# A Rewriting-Logic-Based Technique for Modeling Thermal Systems

Muhammad Fadlisyah

University of Oslo

Erika Ábrahám

RWTH Aachen

Daniela Lepri

University of Oslo

Peter Csaba Ölveczky

University of Oslo

This paper presents a rewriting-logic-based modeling and analysis technique for physical systems, with focus on thermal systems. The contributions of this paper can be summarized as follows: (i) providing a framework for modeling and executing physical systems, where both the physical components and their physical interactions are treated as first-class citizens; (ii) showing how heat transfer problems in thermal systems can be modeled in Real-Time Maude; (iii) giving the implementation in Real-Time Maude of a basic numerical technique for executing continuous behaviors in object-oriented hybrid systems; and (iv) illustrating these techniques with a set of incremental case studies using realistic physical parameters, with examples of simulation and model checking analyses.

## 1 Introduction

The rewriting-logic-based *Real-Time Maude* tool [12] has proved to be very useful for formally modeling and analyzing a wide range of advanced real-time systems (see, e.g., [6, 9, 11, 13, 14]) that are beyond the scope of timed-automaton-based tools. One important question to investigate is to what degree – and *how* – Real-Time Maude can be successfully applied to formally model and analyze *hybrid* systems.

As part of this investigation, this paper presents a rewriting-logic-based modeling and analysis technique for *physical systems*, which consist of a set of *components* that behave and interact according to the laws of physics. On the one hand, the physical quantities of the components *continuously* evolve in time. On the other hand, the components may also have *discrete* state parts, thus exhibiting *hybrid* behaviors.

We propose a technique to generate *executable models* of physical systems in Real-Time Maude. Our modeling technique is based on an adaptation of the *effort* and *flow* approach described in [17], explicitly modeling the transfer of power between the components. For the continuous behavior of physical systems, which is described by differential equations, the execution is based on the Euler method (see, e.g., [5]) giving approximate solutions to ordinary differential equations. This numerical method operates with a discrete sampling strategy over a continuous time domain.

This paper focuses on *thermal systems*, which are physical systems that deal with heat transfer and temperature change problems. Thermal systems appear in many important computer-controlled (or computer-controllable) applications, such as house heating systems, human body thermoregulatory systems, and nuclear power plants. We describe the use of our modeling and execution technique with a sequence of increasingly complex thermal systems. We first model simple thermal systems, such as "a cup of coffee in a room"; then we add a coffee heater to the system; and, finally, we add an automatic coffee heater as a controlling system that manages the coffee temperature.

The main contributions of this paper can be summarized as follows:

- We develop a framework for *modeling* general *physical systems* in Real-Time Maude.

- This framework is used as the basis to define a technique to model heat transfer in *thermal systems*. We provide some basic classes, equations, and rules that can be used or extended to build models of thermal systems.

- The implementation of a basic numerical technique to execute continuous behavior of object-oriented real-time systems, which allows the analysis of their hybrid behavior.

- Incremental *case studies*, using realistic physical parameters, with examples of simulation and reachability analysis.

We give an overview of Real-Time Maude in Section 2. Section 3 gives a brief introduction to thermal systems. Section 4 presents a high-level overview of modeling physical and thermal systems with the effort and flow approach, and approximating their behavior. Section 5 describes our modeling and execution framework for thermal systems using Real-Time Maude. Several case studies are given in Section 6. Section 7 discusses related work, and Section 8 mentions future work and summarizes the paper.

## 2   Real-Time Maude

In this section we briefly introduce Real-Time Maude [12], a rewriting-logic-based tool supporting the formal modeling and analysis of real-time systems. A Real-Time Maude *timed module* specifies a *real-time rewrite theory* $(\Sigma, E, IR, TR)$, where:

- $(\Sigma, E)$ is a *membership equational logic* [2] theory with $\Sigma$ a signature[1] and $E$ a set of *confluent and terminating conditional equations*. $(\Sigma, E)$ specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.

- *IR* is a set of (possibly conditional) *labeled instantaneous rewrite rules* of the form

    rl [*l*]  : *t* => *t'*
    crl [*l*]  : *t* => *t'* if *cond*

  specifying the system's *instantaneous* (i.e., zero-time) one-step transitions from an instance of *t* to the corresponding instance of *t'*, where *l* is a *label*. Conditional instantaneous rules apply only if their conditions hold. The rules are applied *modulo* the equations $E$.[2]

- *TR* is a set of (possibly conditional) *tick (rewrite) rules*, written with syntax

    rl [*l*]  : {*t*} => {*t'*} in time $\tau$
    crl [*l*]  : {*t*} => {*t'*} in time $\tau$ if *cond*

  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and $\tau$ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form {*t*} using the equations in the specifications. The form of the tick rule then ensures that time advances uniformly in all parts of the system.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol *f* is declared with the syntax op *f* : $s_1 \ldots s_n$ -> *s*, where $s_1 \ldots s_n$ are the sorts of its arguments, and *s* is its (value) *sort*. Equations are written with syntax eq *t* = *t'*, and ceq *t* = *t'* if *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords var and vars. We refer to [2] for more details on the syntax of Real-Time Maude.

In object-oriented Real-Time Maude modules, a *class* declaration

---

[1]i.e., $\Sigma$ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[2]$E$ is a union $E' \cup A$, where $A$ is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo A*. Operationally, a term is reduced to its $E'$-normal form modulo $A$ before any rewrite rule is applied.

```
class C | att₁ : s₁, ... , attₙ : sₙ .
```

declares a class $C$ with attributes $att_1$ to $att_n$ of sorts $s_1$ to $s_n$. An *object* of class $C$ in a given state is represented as a term $< O : C \mid att_1 : val_1, ..., att_n : val_n >$ 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 $att_n$. In a concurrent object-oriented system, the state is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and *messages* (that are terms of sort `Msg`). Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

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

```
rl [l] :  < O : C | a1 : x, a2 : y, a3 : z >
           < O' : C | a1 : w, a2 : 0, a3 : v >
         =>
           < O : C | a1 : x + w, a2 : y, a3 : z >
           < O' : C | a1 : w, a2 : x, a3 : v >
```

defines a family of transitions where two objects of class `C` synchronize to update their attributes when the `a2` attribute of one of the objects has value `0`. The transitions have the effect of altering the attribute `a1` of the object `O` and the attribute `a2` of the object `O'`. "Irrelevant" attributes (such as `a3` and `a2` of `O`, and the *right-hand side* occurrence of `a1` of `O'`) need not be mentioned in a rule.

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

A Real-Time Maude specification is *executable* under reasonable conditions, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system *up to a certain duration*. It is written with syntax (`trew` $t$ `in time <=` $\tau$ `.`), where $t$ is the initial state and $\tau$ is a term of sort `Time`. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* can be reached from the initial state such that a given *condition* is satisfied. The *timed search* command, having the syntax

```
(tsearch [1] t =>* pattern such that cond in time <= τ .)
```

works similar but it restricts the search to states reachable from the initial state within time $\tau$.

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a linear temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form {*statePattern*} `|=` *prop* = *b*, for *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in all states {*t*} where {*t*} `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

```
(mc t |=t formula in time <= τ .)
```

for initial state *t* and temporal logic formula *formula* .

