_0 *arrow* *pattern* .)

which abstracts from the "system clock," can be used when the reachable state space is finite.

Example 4. We continue analyzing our retrograde clock with the time sampling strategy chosen above. Although the time domain is dense, the reachable state space (from `{clock(0)}`) should be finite when the time sampling strategy is taken into account. We check whether it is possible to reach a bad state where a running clock shows 25 or more:

```
Maude> (utsearch [1]
          {clock(0)} =>* {clock(T:Time)} such that T:Time >= 25 .)

No solution
```

Checking whether it is possible to reach a state where the running clock shows 1/2 also returns "`No solution`":

```
Maude> (utsearch [1] {clock(0)} =>* {clock(1/2)} .)

No solution
```

In this case, our time-sampling-based analysis is incorrect, since in the model it is indeed possible to reach the state `{clock(1/2)}`.

Example 5. The reachable state space from `initState` is infinite in our round trip time protocol, since the (local) clock values may grow beyond any bound and since the state may contain any number of old messages. Search should therefore be *time-bounded* to ensure termination. We set the time sampling strategy with the command `(set tick def 1 .)` to cover the discrete time domain.

The command

```
Maude> (tsearch [1]
           initState =>* {< O:Oid : Node | rtt : T:Time > REST:Configuration}
             such that T:Time > 4  in time <= 9 .)

No solution
```

then checks whether a state with a `rtt` value > 4 can be reached within time 9.

LTL Model Checking. Real-Time Maude extends Maude's explicit-state LTL model checker to timed modules. *Atomic propositions* are terms of sort `Prop`. A useful feature is the possibility to define *parametrized* atomic propositions $p(t_1, \ldots, t_n)$ as follows:

```
  op p : s1 ... sn -> Prop [ctor] .
```

The semantics of such state propositions are given by equations of the forms

$$\texttt{eq } \{\mathit{statePattern}\} \texttt{ |= } p(t_1, \ldots, t_n) \texttt{ = } b$$

and

$$\texttt{ceq } \{\mathit{statePattern}\} \texttt{ |= } p(t_1, \ldots, t_n) \texttt{ = } b \texttt{ if } \mathit{condition},$$

for b a term of sort `Bool`, which defines the state proposition $p(u_1, \ldots, u_n)$ to hold in all states $\{t\}$ where $\{t\}$ `|=` $p(u_1, \ldots, u_n)$ evaluates to `true`.

Real-Time Maude also supports the definition of "clocked" atomic propositions, whose semantics can depend on the elapsed time in the system:

$$\texttt{eq } \{\mathit{statePattern}\} \texttt{ in time } t \texttt{ |= } p(t_1, \ldots, t_n) \texttt{ = } b$$

and

$$\texttt{ceq } \{\mathit{statePattern}\} \texttt{ in time } t \texttt{ |= } p(t_1, \ldots, t_n) \texttt{ = } b \texttt{ if } \mathit{condition}.$$

An LTL formula is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). Then, the *unbounded* (resp., *time-bounded*) LTL model checking commands

`(mc` t_0 `|=u` φ `.)` and `(mc` t_0 `|=t` φ `in time <=` τ `.)`

check whether the formula φ holds in all behaviors from the initial state t_0 (resp., in all behaviors up to time τ). Only *time-bounded* model checking may involve clocked propositions. If the formula does not hold, the model checker returns a behavior that does not satisfy the formula.

The module `TIME-MODEL-CHECKER` should be imported by the modules defining the atomic propositions.

Example 6. The following module defines the *state* propositions `clock-dead` (which holds for all stopped clocks) and `clock-is(`r`)` (which holds if a *running* clock shows r), and the *clocked* proposition `clockEqualsTime` (which holds if the running clock shows the time elapsed in the system):

```
(tmod MODEL-CHECK-DENSE-CLOCK is including TIMED-MODEL-CHECKER .
  protecting DENSE-CLOCK .
  ops clock-dead clockEqualsTime : -> Prop [ctor] .
  op clock-is : Time -> Prop [ctor] .
  vars  R R' : Time .
  eq {stopped-clock(R)}      |=   clock-dead = true .
  eq {clock(R)}              |=   clock-is(R') = (R == R') .
  eq {clock(R)} in time R'   |=   clockEqualsTime = (R == R') .
endtm)
```

A natural correctness requirement is that the clock shows the elapsed time in the system, either until time 24 is reached or until the clock stops:

```
Maude> (mc {clock(0)} |=t clockEqualsTime U (clock-dead \/ clock-is(24))
            in time <= 50 .)

Result Bool :  true
```

Timed CTL Model Checking. Real-Time Maude has recently been equipped with an explicit-state *Timed CTL* (TCTL) model checker for *timed* temporal logic properties such as "every request will be followed by a response *within 100 ms*" or "the minimum time between two p-states is *10 s*" [34,35].

In TCTL, the temporal modalities are annotated with time intervals: $\Box_{\leq 5}\ \phi$, $\Diamond_{>8}\ \phi$, and $\phi_1\ \mathcal{U}_{[2,9]}\ \phi_2$, and so on. We refer to [35] for the full syntax of TCTL in Real-Time Maude. The model checker uses the logical operators `not`, `and`, `or`, `implies`, and so on. A formula $\forall\Box_{\leq 20}\ \phi$ is written `AG[<= than 20]` ϕ, a formula $\exists\Diamond_{>11}\ \phi$ is written `EF[> than 11]` ϕ, and a formula $\forall\phi_1\ \mathcal{U}_{[2,9]}\ \phi_2$ is written `A` ϕ_1 `U[c 2, 9 c]` ϕ_2.

The TCTL model checking has the syntax

(`mc-tctl` t_0 `|=` *formula* `.)`

At the moment, the model checker only gives a "yes/no" answer, and does not have a "time-bounded" version to deal with systems with infinite reachable state space. The module `TCTL-MODEL-CHECKER` should be imported when using the TCTL model checker.

Example 7. We check whether from any state where the running clock shows 5, it is possible to reach, in exactly 24 time units, a state where the running clock also shows 5:

```
Maude> (mc-tctl
       {clock(0)} |= AG (clock-is(5) implies EF[c 24,24 c] clock-is(5)) .)

Property satisfied
```

Other Commands. Real-Time Maude also provides commands for finding the shortest time it takes a desired state, and the longest time it takes to find the desired state for the first time.

3.3 Implementation

Real-Time Maude is implemented in Maude as an extension of Full Maude [22]. For efficiency purposes, all simulation, search, and LTL model checking commands have been implemented by transforming a Real-Time Maude module M, a time sampling strategy s, an initial state t_0, and a Real-Time Maude command C into a core Maude module M', an initial state t_0', and a Maude command C' that is then executed in Maude [49]. The results from Maude are then transformed back into suitable Real-Time Maude output.

The target Maude module M' is a function of *both* the Real-Time Maude command and the time sampling strategy used. For the *unbounded* search (`utsearch`) and LTL model checking (`(mc ... |=u ... .)`) commands, the transformation abstracts from the "system clock". Therefore, if the reachable state space is finite in the original model, it will remain so in the transformed Maude model M'. However, simulation and *time-bounded* search (`tsearch`) and LTL model checking (`(mc ... |=t ...)`) need to keep track of elapsed time; therefore, the states in the resulting Maude module M' have the form t `in time` r, where t is the state component, and r is the "time stamp" (or "system clock"). A single state t in M can then lead to multiple "time-stamped states" t `in time` r_1, t `in time` $r_2, \ldots$, in M', which leads to an infinite reachable timestamped state space; however, with an appropriate time sampling strategy, the set of such states reachable within a given upper time bound should be finite.

The other commands (find earliest/latest and TCTL model checking) cannot be transformed into Maude commands and are implemented directly using Maude's meta-level.

4 Sound and Complete Real-Time Maude Analysis

Section 3 explains that the time-sampling-based explicit-state reachability analysis and temporal logic model checking methods used by Real-Time Maude are in general not sound and complete, since only a subset of all possible behaviors are explored. Section 4.1 shows that maximal-time-sampling-based reachability and LTL model checking analyses *are* sound and complete for many interesting systems encountered in practice, and for which there were previously no such soundness/completeness results. Section 4.2 discusses the semantics of TCTL and soundness and completeness of TCTL model checking in Real-Time Maude.

4.1 Sound and Complete Reachability Analysis and LTL Model Checking in Real-Time Maude

Explicit-state model checking in a dense time domain cannot visit all moments in time. If the time domain is discrete, all behaviors can be explored by applying the time sampling strategy that advances time by the smallest time unit in each application of a tick rule. However, in many applications, this would be prohibitively expensive. For example, in the wireless sensor network algorithm analyzed in [52] and mentioned in Sect. 5, the time domain is *milliseconds*, while each round of the algorithm is *1000 s*. Most of the time the system is idling. Clearly, visiting each single moment in time yields a very inefficient simulation, and the large number of states encountered would make any model checking analysis unfeasible. Therefore, using maximal time sampling is in practice necessary also for many discrete-time systems.