Finally, the `find earliest` command determines the shortest time needed to reach a desired state.

For time-nondeterministic tick rules, the above model checking commands are applied according to the chosen time sampling strategy, so that only a subset of all possible behaviors is analyzed.

## 3   Thermal Systems

We define a *thermal system* as a system whose components behave according to the laws of physics and that are related to heat transfer and temperature change. *Heat* is the form of energy that can be transferred from one system to another as a result of temperature difference. The transfer of energy as heat is always from the higher-temperature medium to the lower-temperature one. The flow of heat to a component effects the component by either changing its temperature or by changing the component's phase (such as ice melting to water).

**Temperature change.**   The equation representing the heat gained or lost by an object as it undergoes a temperature change $\Delta T$ is given by $\Delta Q = m \cdot c \cdot \Delta T$, where $m$ is the mass of the object and $c$ is the specific heat of the object at constant volume. The amount of heat transferred per time unit, the *heat transfer rate*, can be described as $\dot{Q} = m \cdot c \cdot \dot{T}$, where $\dot{T}$ represents the change of temperature per time unit.
   Heat can be transferred by three different mechanisms: *conduction*, *convection*, and *radiation*.

- *Conduction* occurs when heat flows through stationary materials. The rate of heat conduction through a medium depends on the geometry of the medium, the thickness of the medium, the material of the medium, and temperature difference across the medium. The *conduction rate* is given by $\dot{Q} = \frac{k \cdot A}{L} \cdot (T_1 - T_2)$, where $k$ is the thermal conductivity of the material, $A$ is the area of the conduction, $L$ is the thickness of the material through which the conduction occurs, and $T_1, T_2$ are the temperatures of the two different media.

- *Convection* occurs when a moving fluid transports heat from a hotter object to a colder object. The rate of heat convection is proportional to the temperature difference, and is expressed by Newton's law of cooling as $\dot{Q} = h \cdot A \cdot (T_1 - T_2)$, where $h$ is the convection heat transfer coefficient, and $A$ is the surface area through which convection heat transfer takes place.

- *Radiation* occurs directly through the components' surface, without any transfer medium. The rate of heat radiation is given by $\dot{Q} = \varepsilon \cdot \sigma \cdot A \cdot (T_1^4 - T_2^4)$, where $\varepsilon$ is the emissivity of the surface, $\sigma$ is the Stefan-Boltzmann constant, and $A$ is the surface area through which radiation heat transfer takes place.

For a running example, consider a cup of hot coffee in a room. If the temperature of the coffee is higher than that of the room, heat will flow from the coffee to the room, until both of them reach the same temperature. In particular, the heat flows through the cup wall by conduction and radiation, and from the surface of the coffee (assuming there is no lid on the cup) to the room by convection.

**Phase transition.**   Another important aspect that must be taken into account when modeling thermal systems is the different *phases* of a substance. For example, water has three distinct phases: *solid* (ice), *liquid*, and *gas* (vapor). The *latent heat* is the amount of energy released or absorbed by a chemical substance during a phase transition. Note that during a *phase transition* the temperature does not change; the flow of energy goes in as the latent heat of the transition process. For example, to change its phase from solid to liquid, water goes through the melting process which consumes some energy.

## 4   A Framework for Modeling Physical Systems

This section presents a high-level overview of the general method we use to model physical systems and to approximate their behaviour. This method adapts the effort-and-flow-variable approach common
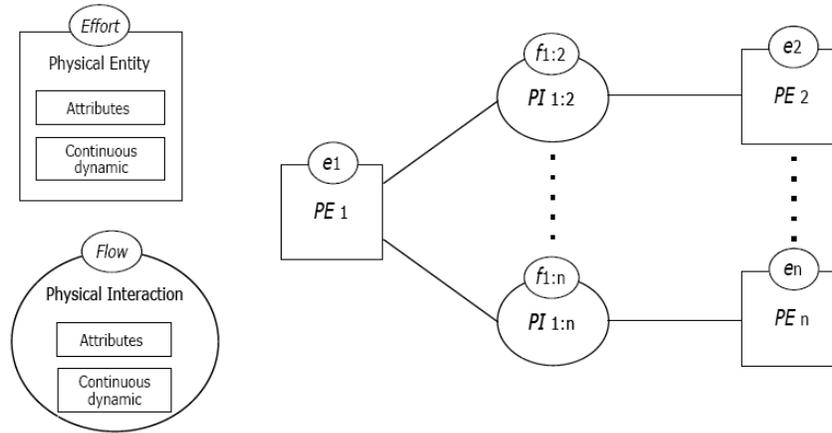
Figure 1: Physical system components.

in the field of physical systems modeling (see, e.g., [17]). In particular, Section 4.1 briefly describes the modeling of physical systems in this framework, and Section 4.2 shows how our method can be instantiated to model thermal systems.

## 4.1 Modeling Physical Systems

A well known modeling approach for physical systems is based on two kinds of variables in the model specification: *effort* and *flow* variables [17].

The components of a physical system can be thought of as energy manipulators which process the energy injected into the system. These energy manipulators interact through energy ports. For this purpose, effort and flow variables are used to represent the power being transmitted through energy ports. The flow variable is associated with the act of delivering energy, whereas the effort variable is associated with the act of measuring the flow of energy. For example, our coffee system could have two effort variables, one denoting the temperature of the coffee and one denoting the temperature of the room, and one flow variable, denoting the flow of heat from the coffee to the room per time unit.

The approach using effort/flow variables is applicable to different areas of physical systems. In mechanical translation systems, the pair of effort and flow variables are force and velocity; in mechanical rotation systems, torque and angular velocity; in electrical systems, voltage and current; in fluidic systems, pressure and volume flow rate; in thermal systems, temperature and heat flow rate [1].

Our idea is to think of a physical system as the combination of two kinds of components: *physical entities* and *physical interactions*, as shown in Fig. 1.

A *physical entity* (such as a cup of coffee) consists of a set of *attributes* and a *continuous dynamics*. We consider three kinds of attributes: *continuous* variables (denoting physical quantities, such as the temperature, that change with time), *discrete* variables, and *constants*. A physical entity has one *effort variable*, which is a continuous variable and the "main" attribute of the physical entity. The *continuous dynamics* of a physical entity is given by a differential equation which is used to compute the value of the effort variable. A physical entity can have one or more physical interactions with one or more physical entities. The values of the flow variables of these interactions are used in the computation of the continuous dynamics of the physical entity.

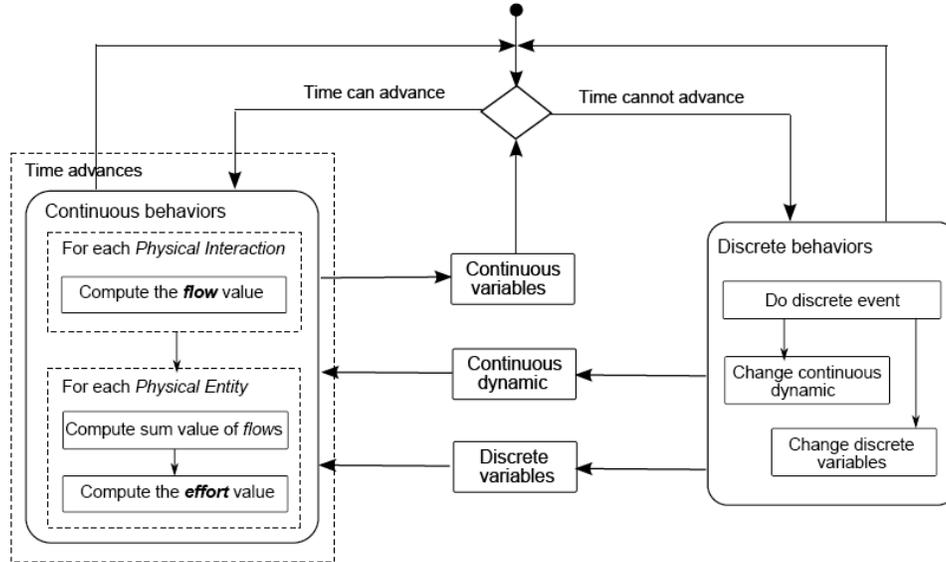A *physical interaction* (such as the heat flow from the coffee to the room) represents an interaction

Figure 2: Physical system behaviors.

between two physical entities. It consists of one *flow variable*, a set of *attributes*, and a *continuous dynamics*. The flow variable is a continuous variable. The continuous dynamics is described by a differential equation. The values of the effort variables from the two physical entities are used in the computation of the continuous dynamics of the physical interaction.

### 4.1.1 Physical System Behavior

The basic behavior of physical system components is their continuous behavior. However, there are natural phenomena that exhibit a combination of continuous and discrete behaviors. In our method, the continuous behavior is performed as long as time advances. When a discrete event must happen, time cannot advance until the event has been performed. Figure 2 shows the execution of physical system behaviors.

**Approximating continuous behaviors.** We use a numerical approach to approximate the continuous behaviors of the physical system components by advancing time in discrete time steps, and computing the values of the continuous variables at each "visited" point in time. There are two main steps in the computation of the continuous behavior which is performed in each time step. The first step is to compute the new value of the flow variable of each physical interaction in the system. This computation is based on the continuous dynamics of the physical interaction, the values of its attributes, and the "old" values (i.e., the values computed at the previous visited point in time) of the effort variables of the two physical entities connected to the interaction. In the configuration of physical system components in Figure 1, the new values of the flow variables $f_{1:2}$ to $f_{1:n}$ of the physical interactions $PI_{1:2}$ to $PI_{1:n}$, respectively, are computed. The computation of the new value of $f_{1:2}$ of $PI_{1:2}$, for example, is not only based on the attributes of the physical interaction, but also on the "old" values of $e_1$ of $PE_1$ and $e_2$ of $PE_2$.

The second step is to compute the "new" value of the effort variable of each physical entity in the system. Beside the current value of the effort variable, the continuous dynamic of the physical entity, and

the values of its attributes, the computation also needs the sum of the (new) values of the flow variables of all the physical interactions connected to it. For example, in the system in Fig. 1, the new values of the effort variables $e_1$ to $e_n$ of the physical entities $PE_1$ to $PE_n$, respectively, are computed. The computation of the new value of $e_1$ of $PE_1$, for example, also depends on the sum of the new values of $f_{1:2}$ to $f_{1:n}$.

The difference between computing the new values of the effort variables and the flow variables is that for the former we need a differential equation solution technique, whereas for the latter we do not need such a technique since a flow variable represents a rate.

**Discrete behaviors.** The conditions for triggering discrete events depend on the values of the continuous variables. We consider two kinds of discrete events: one that changes the continuous dynamics of a physical entity or interaction (e.g., when the coffee starts boiling, its continuous dynamics is changed), and one that changes the value of a discrete variable. The change caused by a discrete event will effect the computation of the continuous behavior in the next time step.

## 4.2 Applying the Framework for Modeling Thermal Systems

We can instantiate the above framework to thermal systems as follows: a physical entity is a *thermal entity*, whose effort variable ($T$) denotes the temperature of the entity and whose continuous dynamics is a differential equation defining the heat gained or lost by the entity as its temperature changes. Likewise, a physical interaction is a *thermal interaction* whose flow variable ($\dot{Q}$) denotes the heat flow rate. We may have different kinds of thermal interactions: conduction, convection, and radiation; their continuous dynamics are given by differential equations used for the heat transfer rates. Table 1 shows the thermal system components, and Fig. 3 shows a model of our example with a hot cup of coffee in a room.

Table 1: Thermal system components.

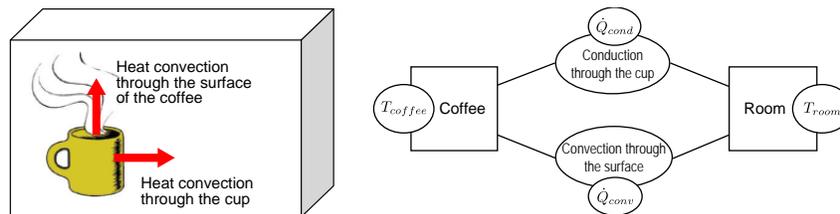| Components | Effort/Flow | Attributes | Continuous dynamic |
|---|---|---|---|
| Thermal entity | Effort: $T$ | $m$, $c$ | $\dot{T} = \frac{\Sigma \dot{Q}}{m \cdot c}$ |
| Thermal interaction: conduction | Flow: $\dot{Q}$ | $A$, $k$, $L$ | $\dot{Q} = \frac{k \cdot A}{L} \cdot (T_1 - T_2)$ |
| Thermal interaction: convection | Flow: $\dot{Q}$ | $A$, $h$ | $\dot{Q} = h \cdot A \cdot (T_1 - T_2)$ |
| Thermal interaction: radiation | Flow: $\dot{Q}$ | $A$, $\varepsilon$, $\sigma$ | $\dot{Q} = \varepsilon \cdot \sigma \cdot A \cdot (T_1^4 - T_2^4)$ |



Figure 3: A cup of coffee in a room with the convection and conduction heat transfers between them.

# 5  Modeling Thermal Systems in Real-Time Maude

This section explains how we can apply the general methods for modeling physical systems given in Section 4 to model and execute thermal systems in Real-Time Maude. In particular, Section 5.1 defines classes for thermal entities and interactions, Section 5.2 explains how the Euler method (see, e.g., [5]) can be used to define in Real-Time Maude an approximation of the continuous dynamics of a thermal system, Section 5.3 describes how hybrid behaviors of thermal systems can be modeled, and Section 5.4 explains how external heat sources can be easily added to our models.

## 5.1  Defining Thermal System Components

In our method, thermal entities (physical entities in thermal systems) can be defined in Real-Time Maude as object instances of the following class `ThermalEntity`:

```
class ThermalEntity | heatCap : PosRat,   mass : PosRat,
                      temp : Rat,         tempDisplay : String,
                      heatTrans : Rat,    mode : CompMode .

sort CompMode .    op default : -> CompMode [ctor] .
```

The attribute `heatCap` denotes the heat capacity and `mass` the mass of the entity; `temp` denotes the temperature, which corresponds to the effort variable in thermal systems; `tempDisplay` is used to display the temperature in a more readable format than as a rational number; finally, `heatTrans` and `mode` are needed, as we will see later, to implement phase transitions, i.e., to encode the computation mode of a thermal entity, which is related to its discrete behavior.