An important question, posed as **Question 2** in the introduction, is whether there are interesting classes of systems for which maximal-time-sampling-based Real-Time Maude analysis is guaranteed to give the correct result. In [48], José Meseguer and I answer this question affirmatively by identifying classes of real-time rewrite theories for which maximal-time-sampling-based reachability analysis and LTL model checking are sound and complete analysis methods. In particular, we show that such analyses are sound and complete when:

- The system is *time-robust*, which means that no instantaneous action can take place after a tick step that does not advance time as much as possible.
- The atomic propositions appearing in the formula are *tick-invariant*, which means that their valuation is not affected by applying a tick rule.

More precisely, if $\mathcal{R}$ is a time-robust real-time rewrite theory, and $\mathcal{R}^{mts}$ is the theory obtained by applying the tick rules in $\mathcal{R}$ according to the maximal time sampling strategy, then for any CTL^* formula ϕ excluding the next-state operator $\bigcirc$ and involving only tick-invariant atomic propositions, we have

$$\mathcal{R}, t_0 \models \phi \qquad \text{if and only if} \qquad \mathcal{R}^{mts}, t_0 \models \phi.$$

The paper [48] also gives soundness and completeness results with somewhat weaker restrictions. Furthermore, it gives some easily checkable conditions for time-robustness of object-oriented Real-Time Maude specifications.

Although time-robustness and tick-invariance seem to be fairly restrictive conditions, many systems encountered in practice satisfy these conditions. For example, in many network protocols, actions are triggered either by the expiration of a timer or by the reception of a message with fixed delay; these systems are time-robust. Tick-invariance is almost always satisfied in practice, since the tick rules typically only affect timers and local clocks, which almost never affect the valuation of an atomic proposition.

Our retrograde clock is *not* time-robust, since the rule `batteryDies` can be applied at any time—including after non-maximal tick steps. Likewise, our round trip time example is not time-robust, since a message can be received at any time after its minimum delay has expired. If we replace the equation

```
eq mte(dly(M:Msg, T)) = INF .
```

with

```
eq mte(dly(M:Msg, T)) = T .
```

then the system is time-robust, since every message now has a fixed deadline (since time cannot advance when there is an unread ripe message in the state).

Among the large applications mentioned in Sect. 5, the OGDC wireless sensor network algorithm, the AER/NCA active network multicast protocol, Ptolemy II DE models, Timed Rebeca models, etc., are all time-robust. All of the these systems/languages go beyond the class of systems that can be captured by timed automata. Our results therefore yield sound and complete model checking procedures for dense-time systems beyond timed automata. On the other hand, the class of timed automata include many non-time-robust systems.

4.2 Sound and Complete TCTL Model Checking

In contrast to untimed LTL model checking, TCTL model checking using maximal time sampling is *not* sound and complete for time-robust real-time rewrite theories. Assume, for example, that from a state `{f(0)}`, the next event will take place after time 3; that is, we have the tick rule

```
crl [tick] : {f(T)}  => {f(T + T')} in time T' if T' <= 3 - T .
```

Consider the property $\exists \Diamond_{[1,2]}$ `true`; that is, it is possible to reach a state in some time $t \in [1, 2]$ where `true` holds. This property holds from initial state `{f(0)}`, since there is a non-maximal tick step to `{f(1)}` in time 1, and `true` holds in `{f(1)}`. However, maximal time sampling would rewrite `{f(0)}` in one step to `{f(3)}`, and hence the above property would *not* be satisfied.

Now consider the property $\forall \Diamond_{[1,2]}$ `true`; that is, in *all* behaviors we can reach a state some time in the time interval $[1, 2]$. Does this property hold in our specification? If we only consider all possible behaviors in the system, this property clearly does *not* hold, since time can advance from `{f(0)}` to `{f(3)}` in *one* tick step that advances time by 3 time units. However, it may be natural to understand the above tick rule as modeling a "continuous process" from `{f(0)}` to `{f(3)}`. In that case, this second property should hold.

These two ways of interpreting a real-time rewrite theory are called, respectively, the *pointwise* semantics and the *continuous* semantics.

In [35], Daniela Lepri, Erika Ábrahám, and I show that we can obtain sound and complete TCTL model checking of time-robust theories in *both* the pointwise and the continuous semantics, by advancing time by a value r_0 in each tick step. This value r_0 is "half the greatest common divisor" of the following values:

- all the tick durations obtained by applying the tick rules according to the maximal time sampling strategy; and
- all the non-zero time bounds occurring in the TCTL formula being analyzed.

Such TCTL model checking can be performed by giving the command

(`mc-tctl-gcd` t_0 `|=` ϕ `.)`

5 Some Real-Time Maude Applications

This section gives a brief overview of some Real-Time Maude applications, thereby answering **Question 1** in the introduction: are there interesting systems where the expressiveness and the more advanced modeling features of Real-Time Maude are needed, and where Real-Time Maude analysis still can provide useful results? In particular, Sect. 5.1 summarizes some "concrete" applications of Real-Time Maude. Real-Time Maude has been particularly useful to provide a formal semantics and formal analysis capabilities to domain-specific modeling languages for real-time systems; such applications are discussed in Sect. 5.3. Other applications of Real-Time Maude are mentioned in Sects. 5.2 and 5.4.

5.1 Some Concrete Applications

This section gives an overview of some "concrete" applications of Real-Time Maude.

Wireless Sensor Network Algorithms. OGDC [65] is a sophisticated state-of-the-art density control algorithm for wireless sensor networks developed at UIUC. The goal of a density control algorithm is to maximize the lifetime of a wireless sensor network by periodically turning nodes on and off while maintaining coverage of the entire area. The OGDC algorithm is based on always trying to turn on the "best placed" node, w.r.t. nodes that are already turned on, next. The OGDC developers used the ns-2 network simulator with a wireless extension to show that OGDC outperforms other density control algorithms.

Wireless sensor networks pose(d) many challenges to formal methods, including new forms of communication (area broadcast with delays, possibly sent with different signal strength), the need to analyze both performance and correctness, and so on. OGDC adds the need to deal with geometrical areas, angles, amount of overlap between multiple coverage areas, and so on.

Thorvaldsen and I model and analyze OGDC in [52]. The key Real-Time Maude features were the ability to easily define the new form of communication, defining data types for geometrical areas, and defining complex functions on such areas. To the best of our knowledge, the Real-Time Maude analysis of OGDC was the first formal analysis of an advanced wireless sensor network algorithm.

We performed a series of simulations of OGDC with up to 800 sensor nodes. The Real-Time Maude simulations gave performance figures very similar to the ns-2 simulations when we did *not* consider transmission delays. Since the OGDC developers did not include delays in their simulations, this indicates that *Real-Time Maude simulations provide quite accurate performance estimates of OGDC*. However, messaging delays play a crucial role in the OGDC algorithm. Real-Time Maude simulations *with delays* showed that the performance of OGDC is actually more than twice as bad as in the ns-2 simulations. Furthermore, we found a significant flaw in OGDC that explains its bad performance.

The techniques in [52] were then used and extended by Katelman, Meseguer, and Hou in their definition of the LMST topology control algorithm for wireless sensor networks in [31]. The goal is to minimize power consumption by adjusting the broadcast signal strength while maintaining network connectivity. Real-Time Maude was used to verify network connectivity by model checking a number of 4-node configurations. Their Real-Time Maude model was then extended with a number of probabilistic "implementation" features, such as quartz clock drift and 802.11 MAC contention, and the resulting probabilistic rewrite theory was subjected to *statistical model checking* using the VeStA [59] tool.

Mobile Ad Hoc Networks. Wireless mobile ad hoc networks (MANETs) combine wireless communication with node mobility. However, it is quite challenging to provide realistic models of mobility combined with wireless communication, since both the sender and a potential receiver may move—possible into or out of transmission range—*during* the "transmission time." In [37], Si Liu, José Meseguer, and I define a framework for modeling popular node mobility models together with wireless communication in Real-Time Maude.

In [38], we used our framework to model and analyze the well known leader election protocol for MANETs by Vasudevan, Kurose, and Towsley [62]. Our more detailed and flexible model of MANETs allowed us to study the protocol under various mobility and communication scenarios, including unidirectional communication links resulting from transmitting with different signal strength.

Scheduling Algorithms. The CASH algorithm is a sophisticated state-of-the-art scheduling algorithm developed by Marco Caccamo at UIUC. The idea is that tasks that do not need all of their allocated CPU time can put the unused budgets into a queue, so that other tasks can use that CPU time for improved system performance. Caccamo and I used Real-Time Maude to model and analyze a proposed optimization of CASH [46]. Real-Time Maude simulation showed that the queue of unused budgets can grow beyond any bound; since unbounded data structures are needed, the algorithm cannot be modeled by, e.g., timed automata. Real-Time Maude search uncovered a subtle behavior in the proposed optimization that led to missed hard deadlines. Furthermore, by using a pseudo-random function, we could generate tasks with "random" arrival and execution times, and use rewriting to perform "Monte-Carlo simulations." Extensive such simulation indicated that it is unlikely that the missed deadline could be found by simulation alone.

Prabhakar, Liu, and I have shown in [51] how resource-sharing algorithms, such as the priority inheritance and the priority ceiling protocol, can be formalized and analyzed using Real-Time Maude.

Embedded Car Software. Real-Time Maude has been used by a Japanese research institute to find several time-dependent bugs in embedded car software used by major car makers. The time sampling approach of Real-Time Maude was supposedly crucial to detect the bugs, which could not be found by the usual model-checking tools employed in industry.