We define a class for general *thermal interactions* (physical interactions in thermal systems), and three subclasses for the three different heat transfer mechanisms:

```
class ThermalInteraction | entity1 : Oid,   entity2 : Oid,   area : PosRat,
                           Qdot : Rat,      QdotDisplay : String .

class Conduction | thermCond : PosRat, thickness : PosRat .
class Convection | convCoeff : PosRat .
class Radiation | emmissiv : PosRat .

subclass Conduction Convection Radiation < ThermalInteraction .
```

In the `ThermalInteraction` class, `Qdot` is the attribute corresponding to the heat flow rate $\dot{Q}$ of the thermal interaction. `QdotDisplay` is used to display the heat flow rate in a readable format. `entity1` and `entity2` are object identifiers of the two thermal entities connected by an object of this class, and `area` is the area of the interaction common for all three interaction types. The remaining interaction-specific attributes, e.g., for conductivity the thermal conductivity `thermCond` of the material and the thickness `thickness` of the material through which the conduction occurs, are specified in the corresponding subclasses.

## 5.2  Approximating the Continuous Behaviors using the Euler Method

We use ordinary differential equations for specifying the continuous dynamics of physical entities and interactions. For executing the continuous behavior of a physical system by using a discrete-time sampling strategy, we need an approximation to compute the values of the effort and flow variables. In this

paper we adapt the Euler method [5], a numerical approach to solve differential equations. Assume that the value of the continuous variable $y$ at time $t$ is defined by the differential equation $\dot{y} = f(y,t)$. Assume furthermore that the value of $y$ at time $t_n$ is $y_n$. When time advances, the Euler method uses the function $y_{n+1} = y_n + h \cdot f(y_n, t_n)$ to compute the value of $y$ at time point $t_{n+1} = t_n + h$, where $h$ is the time step.

To implement the Euler method, we define the following functions :

- `computeQdot` that computes the heat flow rate $\dot{Q}$ of each thermal interaction,

- `computeTemp` that computes the temperature $T$ of each thermal entity in the system,

- `sumQdots` that computes the sum of the $\dot{Q}$s of the thermal interactions of each entity, and

- `ODESol` that computes the next value $y_{n+1} = y_n + h * f(y_n, t_n)$ of a continuous variable $y$ using the Euler method.

We start with the declaration of the following variables:

```
vars T T1 T2 QDOT YN FYN QT NEW-Qdot CURR-TEMP NEW-HEAT-TRANS : Rat .
vars k A L EPSILON c m h : PosRat .      var S : OnOff .
vars E E1 E2 TI HG : Oid .               vars CONFIG REST : Configuration .
```

The function `computeQdot` computes the heat flow rate $\dot{Q}$ of each thermal interaction in the system according to the laws given in Section 3 and Table 1. The following equation computes the `Qdot` (given by $\dot{Q} = \frac{k \cdot A}{L} \cdot (T_1 - T_2)$ as explained in Section 3) of a thermal *conduction* `TI` between two thermal entities `E1` and `E2`, and then recursively applies the `computeQdot` function on the remaining configuration:

```
op computeQdot : Configuration ~> Configuration .
ceq computeQdot(
   < E1 : ThermalEntity | temp : T1 >
   < E2 : ThermalEntity | temp : T2 >
   < TI : Conduction | entity1 : E1, entity2 : E2, thermCond : k, area : A, thickness : L >
   REST)
 =
   < TI : Conduction | Qdot : NEW-Qdot, QdotDisplay : display(NEW-Qdot, precision) >
   computeQdot(< E1 : ThermalEntity | >  < E2 : ThermalEntity | >  REST)
if NEW-Qdot := (k * A * (T1 - T2)) / L .
```

Likewise, the following two equations compute the new value of `Qdot` of, respectively, thermal *convections* and *radiations*, and the last equation removes the `computeQdot` operator when there are no more thermal interactions in the remaining configuration:

```
ceq computeQdot(
   < E1 : ThermalEntity | temp : T1 >
   < E2 : ThermalEntity | temp : T2 >
   < TI : Convection | entity1 : E1, entity2 : E2, convCoeff : h, area : A > REST)
 =
   < TI : Convection | Qdot : NEW-Qdot, QdotDisplay : display(NEW-Qdot, precision) >
   computeQdot(< E1 : ThermalEntity | >  < E2 : ThermalEntity | >  REST)
if NEW-Qdot := h * A * (T1 - T2) .

ceq computeQdot(
   < E1 : ThermalEntity | temp : T1 >
   < E2 : ThermalEntity | temp : T2 >
   < TI : Radiation | entity1 : E1, entity2 : E2, emmissiv : EPSILON, area : A > REST)
 =
```

```
    < TI : Radiation | Qdot : NEW-Qdot, QdotDisplay : display(NEW-Qdot, precision) >
    computeQdot(< E1 : ThermalEntity | >  < E2 : ThermalEntity | >  REST)
if NEW-Qdot := EPSILON * stefBolzConst * A * ((T1 ^ 4) - (T2 ^ 4))  .
```

```
eq computeQdot(CONFIG) = CONFIG [owise] .
```

The function `computeTemp` computes the temperature $T$ of each thermal entity in the system, with the dynamics given by $\dot{T} = \frac{\sum \dot{Q}}{m \cdot c}$, where $\sum \dot{Q}$ is the sum of the $\dot{Q}$s of the thermal interactions of the entity which is computed by the function `sumQdots` below. The function uses a numerical method (`ODESol`) which needs the value of the current temperature (T), the time step used, and the continuous dynamics:

```
op computeTemp : Configuration ~> Configuration .
ceq computeTemp(< E : ThermalEntity | heatCap: C, mass: M, temp: T, mode: default >  REST)
    =
    < E : ThermalEntity | temp : CURR-TEMP, tempDisplay : display(CURR-TEMP, precision) >
    computeTemp(REST)
    if CURR-TEMP := ODESol(T, timeStep, sumQdots(REST, E) / (M * C)))  .
```

```
eq computeTemp(CONFIG) = CONFIG [owise] .
```

The function `sumQdots` computes the sum values of the heat flow rate of all thermal interactions connected to a thermal entity. The direction of the heat flow determines whether a thermal entity gains or loses the heat. This is represented by multiplying the heat flow rate value by minus one for one part of the interaction:

```
op sumQdots : Configuration Oid  ~> Rat .
eq sumQdots(< TI : ThermalInteraction | entity1 : E1, entity2 : E2, Qdot : QDOT >  REST, E)
    =
    if (E == E1 or E == E2) then
        (if E == E1 then -1 * QDOT + sumQdots(REST, E) else QDOT + sumQdots(REST, E) fi)
    else sumQdots(REST, E) fi .
```

```
eq sumQdots(CONFIG, E) = 0 [owise] .
```

The function `ODESol` uses the Euler method to compute the next value $y_{n+1} = y_n + h * f(y_n, t_n)$ of a continuous variable $y$, given the current value $y_n$ of the variable, the time step $h$, and the value $f(y_n, t_n)$ computed from the continuous dynamics of the variable:

```
op ODESol : Rat PosRat Rat -> Rat .
eq ODESol(YN, H, FYN) = YN + H  * FYN .
```

As explained in Section 2, a *tick rule* models the advance of time in a system. We define the following tick rule that advances time by one time step and computes the new temperatures for all entities:

```
rl [tick] : {CONFIG} => {computeTemp(computeQdot(CONFIG),timeStep))} in time timeStep .
```

### 5.3   Defining Thermal System with Hybrid Behaviors

As mentioned in Section 4.1.1, a physical system may also exhibit (at least) two kinds of discrete behaviors. In this paper we focus on discrete events that change the continuous dynamics of an entity or interaction. In thermal systems, we have *phase transitions* between different phases of the entities. For the example of water, a discrete event changes the phase from "solid" to "melting," thereby also changing the continuous dynamics of the water: the received thermal energy is used for the melting process and the temperature of the water does not change during melting. The following changes need to be made to our model to accommodate such hybrid behaviors in physical systems:

1. The basic `ThermalEntity` class must be extended to keep track of the phases of the entity.

2. Instantaneous rewrite rules modeling the discrete change from one phase to another must be added.

3. It must be ensured that the above rules are applied at the right time.

4. The functions computing the values of the continuous variables must take into account the different continuous dynamics in different phases.

**Extending the basic class.**    Let's say we want to define a thermal entity representing a water substance with three phases: solid, liquid, and gas. Between these main phases, the water is either melting, freezing, evaporating, or condensing. Figure 4 shows the various phases (and their corresponding continuous dynamics) and the transitions between them.
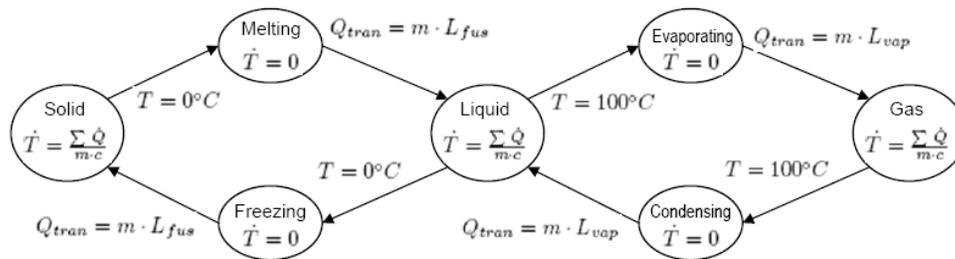


Figure 4: The phases of the water entity.

The class `WaterEntity` extending the base class `ThermalEntity` can be defined as follows:

```
class WaterEntity | phase : MatterState,  heatTrans : Rat,  heatTransDisplay : String .
subclass WaterEntity < ThermalEntity .

sort MatterState .
ops liquid solid gas melting evaporating condensing freezing : -> MatterState .
op phaseChange : -> CompMode .
```

The new attribute `phase` denotes the current phase of the water substance, and `heatTrans` denotes the latent heat of the water in the transitions between the three main phases. Notice that the continuous dynamics of the temperature is the same in all the three main phases of water, whereas the temperature does not change in-between these phases. In addition to the `default` computation mode defined above, we also add a new computation mode `phaseChange` to denote these intermediate phases of water, so that the `computeTemp` function is defined correctly.

**Rewrite rules defining discrete change.**    A phase change is modeled by an instantaneous rewrite rule. We show two of the eight such rules (corresponding to the arrows in Fig. 4) for water:

```
crl [solid-to-melting] :
   < E : WaterEntity | temp : T, phase : solid >
  =>
   < E : WaterEntity | phase : melting, mode : phaseChange, heatTrans : 0 >
   if T >= 0 .

crl [melting-to-liquid] :
```

```
 < E : WaterEntity | phase : melting, heatTrans : QT, mass :  m >
=>
 < E : WaterEntity | phase : liquid, mode : default >
if QT / m >= latentHeatFus .
```

The change from a "main" phase to a "transitional" phase happens when the temperature reaches a given value, whereas a transition the other way happens when the value of the heat accumulated during the transitional phase divided by the mass of the entity reaches some value (such as the latent heat fusion).

We ensure that the above rules are applied when they should by disabling the tick rule when one of the above rules are enabled. This is done by defining a function `timeCanAdvance` on the state, so that `timeCanAdvance` holds only if no such instantaneous rule needs to be applied in the current state. We then modify the tick rule so that it can only be applied to states where `timeCanAdvance` holds. Again, we only show two of the equations defining `timeCanAdvance` and the modified tick rule:

```
eq timeCanAdvance(< E : WaterEntity | phase : solid, temp : T >) =  T < 0 .
eq timeCanAdvance(< E : WaterEntity | phase : melting, heatTrans : QT, mass :  m >) =
     QT / m < latentHeatFus .

crl [tick] : {CONFIG} => {computeTemp(computeQdot(CONFIG), timeStep)} in time timeStep
            if timeCanAdvance(CONFIG) .
```

The change of continuous dynamics caused by a phase change means that there is more than one possible continuous dynamics for a thermal system component. We need to modify the function used to compute the corresponding continuous variable. For the water entity, we need to modify the function `computeTemp`, so that when the entity is in mode `phaseChange` (as opposed to the `default` mode, where the equation in Section 5.2 applies), then the temperature does not change, but the accumulated heat is increased/decreased:

```
ceq computeTemp(< E : WaterEntity | mode : phaseTrans, heatTrans : QT > REST) =
      < E : WaterEntity | heatTrans : NEW-HEAT-TRANS,
                          heatTransDisplay : display(NEW-HEAT-TRANS, precision) >
      computeTemp(REST)
   if NEW-HEAT-TRANS := ODESol(QT, timeStep, sumQdots(REST, E)) .
```

## 5.4   External Heat Sources

It is sometimes convenient to be able to inject heat into a thermal entity. For example, we may want to add a boiler to our coffee system that adds heat to the cup of coffee. In our method, such heat generators can be defined in Real-Time Maude as object instances of the following class `HeatGenerator`:

```
class HeatGenerator | Qdot : Rat, entity : Oid .
```

The attribute `Qdot` denotes the heat flow rate provided by the heat generator; `entity` is the object identifier of the thermal entity connected to the heat generator.