Timing Features in AUTOSAR OS. *AUTOSAR* (AUTomotive Open System ARchitecture) is automotive open system architecture standard intended to unify and standardize automotive software development methodologies. The core group defining AUTOSAR includes BMW, Bosch, Daimler, General Motors, Toyota, and Volkswagen. In [66], Longfei Zhu and others use Real-Time Maude to formalize and analyze a number of timing properties in a part of the AUTOSAR operating system. In particular, many tasks are scheduled by the OS on a specific electronic control unit (ECU): tasks with different priorities as well as interrupts that must be handled. Zhu *et al.* model the task scheduling in AUTOSAR OS and use Real-Time Maude to analyze the following properties:

– schedulability, by searching for a task that misses its deadline;
– non-fault-propagation: other tasks should not miss their deadlines if the execution time of one task is longer than expected; and
– consistent configuration of components.

Google's Megastore and Its Extension. Cloud systems need to replicate data to ensure scalability and high availability. Unfortunately, combining wide-area replication with data consistency is quite hard. Some applications, such as Facebook and online newspapers, can tolerate low levels of consistency. However, to be able to use a cloud infrastructure also for consistency-critical applications such as stock exchange systems, online auctions, and banking and medical systems, replicated data stores must provide *transactions*. Megastore, developed at Google and used for, e.g., Gmail, Android Market, Google+, and Google AppEngine, is one of very few data stores that provide transactions. The problem is that the only publicly available description of Megastore is short and informal.

To facilitate the widespread study, adoption, and further development of Megastore's novel approach to transactions on replicated data, a much more detailed and precise description is needed.

In [29], Jon Grov and I develop a fairly detailed Real-Time Maude model of Megastore consisting of 56 rewrite rules. Since our starting point was a brief and informal overview paper, we had to in essence develop our own version of Megastore. We used Real-Time Maude simulation and model checking extensively throughout our development of this very complex system to improve our model to the point where we could not find any flaws during model checking. One Real-Time Maude feature that made this work possible was the ability to define complex atomic propositions; this allowed us to model check the serializability property of distributed concurrent transactions (as well as data consistency).

Megastore combines high performance, availability, and consistency by partitioning data into *entity groups*, and only guarantees data consistency if each transaction only accesses data from a single entity group. Grov and I define in [30] Megastore-CGC, an extension of Megastore that provides consistency also for transactions accessing data from multiple entity groups, thereby increasing the applicability of such cloud data stores. Megastore-CGC achieves this extra consistency without introducing significant additional message exchanges. We use Real-Time Maude to verify key properties, but also to compare the performance of Megastore with that of Megastore-CGC.

Avionics Systems. To smoothly turn an airplane, the airplane's ailerons and its rudder need to move in a synchronized way. (An aileron is a flap attached to the end of each wing, and a rudder is a flap attached to the plane's vertical tail.) A *turning algorithm* takes the desired next direction from the pilot as input, and should give commands to the aileron and rudder controllers to achieve a smooth turn in the desired direction.

In [10], Kyungmin Bae, Joshua Krisiloff, José Meseguer, and I formalize and analyze in Real-Time Maude a textbook turning algorithm for smaller aircrafts. Real-Time Maude simulations revealed that the turning algorithm failed to ensure a smooth turn: the (undesired) *adverse yaw* can become greater than $1.5°$ when the pilot gives a sharp turn command. We then modified the turning algorithm, and verified using Real-Time Maude model checking that, with the new turning algorithm, the plane will reach the desired direction fairly quickly and that the adverse yaw angle is less than $1.0°$ throughout the turn.

José Meseguer and I analyze a different avionics system in [39]: the *active standby* system developed by Steve Miller and Darren Cofer at Rockwell-Collins. In integrated modular avionics (IMA), a *cabinet* is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. There are always two or more cabinets that are physically separated on the aircraft so that physical damage does not take out the computer system. The active standby system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. Each side could fail, and can recover after failure. In case one side fails, the non-failed side should be the active side. In addition, the pilot can toggle the active status of these sides. The architecture of the system is shown in Fig. 1. The active standby system is *virtually synchronous*: it proceeds in rounds, and in each round, the components get an input in all their input channels (depicted as arrows in Fig. 1). LTL model checking showed that the desired properties were not satisfied. However, we could verify weakened versions of the desired requirements, which turned out to be exactly the same properties discovered independently by Cofer and Miller during their NuSMV analysis.

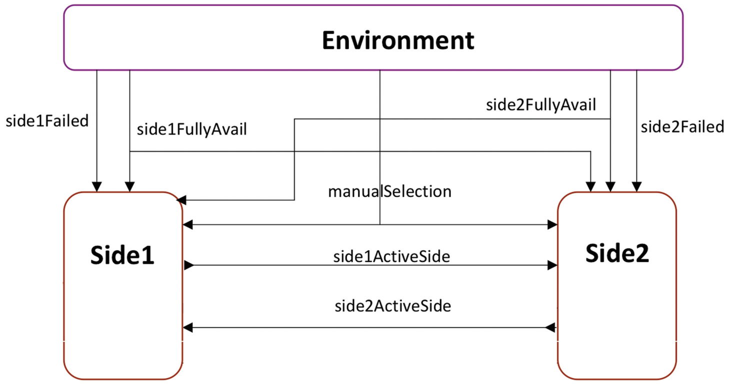


Fig. 1. The architecture of the active standby system.

Multicast Protocols. AER/NCA is a suite of protocols aimed at achieving network-friendly and reliable multicast in *active networks*. The informal specification of AER/NCA that was the starting point of the Real-Time Maude modeling and analysis effort described in [50] consisted of more than 50 pages of prose and informal "use-case" descriptions. Some of the challenges—apart from the sheer size of the protocol suite—included the need to analyze the protocols both in isolation and in combination, a detailed communication model that took packet size, link capacity, etc., into account, and sophisticated functions to update various parameters based on measures such networks congestion.

Class inheritance techniques allowed us analyze both single protocols and different combinations of protocols without modifying the protocols significantly. Another key feature was that Real-Time Maude allowed us to easily define the desired low-level model of communication very easily. In particular, we could analyze the protocols under many different link scenarios by just modifying a few parameters of the link objects. Thanks to this flexibility, our Real-Time Maude analysis—mostly simulation—found *all* the errors that the protocol developers had discovered independently using network simulators and testbeds, but had not told us about. Furthermore, our Real-Time Maude analysis revealed a significant error in the protocols that essentially invalidated the protocol, and that the protocol developers were *not* aware of.

Elisabeth Lien and I used many of the same techniques to model and analyze parts of an earlier version of the NORM multicast protocol developed by the

IETF [36]. Our model checking efforts uncovered a few errors; however, those had been corrected in later versions of NORM.

Some Other Applications. Minyoung Kim *et al.* use Real-Time Maude rewriting in [33] to estimate the performance of several power management schemes for an MPEG video streaming client. Shin Nakajima explains in [41] how Real-Time Maude can be used to model check *power consumption automata* to analyze power consumption in smartphones. Finally, in [64] Martin Wirsing *et al.* model pervasive user-centric applications—in particular, an interactive advertising board that monitors whether a person is standing in front of the board and acts accordingly—and verify them using timed temporal logic model checking.

5.2 Formalizing Formal Patterns

José Meseguer describes *formal patterns* as the formal counterparts of the well known *design patterns* in software engineering [40]. Formal patterns provide solutions to frequently occurring problems in system design, but are in additional formally specified and verified. The point is that the effort spent on verifying a formal pattern is amortized over all the instances of the pattern; they all satisfy the correctness properties. A formal pattern can be seen as a theory transformation $\mathcal{P}$ transforming a system T, with additional parameters $\varGamma$, into a system $\mathcal{P}(T,\varGamma)$ that satisfies the correctness properties of the pattern. Real-Time Maude has been used to formalize a number of formal patterns, including the following.

The Command Shaper Pattern for Medical Devices. The *command shaper* pattern developed by Mu Sun, Meseguer, and Lui Sha aims at ensuring the safe operation of medical devices connected to patients [61]. The pattern transforms a controller T that sends commands to a medical device to a new controller $\mathcal{CS}(T,\varGamma)$ to ensure that:

– the patient is not in a *stress situation* for too long; and
– the time between stress periods is sufficiently long.

The developers mention three instances of their pattern:

1. Modern pacemakers are flexible and allow a faster heart rate for limited durations, e.g., when a person is exercising. The command shaper pattern transforms a system T controlling the heart rate, together with parameters $\varGamma$ denoting the durations of the stress periods, rest times, etc., into the device controller $\mathcal{CS}(T,\varGamma)$ that ensures that the pacemaker does not provide heart rates above normal for too long, and that the rest time between strenuous activities is sufficiently long.
2. A patient in pain can control an infusion pump to administer, e.g., morphine. To avoid the patient overdosing by operating the infusion pump incorrectly, her commands can go through the command shaper to ensure that morphine

is only pumped for certain durations, and that the time between infusions is sufficiently long.
3. A mechanical *ventilator* helps a patient breathing, e.g., during surgery. However, it sometimes needs to be turned off, for example to avoid blurry pictures when taking X-rays. The command shaper pattern can ensure that the ventilator is not turned off for too long, and that the time between each pause of the ventilator is sufficiently long.