Adding a heat generator to a thermal entity means that beside the heat flows from the interactions with other thermal entities, the heat flow from the heat generator influences the change of the temperature of the entity. We need to modify the function used to compute the sum of heat flow rate values of a thermal entity to include the heat flow rate from the connected heat generator:

```
eq SumQdots((< HG : HeatGenerator | Qdot : QDOT, entity : E1 > REST), E) =
     if (E == E1) then QDOT + SumQdots(REST, E) else SumQdots(REST, E) fi .
```

Table 2: Initial values for object constant parameters ($\pi = \frac{22}{7}$).

| room | | | | cup of coffee | | | |
|---|---|---|---|---|---|---|---|
| airDensity | air density [a] | $\frac{12}{10}$ | $kg/m^3$ | cupRadius | inner radius | $\frac{4}{100}$ | $m$ |
| roomVolume | volume | $64$ | $m^3$ | cupHeight | height | $\frac{9}{100}$ | $m$ |
| roomMass | mass | $\frac{384}{5}$ | $m^3$ | cupCircum | circumference | $\frac{44}{175}$ | $m$ |
| roomHC | heat capacity | $\frac{105}{10}$ | $kJ/kg$ | cupBaseArea | circle area | $\frac{22}{4375}$ | $m^2$ |
| h | h[b] | $\frac{20}{1000}$ | $\frac{kW}{m^2\cdot{}^\circ C}$ | cupSideArea | side area | $\frac{99}{4375}$ | $m^2$ |
| | | | | cupThickness | cup thickness | $\frac{1}{200}$ | $m$ |
| | | | | waterDensity | water density | $1000$ | $kg/m^3$ |
| | | | | coffeeVolume | volume | $\frac{99}{218750}$ | $m^2$ |
| | | | | coffeeMass | mass | $\frac{396}{875}$ | $kg$ |
| | | | | coffeeHC | heat capacity | $\frac{42}{10}$ | $kJ/kg$ |
| | | | | k | k[c] | $\frac{15}{10000}$ | $\frac{kW}{m\cdot{}^\circ C}$ |

[a]At sea level and $20^\circ C$, air has a density of approximately $1.2 kg$ (http://en.wikipedia.org/wiki/Density_of_air).
[b]Free convection of gases is between 2-25 $\frac{W}{m^2\cdot{}^\circ C}$ .
[c]Thermal conductivity of porcelain

## 6 Case Studies

This section shows how our method can be used to model and analyze some simple thermal systems. In particular, in Section 6.1 we model and analyze a cup of hot coffee in a room, with two kinds of interactions (*conduction* and *convection*). In Section 6.2 we add a heater that sends constant heat to the cup of coffee. Finally, in Section 6.3 we add a "smart" heater that switches itself *on* or *off* in order to keep the temperature of the coffee in a desired interval. It is worth noticing that, using the definitions in the previous section, the definition of the first two models reduces to defining appropriate initial states. Table 2 shows the constants used for modeling the room and the cup of coffee. The analyses are performed on an *AMD Athlon(tm) 64 Processor 3500+* with 2GB of RAM, using a `timeStep` of 1, unless something else is specified.

### 6.1 Case Study 1: A Cup of Coffee in a Room

In our first case study we model the simple coffee system in Fig. 3 in Section 4.2 by defining the corresponding initial state as follows:

```
op cs1 : -> GlobalSystem .
eq cs1 =
  {< coffee : ThermalEntity | heatCap : coffeeHC, mass : coffeeMass, heatFlow : 0,
                              temp : 70, tempDisplay : "", mode : default, heatTrans : 0 >
   < room : ThermalEntity | heatCap : roomHC, mass : roomMass, temp : 20,
                            tempDisplay : "", heatFlow : 0, mode : default, heatTrans : 0 >
   < crConduct : Conduction | entity1 : coffee, entity2 : room, thermCond : k, Qdot : 0,
                              area : condArea, thickness : cupThickness, QdotDisplay : "" >
   < crConvect : Convection | entity1 : coffee, entity2 : room, convCoeff : h,
                              area : convArea, Qdot : 0, QdotDisplay : "" >} .
```

where `coffeeHC, coffeeMass, ...` are constants of sort `Rat` with values as shown in Table 2. The constant `condArea` is the conduction area, computed as the sum of the coffee circle area on the base of the cup and the coffee cylindrical area on the lateral surface of the cup. The constant `convArea` is the convection area, equal to the coffee circle area, representing the top surface of the coffee.

We can simulate the behavior from this initial state up to a certain time, using Real-Time Maude's timed rewrite command:

```
Maude> (trew cs1 in time <= 1000 .)

result ClockedSystem :
  {< crConduct : Conduction | QdotDisplay : "0.0044820616", ... >
   < crConvect : Convection | QdotDisplay : "0.0000543280", ... >
   < coffee : ThermalEntity | TempDisplay : "21.6767974687", ... >
   < room : ThermalEntity | TempDisplay : "21.1390469168", ... >} in time 1000
```
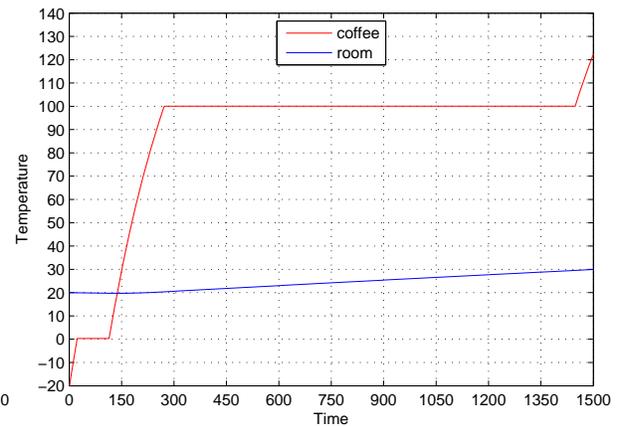
We show only the *display* attributes of the objects. The simulation takes about a minute and a half. The result is as expected: in 1000 seconds, the temperature of the coffee is sensibly decreased, reaching almost that of the room, whose temperature has increased slightly due to the heat from the coffee.

Moreover, we can collect the values for coffee and room temperatures at each time step during the simulation. This has been done by adding a "collector" object to the initial configuration, which, at each tick rule application, records the current time and temperature values. This allows us to plot the coffee and room temperatures as a function over time, as shown in Figure 5(a).
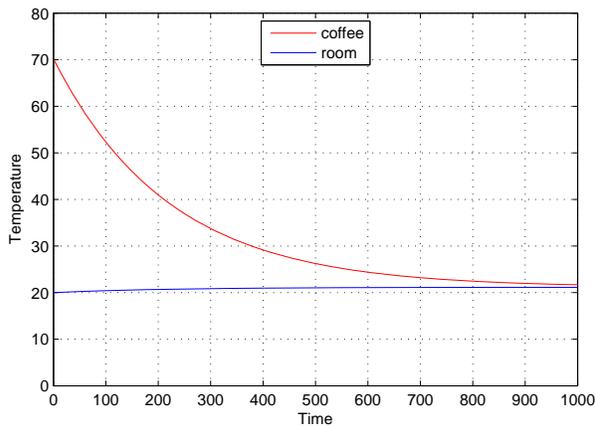


(a) Case study 1: simulation                    (b) Case study 2: simulation

Figure 5: Simulation plots for the case studies 1 and 2.

Suppose we would like to check how long it takes for the coffee and room to reach about the same temperature, with a given tolerance. This can be done by searching for one reachable state where the difference of the two temperatures is less than the given tolerance. This kind of search can be performed in Real-Time Maude without any restriction on time, since we can always reach such a desired state within a finite amount of time:

```
Maude> (tsearch [1] cs1 =>*
          {< coffee : ThermalEntity | temp : Tcoffee:Rat >
           < room : ThermalEntity | temp : Troom:Rat >  C:Configuration}
             such that (abs(Tcoffee:Rat - Troom:Rat) <= 1/1000 ) with no time limit .)
```
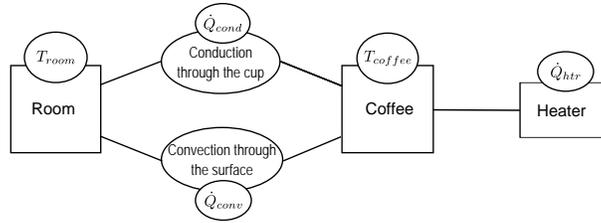
Figure 6: Case study 2: Modeling a cup of coffee in a room with a manual coffee heater.

It takes about 17 minutes for Real-Time Maude to find a solution, which indicates that it takes around 2340 seconds for the room and coffee to reach the same temperature.

## 6.2 Case Study 2: Adding a Boiler to the Cup of Coffee

We extend our first case study by adding a heater that is always on and therefore sends a constant flow of heat to the cup of coffee, as illustrated in Fig. 6. Furthermore, we start with an initial coffee temperature of $-10°C$, so that this example shows how the coffee entity exhibits a *hybrid* behavior, when given a sufficient constant heat, going through all three main phases: solid, liquid, and gas.

We now declare the `coffee` object to be a `WaterEntity` object, and add an object `boiler` of class `HeatGenerator`. The other objects (`room`, `crConduct`, and `crConvect`) are defined as in Section 6.1:

```
op cs2 : -> GlobalSystem .
eq cs2 =
  {< coffee : WaterEntity | heatCap : coffeeHC, mass : coffeeMass, temp : -10,
                            tempDisplay : "", heatFlow : 0, mode : default, phase : solid,
                            heatTrans : 0, heatTransDisplay : "" >
  ...
  < boiler : HeatGeneration | entity : coffee, Qdot : 15/10 >} .
```

As for the first case study, we can simulate the above system and plot the room and coffee temperatures as functions of time. Figure 5(b) shows the simulation up to time 1500. We notice that the coffee temperature remains constant during the transition phases (melting and evaporation), while it increases when the coffee is in liquid, solid, or gaseous state. The fact that the melting temperature is somewhat greater than 0 $°C$ is a consequence of the approximation in the computation of the continuous behavior and the chosen `timeStep`: a bigger `timeStep` will lead to a coarser numerical approximation in the Euler algorithm. The room tends to become warmer, compared to the first case study; this is due to the presence of the heater, which, through the coffee, indirectly transfers heat to the room.

If we would like to know how long it takes for the iced coffee to start to melt, we can use Real-Time Maude's `find earliest` command to find the first reachable state such that the coffee is `melting`:

```
Maude> (find earliest cs2 =>* {C:Configuration < coffee : WaterEntity | phase : melting >} .)
```

In 224*ms* Real-Time Maude returns a solution showing that the ice coffee starts to melt after 22 seconds.

## 6.3 Case Study 3: Keeping the Coffee Warm

We keep the set-up of the second case study, but now we define a more sophisticated heater, namely, one that senses the coffee temperature at each time step and tries to keep the temperature of the coffee between $70°C$ and $80°C$ by turning itself on and off. We need to modify the previous model by:

- adding two instantaneous rules: one for turning the heater on when the coffee temperature is below $70°C$, and one for turning the heater off when the coffee is warmer than $80°C$, and

- adding a condition to the tick rule so that time does not advance when one of the above rules should be taken.

For convenience, we define a subclass `SmartHeater` of the class `HeatGenerator` for such smart heaters:

```
class SmartHeater | status : OnOff, lowTemp : Rat, highTemp : Rat .
subclass SmartHeater < HeatGenerator .

sort OnOff .   ops on off : -> OnOff .
```

The rules for turning the heater on and off are immediate:

```
crl [turnOff] :
    < E : ThermalEntity | temp : T1 >
    < HG : SmartHeater | entity : E, status : on , highTemp : T2 >
   =>
    < E : ThermalEntity |  >
    < HG : SmartHeater | status : off , Qdot : 0 >
   if T1 >= T2 .

crl [turnOn] :
    < E : ThermalEntity | temp : T1 >
    < HG : SmartHeater | entity : E, status : off , lowTemp : T2 >
   =>
    < E : ThermalEntity |  >
    < HG : SmartHeater | status : on , Qdot : 15/10 >
   if T1 <= T2 .
```

We also define a function `heatersOK`, which holds in a state when none of the above two rules can be applied, and modify the tick rule to take this into account:

```
op heatersOK : Configuration -> Bool [frozen (1)] .

eq heatersOK(< HG : SmartHeater | entity : E, status : S, lowTemp : T1, highTemp : T2 >
             < E : ThermalEntity | temp : T >  REST)
   = ((S == on and T < T2) or (S == off and T > T1))
       and heatersOK( < E : ThermalEntity | >  REST) .
eq heatersOK(CONFIG) = true [owise] .

crl [tick] : {CONFIG} => {computeTemp(computeQdot(CONFIG), timeStep)} in time timeStep
             if timeCanAdvance(CONFIG) and heatersOK(CONFIG) .
```

The initial state is as in the second case study, with the exception that the "dumb" heater has been replaced by a smart heater:

```
op cs3 : -> GlobalSystem .
eq cs3 =
  {< coffee : WaterEntity | temp : -20, phase : liquid, ... >
   < coffeeHeater : SmartHeater | entity : coffee, status : off, Qdot : 0, lowTemp : 70,
                                  highTemp : 80 >
   ...} .
```
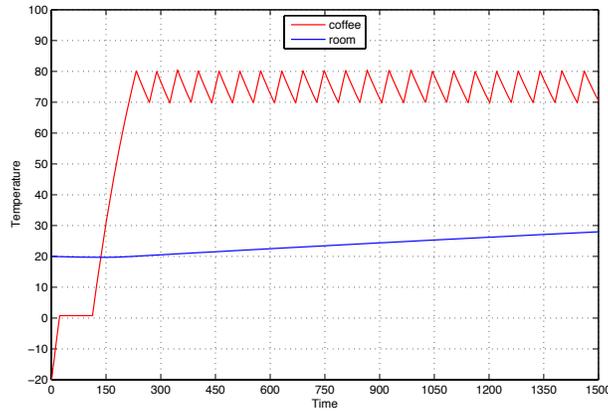
Figure 7: Case study 3: Simulation.

Figure 7 shows the coffee and room temperatures as functions of the elapsed time in a Real-Time Maude simulation of this system.

Finally, we use time-bounded LTL model checking to analyze the stability property that once the temperature of the coffee has reached $69.5°C$, its temperature will remain in the interval between $69.5°C$ and $80.5°C$ (we set the interval as 69.5 to 80.5 instead of 70 to 80 for taking into account the imprecision from the approximation using the Euler method). We define the atomic proposition `temp-ok` to hold in all states where the coffee temperature is between $69.5°C$ and $80.5°C$:

```
op temp-ok : -> Prop [ctor] .
ceq {REST < coffee : WaterEntity | temp : T >} |= temp-ok = (T >= 139/2 and T <= 161/2) .
```

The property can be checked by executing using time-bounded model checking:

```
Maude> (mc cs3 |=t [] (temp-ok -> [] temp-ok) in time <= 1500 .)

Result Bool :
  true
```

# 7 Related Work

Modelica [3] is an object-oriented language for modeling physical systems where hybrid differential algebraic equations model the continuous dynamics. The language supports the specification of linear and non-linear equations. Since the language does not have a formal semantics, the precise meaning of a model depends on the simulation tool used. Furthermore, there are no reachability and temporal logic analysis tools for Modelica.