The PALS and Multirate PALS Synchronizers for Cyber-Physical Systems. Many cyber-physical systems, such as avionics, automotive, and robotics systems, are *virtually synchronous*: they proceed logically in rounds, and in each round they read input, update their local states, and produce outputs. However, such distributed systems are hard to design because of asynchrony, clock skews, and network delays. Furthermore, the model checking verification of such systems quickly becomes unfeasible due to the state space explosion caused by asynchrony. The idea of the PALS pattern [3]—developed by Steve Miller and Darren Cofer at Rockwell-Collins, Lui Sha, José Meseguer, Mu Sun, and Abdullah Al-Nayeem at UIUC, and myself—is to reduce the design and verification of a virtually synchronous CPS to designing and verifying its much simpler underlying synchronous design. Formally, PALS is a pattern transforming a synchronous design T and performance bounds Γ on the network delays, clock skews, execution times into the corresponding distributed real-time system PALS(T, Γ), that satisfy the same temporal logic properties as T [39].

The benefits of PALS can be illustrated by the active standby avionics system mentioned in Sect. 5.1. The synchronous system has 185 reachable states, whereas the number of reachable states in the simplest possible distributed version, with perfect clocks and no network delays, is 3,047,832. If the network delay can be either 0 and 1, then no model checking is feasible. Other instances of PALS include the LMST wireless sensor network protocol [32] and, presumably, the well known steam-boiler controller benchmark [1].

One limitation of PALS and other synchronizers is the assumption that all components share the same period. However, different controllers may operate at different rates. For example, the aileron controllers and the rudder controller of an airplane typically operate with different periods, yet they must synchronize to turn an airplane. Bae, Meseguer, and I therefore extended PALS to the Multirate PALS pattern to deal with virtually synchronous hierarchical multirate control systems [7]. Figure 2 shows the hierarchical multirate nature (with the period of each subsystem given in parenthesis) of the airplane turning system mentioned in Sect. 5.1. The number of states reachable in 3 s in the synchronous version of this system is 2,111, whereas the number of states reachable in 3 s is 4,415,784 in the much simplified asynchronous setting.

5.3 Semantics and Formal Analysis of Modeling Languages

Most modeling languages for real-time embedded systems (RTESs) that are used in industry currently lack a formal semantics, which not only limits unambiguous

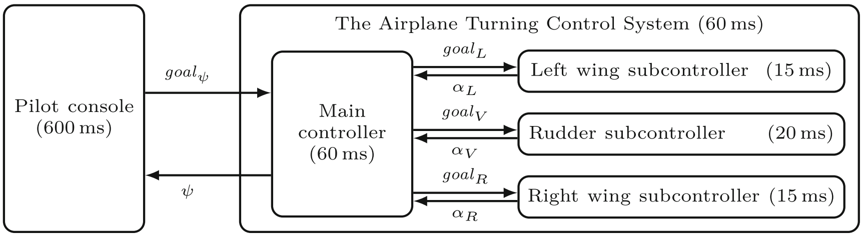

Pilot console (600 ms)
$goal_\psi$
ψ
The Airplane Turning Control System (60 ms)
Main controller (60 ms)
$goal_L$
α_L
$goal_V$
α_V
$goal_R$
α_R
Left wing subcontroller (15 ms)
Rudder subcontroller (20 ms)
Right wing subcontroller (15 ms)

Fig. 2. The architecture of our airplane turning control system.

communication between model developers, but also implies that models described in such languages cannot be subjected to formal analysis. Furthermore, some modeling languages are not executable, which limits the possibility to even simulate their models. There is therefore a clear need for:

– A formal semantic framework in which the precise semantics of a modeling language for RTESs can be defined in a natural way; and
– associated simulation and formal analysis tools which support the automated formal analysis of models in such languages.

To be useful for model-based system engineering in practice, the formal analysis framework should also:

1. Allow model developers to define analysis commands without understanding the formal language or the formal representation of their models; and
2. provide formal analysis results, such as counterexamples in temporal logic model checking, that the model developer can easily understand.

A number of advanced modeling tools provide a code generation infrastructure to support the generation of deployment code from a design model. Once the formal semantics of a modeling language has been defined, we can leverage this code generation infrastructure to automatically generate a *formal verification model* from the informal design model, enabling a *formal model-engineering* process that combines the convenience of modeling using an informal but intuitive modeling language with formal analysis.

An important point is that informal modeling languages invariably are fairly expressive; all of the languages mentioned in this section are Turing-complete. Therefore, they cannot be given a semantics using a decidable formalism. However, Real-Time Maude, with its natural model of time and its expressiveness, should make it a suitable semantic framework for modeling languages for RTESs. Furthermore, since it provides both rewriting and model checking, it should also be a suitable simulation and formal analysis back-end for such languages.

Real-Time Maude addresses the desiderata (1) and (2) above as follows:

1. *Parametric* atomic state propositions allow us to (pre-)define useful parametric state propositions in the Real-Time Maude interpreter of a language, making it easy for the user to define temporal logic formulas.

2. A key requirement to (i) understanding the results of Real-Time Maude analyses, and (ii) being able to map them back into the original modeling formalism is to have, respectively, a small *representational distance* between the original models and their formal counterparts, and a one-to-one correspondence between these models. Since hierarchical composition and encapsulation play key roles in modeling languages for industrial systems, the possibility of defining *hierarchical* objects enable us to achieve both small representational distance and the above one-to-one correspondence.

This section summarizes some uses of Real-Time Maude to define the semantics and provide a formal analysis back-end for RTESs modeling languages.

Ptolemy II Discrete-Event Models. Ptolemy II [23] is a well-established graphical modeling language and simulation tool for real-time and embedded systems used in industry. In Ptolemy II, real-time systems are modeled as *discrete-event* (DE) models. Like many graphical modeling languages, Ptolemy II DE models lack formal verification capabilities.

A Ptolemy II model consists of a set of *actors* with *input ports* and *output ports* used to pass *events* between actors. There are different kinds of actors, including clocks that generate events, timers, *delay* actors that output their input event after a fixed delay, and finite state machine (FSM) actors; furthermore, a Ptolemy II model can be encapsulated as a *composite* actor. Each event has a *timestamp* denoting the model time at which the event occurs.

An *event queue* is used for the execution. In each iteration of the system, the events in the queue with the smallest timestamp are executed. All components with input execute *synchronously*. Since connections are instantaneous and the components execute in lock-step, we must compute the *fixed point* of the input for each component in the round before its execution; this input comes from the output of another actor's execution in the same synchronous round. Figure 3 shows a hierarchical Ptolemy II model of a fault-tolerant traffic light system, consisting of one car light and one pedestrian light, at a pedestrian crossing.

Defining the formal semantics of Ptolemy II DE models is challenging; in addition to FMSs with unbounded variables, it involves unbounded queues, and, in particular, computing fixed points in hierarchical systems. Kyungmin Bae and others define the Real-Time Maude semantics of Ptolemy II DE models in [11].

We have used Ptolemy II's code generation infrastructure to integrate both the synthesis of a Real-Time Maude model from a Ptolemy II model as well as Real-Time Maude model checking of the synthesized model into Ptolemy II itself. When the blue `RTMaudeCodeGenerator` button in a Ptolemy II DE model is double-clicked, Ptolemy II opens a dialog window (shown in Fig. 4) which allows the user to give model checking commands to formally analyze her model.

We have also predefined in our model checker useful atomic propositions. For example, the proposition

$$actorId \;\texttt{|}\; var_1 \texttt{=} value_1 \texttt{,} \ldots \texttt{,}\; var_n \texttt{=} value_n$$

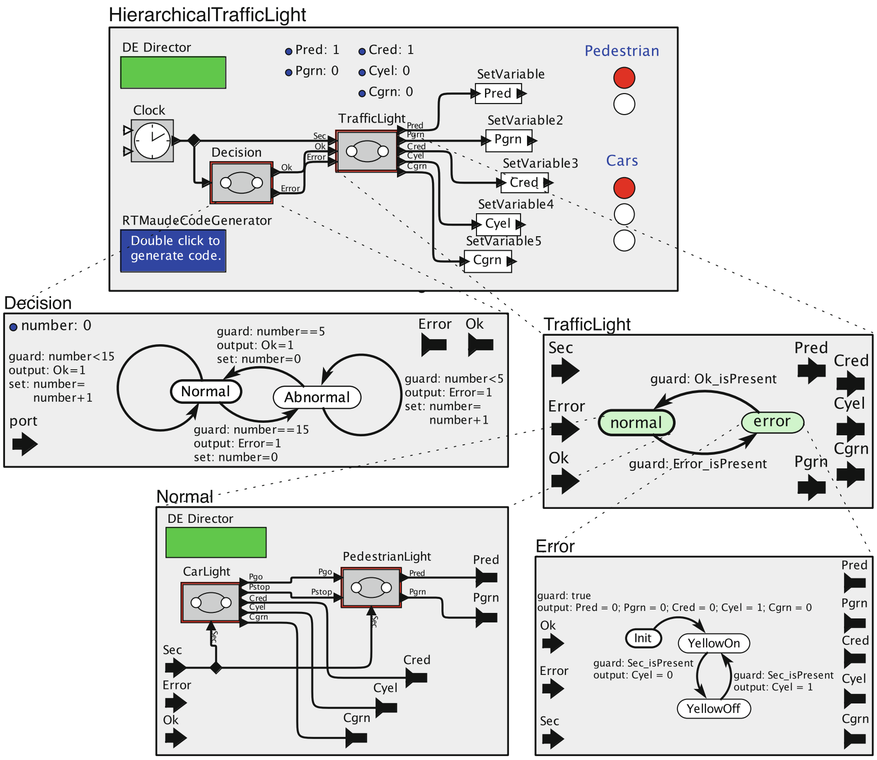


Fig. 3. A hierarchical fault-tolerant traffic light system in Ptolemy II.

holds if the value of the parameter var_i of the actor *actorId* equals $value_i$. Similarly, *actorId* `| port` *p* `is` *value* and *actorId* `| port` *p* `is` *status* hold if, respectively, the port *p* of actor *actorId* has the value *value* and status *status*.