HyVisual [8] is an actor-based tool environment for the simulation of continuous and hybrid systems. Hybrid behaviors are specified as finite state machines in which a state can be refined into a dynamic system represented as ordinary differential equations. HyVisual does not provide model checking analyses.

OO-DPT [16] is an object-oriented approach for modeling hybrid supervisory systems. This approach proposes a Petri-net-based formal technique named *object-oriented differential predicate transition net* (OO-DPT net). This technique defines an interface between differential equation systems and Petri nets to model hybrid behaviors. Reachability and safety properties analysis is the target of this approach, but as of yet, there is no tool support for OO-DPT.

HyTech [4] is a tool for modeling and verifying hybrid systems represented as linear hybrid automata. The discrete behaviors are represented as a set of locations and discrete transitions between them where the continuous dynamics is defined in each location. The continuous dynamics is specified as polyhedral differential inclusions. The restrictions related to the formalism used by the tool confines the representation of continuous dynamics to the ones usually used in modeling physical systems. The main difference between HyTech and Real-Time Maude is the expressiveness and generality of the formalism, where in Real-Time Maude all kinds of data types, functions, communication models, dynamic object creation, etc., can be specified.

SHYMaude (Stochastic Hybrid Maude) [10] is a modeling language for object-based stochastic hybrid systems that extends the PMaude [7] tool for probabilistic rewrite theories. Stochastic hybrid systems consist of distributed stochastic hybrid objects that interact with each other by asynchronous message passing. The continuous behaviors are governed by stochastic differential equations, and the discrete changes are probabilistic. For simulation, the continuous dynamics of stochastic differential equations is approximated by using the Euler-Maruyama method, a generalization of Euler method to approximate numerical solution of stochastic differential equations. For formal analyses, the interface of the statistical model checking tool VeStA [15] to Maude system is used. Despite the obvious similarities that both approaches are object-oriented approaches to hybrid systems in extensions of Maude, our work and the SHYMaude work are in fact quite different. In [10] the hybrid objects are autonomous and only interact with other hybrid objects by message passing, whereas in our work, the main focus is on the *physical interaction* (such as heat flow) between the hybrid objects.


# 8   Concluding Remarks

We have presented a general object-based method for formally modeling and analyzing thermal systems in Real-Time Maude. In contrast to most other approaches, we also focus on the physical interactions between different physical components. We explain how the Euler method can be adapted to approximate continuous behaviors, and have illustrated our method on three variations of a simple coffee-and-room system with realistic parameters.

This work should be seen as a first exploration into how hybrid systems can be modeled and analyzed in Real-Time Maude. Although we believe that Real-Time Maude – because of its expressiveness and generality, support for objects, user-definable data types, different communication models, etc. – should be a suitable candidate for modeling and analyzing advanced hybrid systems; this of course has to be validated by applying our techniques on advanced hybrid systems.

Euler is a simple numerical algorithm for solving ordinary differential equations (ODE). It could be interesting to implement in Real-Time Maude some other numerical approaches for solving ODE (e.g. the midpoint method, the linear multi-step method, or the Runge-Kutta method) that offer more precise results than Euler, probably at the cost of computational efficiency. Another point to explore is the possibility of using the other time sampling strategies offered by Real-Time Maude, like the *maximum time advance* strategy; applying this strategy to continuous behavior would require an execution strategy that can "predict" the maximum time that the system can advance, before some instantaneous rule has to be taken, possibly using an advanced numerical method for ODE with *dynamic step*.


# References

[1]  J.P. Bentley (2005): *Principles of Measurement Systems*. Pearson Education Limited, 4th ed. edition.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer & C. Talcott (2007): *All About Maude - A High-Performance Logical Framework*, Lecture Notes in Computer Science 4350. Springer.

[3] P. Fritzson (2003): *Principles of Object-Oriented Modeling and Simulation with Modelica*. Wiley-IEEE Computer Society Pr.

[4] T.A. Henzinger, P. Ho & H. Wong-Toi (1997): *HYTECH: A Model Checker for Hybrid Systems*. In: *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, Springer-Verlag, London, UK, pp. 460–463.

[5] J.D. Hoffman (2001): *Numerical Methods for Engineers and Scientists*. Marcel Dekker, Inc, 2nd edition.

[6] M. Katelman, J. Meseguer & J. Hou (2008): *Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking*. In: G. Barthe & F. de Boer, editors: *Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, Lecture Notes in Computer Science 5051, Springer, pp. 150–169.

[7] N. Kumar, K. Sen, J. Meseguer & G. Agha (2003): *A Rewriting Based Model for Probabilistic Distributed Object Systems*. In: *Proc. Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, Lecture Notes in Computer Science 2884, Springer.

[8] E.A. Lee & H. Zheng (2006): *HyVisual: A Hybrid System Modeling Framework Based on Ptolemy II*. In: *IFAC Conference on Analysis and Design of Hybrid Systems*. Available at `http://chess.eecs.berkeley.edu/pubs/54.html`.

[9] E. Lien & P. C. Ölveczky (2009): *Formal Modeling and Analysis of an IETF Multicast Protocol*. In: *Proc. Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*, IEEE.

[10] J. Meseguer & R. Sharykin (2006): *Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems*. In: *Proc. Hybrid Systems: Computation and Control (HSCC'06)*, Lecture Notes in Computer Science 3927, Springer, pp. 460–475.

[11] P. C. Ölveczky & M. Caccamo (2006): *Formal Simulation and Analysis of the CASH Scheduling Algorithm in Real-Time Maude*. In: L. Baresi & R. Heckel, editors: *Fundamental Approaches to Software Engineering (FASE'06)*, Lecture Notes in Computer Science 3922, Springer, pp. 357–372.

[12] P. C. Ölveczky & J. Meseguer (2007): *Semantics and Pragmatics of Real-Time Maude*. Higher-Order and Symbolic Computation 20(1-2), pp. 161–196.

[13] P. C. Ölveczky, J. Meseguer & C. L. Talcott (2006): *Specification and Analysis of the AER/NCA Active Network Protocol Suite in Real-Time Maude*. Formal Methods in System Design 29(3), pp. 253–293.

[14] P. C. Ölveczky & S. Thorvaldsen (2009): *Formal Modeling, Performance Estimation, and Model Checking of Wireless Sensor Network Algorithms in Real-Time Maude*. Theoretical Computer Science 410(2-3), pp. 254–280.

[15] K. Sen, M. Viswanathan & G. Agha (2005): *On Statistical Model Checking of Stochastic Systems*. In: *Proc. Computer Aided Verification (CAV'05)*, Lecture Notes in Computer Science 3576, Springer, pp. 266–280.

[16] E. Villani, P. E. Miyagi & R. Valette (2007): *Modelling and Analysis of Hybrid Supervisory Systems: A Petri Net Approach*. Springer.

[17] P. E. Wellstead (1979): *Introduction to physical system modelling*. London : Academic Press.