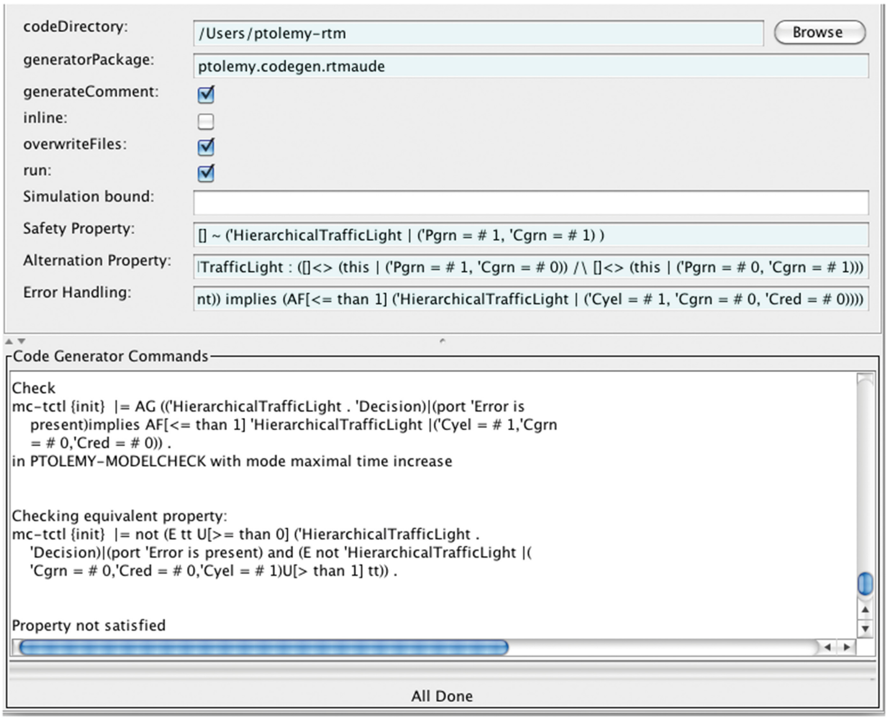
The Real-Time Maude formalization of a Ptolemy II DE model is time-robust, and the above atomic propositions are tick-invariant; Real-Time Maude model checking using the efficient maximal time sampling strategy therefore yields a sound and complete model checking procedure for Ptolemy II DE models.

In the traffic light system, the following *timed* CTL property states that the car light will turn yellow, *and only yellow*, within 1 time unit of a failure:

```
AG (('HierarchicalTrafficLight . 'Decision | port 'Error is present) implies
    AF[<= than 1] ('HierarchicalTrafficLight | 'Cyel = 1, 'Cgrn = 0, 'Cred = 0))
```

Model checking shows that this property is not satisfied (see Fig. 4), which made us aware of a previously unknown error: the car light may show red or green in addition to blinking yellow during a failure.

AADL. The *Architecture Analysis & Design Language* (AADL) [28,58] is an industrial modeling standard used in avionics, aerospace, automotive, medical



codeDirectory: /Users/ptolemy-rtm

generatorPackage: ptolemy.codegen.rtmaude

generateComment: ✓

inline:

overwriteFiles: ✓

run: ✓

Simulation bound:

Safety Property: [] ~ ('HierarchicalTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))

Alternation Property: TrafficLight : ([]<> (this | ('Pgrn = # 1, 'Cgrn = # 0)) /\ []<> (this | ('Pgrn = # 0, 'Cgrn = # 1)))

Error Handling: nt)) implies (AF[<= than 1] ('HierarchicalTrafficLight | ('Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0)))

Code Generator Commands

```
Check
mc-tctl {init} |= AG (('HierarchicalTrafficLight . 'Decision)|(port 'Error is
   present)implies AF[<= than 1] 'HierarchicalTrafficLight |('Cyel = # 1,'Cgrn
   = # 0,'Cred = # 0)) .
in PTOLEMY-MODELCHECK with mode maximal time increase


Checking equivalent property:
mc-tctl {init} |= not (E tt U[>= than 0] ('HierarchicalTrafficLight .
   'Decision)|(port 'Error is present) and (E not 'HierarchicalTrafficLight |(
   'Cgrn = # 0,'Cred = # 0,'Cyel = # 1)U[> than 1] tt)) .


Property not satisfied
```

All Done

Fig. 4. Dialog window for the Real Time Maude code generation and analysis.

devices, and robotics communities—including Honeywell, Rockwell-Collins, Lockheed Martin, General Dynamics, Airbus, the European Space Agency, Dassault, EADS, Ford, and Toyota—to describe an embedded real-time system as an assembly of software components mapped onto an execution platform. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

The AADL standard is defined using English prose, which makes it ambiguous and also fails to make explicit important assumptions. In joint work with José Meseguer, I have defined the Real-Time Maude semantics of a subset of the *software components* of AADL [45]. This subset defines the architectural and behavioral specification of a system as a set of hierarchical components with ports and connections, with the thread behaviors given by Turing-complete transitions systems defined in AADL's *behavior annex*. Together with Artur Boronat, we have also developed an OSATE plug-in that generates a Real-Time Maude specification from an AADL model.

Papers by Belala *et al.* describe other efforts to define the semantics of a subset of behavioral AADL models in Real-Time Maude [15,16].

Synchronous AADL and Multirate Synchronous AADL. As mentioned in Sect. 5.2, the PALS pattern can greatly simplify the design and verification of virtually synchronous cyber-physical systems. To make the PALS methodology easily accessible to the modeler, Bae, Meseguer, Al-Nayeem and I have in [8]:

- Defined the *Synchronous AADL* language to allow a modeler to define her synchronous PALS model in AADL.
- Defined the Real-Time Maude semantics of Synchronous AADL.
- Implemented the *SynchAADL2Maude* [9] OSATE plug-in to support the development and verification of Synchronous AADL models in OSATE.

For LTL model checking purposes, our tool has pre-defined useful parametric atomic propositions, and we have used *SynchAADL2Maude* to model and verify (the weakened) correctness requirements of the *Active Standby* system mentioned in Sect. 5.1. Furthermore, Al-Nayeem has also automated the generation of an AADL model of the real-time system PALS(T, Γ) from a synchronous PALS model T defined using Synchronous AADL.

In the same way, Bae, Meseguer, and I have defined the *Multirate Synchronous AADL* language to support the modeling of Multirate PALS synchronous designs in AADL [12]. We have defined the Real-Time Maude semantics of our language, and have integrated the *MR-SynchAADL* tool to support the modeling and Real-Time Maude analysis of Multirate Synchronous AADL models inside OSATE.

To support the easy definition of temporal logic properties, we have again defined a number of useful parametric atomic propositions, including

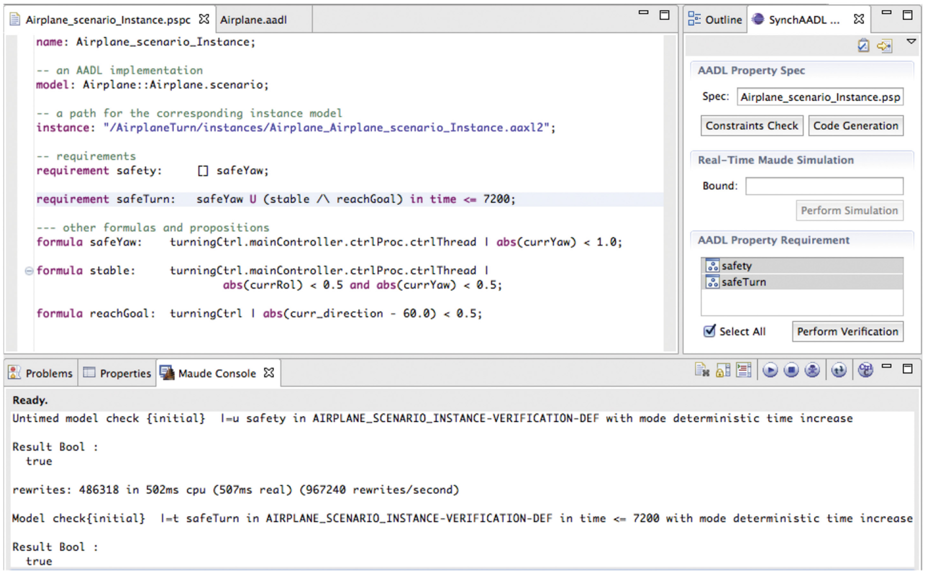
full component name `|` *boolean expression*

which holds if *boolean expression* evaluates to *true* in the given component.

Figure 5 shows the MR-SynchAADL window for the airplane turning algorithm system in Sect. 5.1. In the editor part, two system requirements, `safety` (the yaw angle is always less than 1.0°) and `safeTurn` (the yaw angle is less than 1.0° until we eventually reach the goal direction and the plane is stable) are specified using the requirement specification language and listed in the "AADL Property Requirement" table. The `Constraints Check`, the `Code Generation`, and the `Perform Verification` buttons are used to perform, respectively, the syntactic validation of the model, the Real-Time Maude code generation, and the LTL model checking. The `Perform Verification` button has been clicked and the results are shown in the "Maude Console."

Both Synchronous AADL and Multirate Synchronous AADL models are time-robust and the predefined atomic propositions are tick-invariant, so that Real-Time Maude analyses are sound and complete.

Timed Rebeca. Timed Rebeca [2] is an actor-based modeling language with a Java-like syntax and a simple and intuitive message-driven and object-based computational model. In addition to a statement language with (nondeterministic) assignments, conditionals, and loops, Timed Rebeca also supports the dynamic creation of new actors. Zeynab Sabahi-Kaviani and others define the Real-Time Maude semantics of Timed Rebeca and integrate formal analysis in



```
name: Airplane_scenario_Instance;

-- an AADL implementation
model: Airplane::Airplane.scenario;

-- a path for the corresponding instance model
instance: "/AirplaneTurn/instances/Airplane_Airplane_scenario_Instance.aaxl2";

-- requirements
requirement safety:     [] safeYaw;

requirement safeTurn:   safeYaw U (stable /\ reachGoal) in time <= 7200;

--- other formulas and propositions
formula safeYaw:     turningCtrl.mainController.ctrlProc.ctrlThread | abs(currYaw) < 1.0;

formula stable:      turningCtrl.mainController.ctrlProc.ctrlThread |
                              abs(currRol) < 0.5 and abs(currYaw) < 0.5;

formula reachGoal:  turningCtrl | abs(curr_direction - 60.0) < 0.5;
```

```
Ready.
Untimed model check {initial}  |=u safety in AIRPLANE_SCENARIO_INSTANCE-VERIFICATION-DEF with mode deterministic time increase

Result Bool :
  true

rewrites: 486318 in 502ms cpu (507ms real) (967240 rewrites/second)

Model check{initial}  |=t safeTurn in AIRPLANE_SCENARIO_INSTANCE-VERIFICATION-DEF in time <= 7200 with mode deterministic time increase

Result Bool :
  true
```

Fig. 5. MR-SynchAADL window in OSATE.

the Rebeca toolset in [56,57]. Real-Time Maude's possibility of creating new objects dynamically is key to define the semantics of Timed Rebeca. It is again worth mentioning that the resulting Real-Time Maude specifications are time-robust.

Timed Model Transformations Frameworks. The MOMENT2 [20] formal model transformation framework is based on a formalization of MOF meta-models in rewriting logic. The static semantics of a system is given as a class diagram (or meta-model) describing the set of valid system states (or models) that are represented as object diagrams, and the dynamics of a system is defined as an *in-place model transformation*. In joint work with Artur Boronat, I have extended MOMENT2 to support the definition of timed behaviors by providing a set of basic constructs such as clocks and timers [19], and have defined the semantics of such timed model transformations in Real-Time Maude [19].

The e-Motions model transformation framework [55] for domain-specific visual languages is also based on EMF. Although the specification of behaviors is based on in-place model transformations, e-Motions does not use ("low-level") constructs such as clocks and timers to support timed model transformations. Instead, timed behaviors are defined by different kinds of *timed model transformation rules* of the form

$$[NAC]^* \times LHS \overset{[t_{min},t_{max}]}{\longrightarrow} RHS$$

where *LHS* (its left-hand side), *RHS* (its right-hand side), and the optional *NAC*s (negative application conditions) are model patterns that represent state fragments, and the interval $[t_{min}, t_{max}]$ defines the possible *durations* of the rule. A rule "instance" is *triggered* as soon as it is enabled, and is *executed* some time $t \in [t_{min}, t_{max}]$ after being triggered. An atomic rule can also be declared to be *soft*, which means that it is not triggered eagerly, and/or may be declared to be *periodic*, in which case it is triggered periodically (for each instance) as long it is enabled. *Ongoing rules* do not have a fixed duration but are applied as long as the precondition holds. These are powerful constructs that typically imply that many different rules are being applied simultaneously to an object. This also means that a Real-Time Maude semantics is clearly needed, and is indeed provided by Rivera, Dúran, and Vallecillo in [54,55].

A Modeling Language for Handset Software. In [4], Musab AlTurki and researchers at DOCOMO USA Labs give a Real-Time Maude semantics to a simple but powerful specification language, called $\mathcal{L}$, that is claimed to be well suited for describing a spectrum of behaviors of various software systems. The language provides flexible SDL-inspired timing constructs that yield a more expressive language for timed behaviors than Erlang.

The language has an expression language, imperative features for describing sequential computations, and asynchronously communicating processes that can be dynamically created or destroyed. It is worth remarking that already the dynamic process creation places $\mathcal{L}$ outside the class of systems that can be represented as timed automata; so does its expression language and imperative features which make $\mathcal{L}$ Turing-complete.

5.4 Analysis of Distributed Maude Programs

(Core) Maude provides support for communication with external objects by means of TCP sockets. In this way multiple Maude processes running on different machines can be connected, giving rise to distributed Maude programs. Since such a program cannot be model checked directly, it must be related to a more abstract non-distributed (Real-Time) Maude model for formal analysis purposes. For real-time systems, this involves relating logical time in a Real-Time Maude specification to real physical time in a distributed implementation. This section summarizes three papers that verify Real-Time Maude abstractions of distributed Maude implementations.

Implementing and Analyzing the EIGRP Routing Protocol. In [53], Adrían Riesco and Alberto Verdejo provide a distributed formalization/"implementation" of the *Enhanced Interior Gateway Routing Protocol* (EIGRP), which is a CISCO proprietary distance-vector routing protocol, in Maude.

EIGRP is a *real-time* protocol: a router periodically broadcasts "hello" messages to its one-hop neighbors; if a router does not hear a "hello" message for a certain amount of time, it assumes that the link to that router is down. Since Maude does not provide support for distributed real-time application directly, Riesco and Verdejo use another external object, a Java object with access to real time, for timing purposes as follows: a Maude object can send a message `wait(`n`)` to this external Java "clock" object, asking that object to reply back with a `tick` message after n milliseconds of real time have elapsed.

For analysis purposes, Riesco and Verdejo transform their distributed model back to a single Real-Time Maude system. A key technique here is to formalize the properties of Maude's socket features, so that the specification being model checked resembles the original distributed system as closely as possible.

Two of the advantages of EIGRP compared to other routing protocols is that EIGRP has loop-free routing and fast convergence. Riesco and Verdejo use Real-Time Maude's *find latest* command to analyze the convergence time, its search command to check whether all routes are always loop-free, and its LTL model checker to check whether the best routes are eventually found. No errors in EIGRP were discovered during this analysis.

Generating and Analyzing Distributed Implementations of Orc Programs. Orc is an elegant and powerful programming language for orchestrating web services. In [5], Musab AlTurki and José Meseguer show how (i) one can go from an Orc specification to a distributed Maude implementation of the Orc specification, using their Maude semantics of Orc and Maude sockets; and (ii) how such distributed implementations can be model checked using Real-Time Maude. Orc is a timed language, and, as in [53], AlTurki and Meseguer integrate physical time in their distributed implementations by each distributed node having a local clock object. This clock is a Java object that uses the built-in Java classes `Timer` and `Socket` to send a "tick" message every t time units to the co-located Maude process.

To formally analyze a distributed implementation, AlTurki and Meseguer also formally specify in Real-Time Maude both the internet sockets supporting the distributed implementation and the local clock objects. They illustrate their methodology by taking an Orc specification of a simplified online auction management system, generating the distributed Maude implementation of the system, executing the distributed Maude program using the `erew` command in Maude, transforming this distributed Maude implementation into a Real-Time Maude specification using formal specification of the behaviors of the sockets and external clocks, and, finally, model checking the resulting specification using Real-Time Maude.

Real-Time Emulation of Verified Medical Device Controllers. Mu Sun and José Meseguer go the other way in [60]: from verified Real-Time Maude models to concrete systems operating with their environments in real time. More precisely, they create executable *emulations* of medical device controllers from Real-Time Maude models that are instances of their command shaper pattern. These emulations can be connected to real devices to validate the safety of the device in a real environment. Essentially, an execution "wrapper" around the Real-Time Maude model deals with handling time and messages to and from the external world. Model time is again related to physical time by interacting with a

Java thread. Sun and Meseguer advance time maximally by sending a "time-out request" to the time thread; however, they can also deal with "interrupts" from external devices which may happen at any time. They analyze the time that the "instantaneous" transitions and message passing take (never more than 0.2 s), and make sure that any skew is not multiplied over time.

Sun and Meseguer present two case studies, where the safe device controllers are instances of the command shaper pattern. The first case study is a pacemaker controller. Sun and Meseguer connect their device controller emulator to

– a "user" that sends a "dangerous" sequence of pacemaker commands to the device controller; and
– a Java widget that simulates a pacemaker by receiving pacemaker commands (from the device controller) and drawing the corresponding ECG trace.

The ECG trace shows that the device controller ensured a safe heart rate. In their second case study they actually connect their wrapped Real-Time Maude model of a safe syringe pump controller to a real Multi-Phaser NE-500 syringe pump, and validate their controller by weighing the amount of liquid infused from the syringe pump.

6 Extensions of Real-Time Maude

This section presents two extensions of Real-Time Maude, namely, to hybrid systems and priced timed systems.

6.1 HI-Maude: Object-Oriented Modeling and Formal Analysis of Continuously Interacting Hybrid Systems

Section 5 shows that Real-Time Maude's expressiveness and modeling flexibility have made it possible to successfully apply the tool to a number of large and complex applications, all of which have been modeled in an object-oriented style. An important question is whether Real-Time Maude can also successfully model and analyze complex *hybrid* systems in an object-oriented way.

It is a nightmare to define the continuous dynamics of many hybrid systems, since different components may influence each other's continuous behaviors; we call such systems *(continuously) interacting hybrid systems*. Consider the problem of keeping track of the temperature of a hot cup of coffee in a colder room. The temperature of the coffee will continuously decrease *and* the temperature of the room will continuously *increase* due to heat transfer from the coffee. Although the continuous behaviors of the single components and the heat flow between them are well known, it is very hard to define the continuous behavior of the entire system "in one shot," which is what current formal models of hybrid systems require. Existing formalisms therefore also do not support the object-oriented specification of continuously interacting hybrid systems, since the continuous behavior must be completely redefined for each new configuration of objects (for example, if we have *three* cups of coffee), and therefore cannot be defined at the class level.

Muhammad Fadlisyah, Erika Ábrahám, and I have addressed the formal modeling and analysis of interacting hybrid systems in [24–27]. In our object-oriented modeling methodology, which is based on the *effort/flow* method, *both* the *physical entities* and the *physical interactions* between the entities are modeled explicitly as objects. For example, heat flows from the coffee and the cup to the room through heat *convection*, and heat flows between the coffee and the cup through heat *conduction*. This approach is applicable to different kinds of systems, including mechanical translation systems, mechanical rotation systems, electrical systems, fluidic systems, and thermal systems.

Our modeling methodology is supported by the HI-Maude tool [24], which extends Real-Time Maude. In HI-Maude, one can define the continuous dynamics of *single* physical component objects and *single* interaction objects. HI-Maude then computes the continuous dynamics of the entire system. This enables object-oriented modeling, since both the discrete and the continuous dynamics are defined at the class level, and since the dynamic creation/deletion of physical components is supported. For example, to add another cup of coffee, one could just add (possibly dynamically) a new coffee object, a new cup object, and three new interaction objects to the state.

To analyze hybrid systems—and HI-Maude targets complex systems whose continuous dynamics may be defined by differential equations that are not analytically solvable—we have adapted different numerical methods (the Euler method and the Runge-Kutta methods of 2nd and 4th order) to our modeling methodology to give approximate solutions to coupled differential equations. These approximations are then used in HI-Maude simulation, reachability analysis, and linear temporal logic model checking. For example, HI-Maude's hybrid *rewrite* command is used to simulate one behavior of the system from a given initial state *initState* up to duration *timeLimit*:

```
(hrew initState in time <= timeLimit using numMethod stepsize stepSize .)
```

where *numMethod* $\in$ {`euler`, `rk2`, `rk4`} is the numerical approximation method used and *stepSize* is the time increment used in the approximations.

Since the numerical methods only approximate the real continuous behaviors, *HI-Maude analyses are in general not sound and complete.* The main value of HI-Maude is therefore to formally define and simulate complex hybrid systems.

Case Study: Modeling the Human Thermoregulatory System. The *human thermoregulatory system* (HTS) attempts to ensure human survival and comfort in different environments. The HTS has been studied extensively—sometimes by experimentation on live subjects—to analyze how long a person can survive in cold water, how long he can exercise in hot weather without succumbing to heat stroke, and how to provide the most comfortable environment for pilots, astronauts, and airplane passengers, and so on.

In the HTS, the hypothalamus part of the brain enables the following mechanisms to support *heat loss* from the body when needed: increasing the diameter of blood vessels to let more blood flow underneath the skin (*vasodilation*),

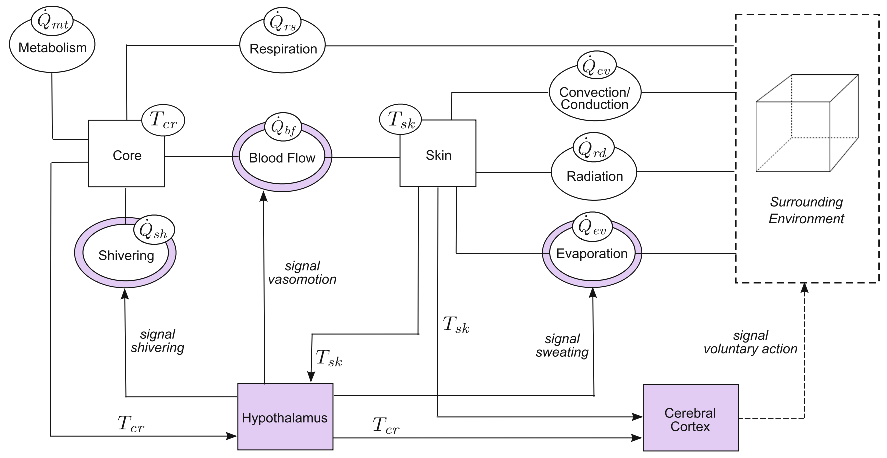


Fig. 6. Effort/flow model of the human thermoregulatory system.

which promotes heat loss by radiation, convection, and conduction; and increasing sweat production, which promotes heat loss by evaporation. When the body temperature is decreasing, the hypothalamus may decrease the diameter of blood vessels to let less blood flow underneath the skin (*vasoconstriction*) to reduce heat loss, and may stimulate the skeletal muscles to cause shivering to increase heat production. Behavioral thermoregulation (e.g., putting on or taking off a jacket) is related to a part of the brain called the cerebral cortex.

In [27] we define a HI-Maude model of the human thermoregulatory system according to accepted physiological facts and models, where the *body core* and the *skin* are two main components. Heat flows between them through blood flows where the diameter of the blood vessels changes continuously. The main forms of heat exchange between skin and environment are by conduction/convection, radiation, and evaporation of sweat; between core and environment heat flows mainly through respiration. Figure 6 shows the physical entities (boxes) and interactions (ovals) in our model, where heat production by metabolism and by shivering are represented as one-sided interactions.

We can connect our fairly sophisticated model to different environments. We chose to analyze possible causes of the accident at the 2010 Sauna World Championships, which ended in a tragedy when the two finalists collapsed with severe burn injuries after about six minutes; one of them died the next day. The cause of this tragedy is still under investigation. Our HI-Maude analyses show that even the average person should endure 12 min in the sauna before the onset of major injuries. We also used HI-Maude to analyze what scenarios could cause major injuries to a five-time world champion in around 6 min [27].

Priced-Timed Maude. Reasoning about the accumulated *cost* (say, *price* or *energy usage*) during behaviors is crucial in embedded systems and sensor networks where minimizing overall energy consumption is critical. The *Priced-Timed Maude* tool [17] extends Real-Time Maude to support the formal modeling of non-hierarchical object-oriented priced and timed systems by adding *priced* rules of the form c `=>` c' `with cost` u `if` *cond*, where c and c' are terms of sort `Configuration`, and priced tick rules of the form `{`t`}` `=>` `{`t'`}` `in time` τ `with cost` u `if` *cond*. Apart from extending Real-Time Maude's analysis commands in the expected way, Priced-Timed Maude also adds commands for finding *optimal* solutions, such as the cheapest behavior leading to a desired state.

Although Priced-Timed Maude has been applied to benchmarks such as energy task graph scheduling, the airplane landing problem, and to the slightly larger problem of efficiently routing passengers within a subway network while minimizing power consumption of the trains, the tool has not been applied to state-of-the-art applications.

7 Concluding Remarks

Real-Time Maude is an expressive modeling language and a formal analysis tool that is particularly useful for defining distributed real-time systems in an object-oriented way. Indeed, virtually all the Real-Time Maude applications summarized in Sect. 5 have been object-oriented Real-Time Maude specifications. I have shown that Real-Time Maude features such as user-definable data types, hierarchical objects, dynamic object creation, unbounded data structures, and the possibility to easily define the appropriate communication model have all been needed to apply the tool on a number of advanced state-of-the-art applications and to define the formal semantics of several modeling languages.

Despite the size and complexity of the applications, Real-Time Maude analysis – both simulation and model checking – could still be used to discover significant previously unknown flaws in many of the applications, as well as to provide formal analysis capabilities for the modeling languages. Furthermore, many of those applications and modeling languages are formalized as time-robust Real-Time Maude specifications, which means that their Real-Time Maude analyses are guaranteed to be sound and complete.

Many large real-time systems are *probabilistic* systems, either because their algorithms are probabilistic in nature or because there is a need to analyze their performance in an environment which can be seen as probabilistic. Real-Time Maude should therefore be extended to model probabilistic behaviors. This would also enable useful and scalable analysis by means of statistical model checking. Finally, Real-Time Maude should also incorporate symbolic analysis techniques, including the use of SMT solvers, to increase the efficiency and analytic power of Real-Time Maude model checking.

Acknowledgments. I thank all my coauthors for very nice collaborations which have motivated, improved, and validated Real-Time Maude. In particular, I would like to thank José Meseguer, who has been part of the entire Real-Time Maude journey, Martin Wirsing, who initiated this journey, and Kyungmin Bae. I am also grateful to Santiago Escobar for inviting me to give an invited talk at WRLA 2014 and for patiently waiting for this paper to be finished.

References

1. Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. LNCS, vol. 1165. Springer, Heidelberg (1996)
2. 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: Proceedings of FOCLASA'11, vol. 58. EPTCS (2011)
3. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: RTSS'09. IEEE (2009)
4. AlTurki, M., Dhurjati, D., Yu, D., Chander, A., Inamura, H.: Formal specification and analysis of timing properties in software systems. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 262–277. Springer, Heidelberg (2009)
5. AlTurki, M., Meseguer, J.: Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In: Proceedings of RTRTS'10, vol. 36. EPTCS (2010)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
7. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Sci. Comput. Programm. **91(A)**, 3–44 (2014)
8. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 651–667. Springer, Heidelberg (2011)
9. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 59–62. Springer, Heidelberg (2012)
10. Bae, K., Krisiloff, J., Meseguer, J., Ölveczky, P.C.: Designing and verifying distributed cyber-physical systems using Multirate PALS: an airplane turning control system case study. Sci. Comput. Programm. (2014, to appear). doi: 10.1016/j.scico.2014.09.011
11. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. Sci. Comput. Program. **77**(12), 1235–1271 (2012)
12. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 94–109. Springer, Heidelberg (2014)
13. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
14. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
15. Belala, F., Benammar, M., Barkaoui, K., Hicheur, A.: Formal modeling and analysis of AADL threads in Real-Time Maude. J. Softw. Eng. Appl. **5**, 187–192 (2012)
16. Benammar, M., Belala, F.: How to make AADL specification more precise. Int. J. Comput. Appl. **8**(10), 16–23 (2010)
17. Bendiksen, L., Ölveczky, P.C.: The Priced-Timed Maude tool. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 443–448. Springer, Heidelberg (2009)
18. Bergstra, J.A., Tucker, J.V.: Algebraic specification of computable and semicomputable data types. Theoret. Comput. Sci. **50**, 137–181 (1987)

19. Boronat, A., Ölveczky, P.C.: Formal real-time model transformations in MOMENT2. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 29–43. Springer, Heidelberg (2010)
20. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
21. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
22. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
23. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity–the Ptolemy approach. Proc. IEEE **91**(2), 127–144 (2003)
24. Fadlisyah, M., Ölveczky, P.C.: The HI-Maude tool. In: Heckel, R., Milius, S. (eds.) CALCO 2013. LNCS, vol. 8089, pp. 322–327. Springer, Heidelberg (2013)
25. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Formal modeling and analysis of hybrid systems in rewriting logic using higher-order numerical methods and discrete-event detection. In: Proceedings of CSSE'11. IEEE (2011)
26. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Object-oriented formal modeling and analysis of interacting hybrid systems in HI-Maude. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 415–430. Springer, Heidelberg (2011)
27. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Formal modeling and analysis of interacting hybrid systems in HI-Maude: What happened at the 2010 Sauna World Championships? Sci. Comput. Programm. (2014). http://dx.doi.org/10.1016/j.scico.2014.06.010
28. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL. Addison-Wesley, Upper Saddle River (2012)
29. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014)
30. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Heidelberg (2014)
31. Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008)
32. Katelman, M., Meseguer, J.: Using the PALS architecture to verify a distributed topology control protocol for wireless multi-hop networks in the presence of node failures. In: Proceedings of RTRTS'10, vol. 36. EPTCS (2010)
33. Kim, M., Dutt, N., Venkatasubramanian, N.: Policy construction and validation for energy minimization in cross layered systems: a formal method approach. In: Proceedings of IEEE RTAS'06 Work-in-Progress Session, pp. 4–7 (2006)
34. Lepri, D., Ábrahám, E., Ölveczky, P.C.: A timed CTL model checker for Real-Time Maude. In: Heckel, R., Milius, S. (eds.) CALCO 2013. LNCS, vol. 8089, pp. 334–339. Springer, Heidelberg (2013)

35. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Sci. Comput. Program. (2014). http://dx.doi.org/10.1016/j.scico.2014.06.006
36. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: Proceedings of SEFM'09. IEEE (2009)
37. Liu, S., Ölveczky, P., Meseguer, J.: A framework for mobile ad hoc networks in Real-Time Maude. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 162–177. Springer, Heidelberg (2014)
38. Liu, S., Ölveczky, P., Meseguer, J.: Formal analysis of leader election in MANETs using Real-Time Maude. In: Festschrift honoring Martin Wirsing. LNCS. Springer, Heidelberg (2015)
39. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoret. Comput. Sci. **451**, 1–37 (2012)
40. Meseguer, J.: Taming distributed system complexity through formal patterns. Sci. Comput. Program. **83**, 3–34 (2014)
41. Nakajima, S.: Model-based power consumption analysis of smartphone applications. In: Proceedings of ACESMB@MoDELS, CEUR Workshop Proceedings, vol. 1084. CEUR-WS.org (2013)
42. Ölveczky, P.C.: Real-Time Maude 2.3 Manual (2007). http://ifi.uio.no/RealTimeMaude/
43. Ölveczky, P.C.: Formal model engineering for embedded systems using Real-Time Maude. In: Proceedings of AMMSE'11, vol. 56. EPTCS (2011)
44. Ölveczky, P.C.: Semantics, simulation, and formal analysis of modeling languages for embedded systems in Real-Time Maude. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 368–402. Springer, Heidelberg (2011)
45. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010, Part II. LNCS, vol. 6117, pp. 47–62. Springer, Heidelberg (2010)
46. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
47. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoret. Comput. Sci. **285**, 359–405 (2002)
48. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. ENTCS **176**(4), 5–27 (2007)
49. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order Symbolic Comput. **20**(1–2), 161–196 (2007)
50. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods Syst. Des. **29**(3), 253–293 (2006)
51. Ölveczky, P.C., Prabhakar, P., Liu, X.: Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In: Proceedings of IPDPS'08. IEEE (2008)
52. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoret. Comput. Sci. **410**(2–3), 254–280 (2009)
53. Riesco, A., Verdejo, A.: Implementing and analyzing in Maude the Enhanced Interior Gateway Routing Protocol. ENTCS **238**(3), 249–266 (2009)

54. Rivera, J.E.: On the semantics of real-time domain specific modeling languages. Ph.D. thesis, Universidad de Málaga (2010)
55. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 174–190. Springer, Heidelberg (2010)
56. Sabahi-Kaviani, Z., Khosravi, R., Ölveczky, P.C., Khamespanah, E., Sirjani, M.: Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude (2014, submitted for publication)
57. Sabahi-Kaviani, Z., Khosravi, R., Sirjani, M., Ölveczky, P.C., Khamespanah, E.: Formal semantics and analysis of Timed Rebeca in Real-Time Maude. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2013. CCIS, vol. 419, pp. 178–194. Springer, Heidelberg (2014)
58. SEA International: Architecture Analysis & Design Language (AADL). Standard AS5506, Revision B, September 2012. http://standards.sae.org/as5506b/
59. Sen, K., Viswanathan, M., Agha, G.A.: VeStA: a statistical model-checker and analyzer for probabilistic systems. In: Proceedings of QEST'05. IEEE (2005)
60. Sun, M., Meseguer, J.: Distributed real-time emulation of formally-defined patterns for safe medical device control. In: Proceedings of RTRTS'10. vol. 36. EPTCS (2010)
61. Sun, M., Meseguer, J., Sha, L.: A formal pattern architecture for safe medical systems. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 157–173. Springer, Heidelberg (2010)
62. Vasudevan, S., Kurose, J.F., Towsley, D.F.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: Proceedings of ICNP'04. IEEE (2004)
63. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Softw. Tools Technol. Trans. **6**(1), 77–97 (2004)
64. Wirsing, M., Bauer, S.S., Schroeder, A.: Modeling and analyzing adaptive user-centric systems in Real-Time Maude. In: Proceedings of RTRTS'10, vol. 36. EPTCS (2010)
65. Zhang, H., Hou, J.C.: Maintaining sensing coverage and connectivity in large sensor networks. Wirel. Ad Hoc Sens. Netw. Int. J. **1**(1–2), 89–124 (2005)
66. Zhu, L., Liu, P., Shi, J., Wang, Z., Zhu, H.: A timing verification framework for AUTOSAR OS component development based on Real-Time Maude. In: Proceedings of TASE'13. IEEE (2013)