Formal Modeling and Analysis of Hybrid Systems in Rewriting Logic using Higher-Order Numerical Methods and Discrete-Event Detection

Muhammad Fadlisyah
University of Oslo, Norway

Peter Csaba Ölveczky
University of Oslo, Norway

Erika Ábrahám
RWTH Aachen University, Germany

***Abstract*—In previous work, we proposed a methodology for the formal modeling, simulation, and model checking of *interacting hybrid systems* in the rewriting-logic-based *Real-Time Maude* tool. In that effort/flow-inspired methodology, both the physical components *and* their interactions are explicitly modeled, so that one only needs to describe the dynamics of *single* components and interactions, instead of having to perform the hard task of defining the continuous dynamics of the entire system.**

We previously used the Euler method to approximate the continuous dynamics defined by ordinary differential equations (ODEs). This paper explains (i) how we adapt, and then formalize in Real-Time Maude, the more precise Runge-Kutta numerical approximation methods to the effort/flow approach, in particular to coupled ODEs; and (ii) how we can use adaptive time increments to better approximate the time at which a discrete event must take place. Finally, we compare the results and the execution times for Real-Time Maude simulation and model checking with our previous approach on some thermal systems.

I. Introduction

Real-Time Maude [1], [2] is a high-performance tool that extends the rewriting-logic-based Maude system [3] to support the formal modeling, simulation, and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and has been useful for analyzing a wide range of advanced applications that are beyond the scope of timed automata, such as communication protocols [4], [5], wireless sensor network algorithms [6], [7], and scheduling algorithms requiring unbounded data structures [8].

This paper is part of an investigation into how Real-Time Maude can be used to formally model, simulate, and analyze *hybrid systems* with both discrete and continuous behaviors. One key challenge that we address, in contrast to many formal approaches to hybrid systems, is the *modularity* and *compositionality* of the specification of the system's continuous dynamics. We target complex hybrid systems in which multiple physical entities interact and may influence each other's continuous behavior. For example, a hot cup of coffee in a room interacts with the room through different kinds of heat transfer (such as thermal convection and conduction), leading to a decrease in the temperature of the coffee *and* to a slight increase in the temperature of the room. Defining "directly" the continuous dynamics of the *whole* system is very hard, as it involves combining the ordinary differential equations (ODEs) defining the dynamics of its components, and is not modular, since we must define anew the system's continuous dynamics for each new configuration of interacting physical components. To achieve modularity and compositionality, we define in [9] an object-oriented modeling methodology, based on the *effort/flow approach* [10], which allows us to specify the continuous dynamics of an entire system by defining it for the system's *single physical entities* (such as the cup of coffee and the room) and its *single physical interactions* (such as conduction and convection). This greatly simplifies the definition of the system's continuous dynamics and eliminates the need for redefining it when the configuration of physical components changes.

Our goal is to generate *executable models* of hybrid systems in Real-Time Maude. For the continuous behavior of physical systems, which is described by ODEs, in our previous work [9] the execution was based on the *Euler method* (see e.g. [11]) with *fixed time increments* to give approximate solutions to ordinary differential equations. That is, to approximate a system's behavior we consider a series of small *time steps*, each of them having the same duration, and approximate the behavior for each small time step.

One contribution of this paper is to describe the integration of more precise approximations based on the *Runge-Kutta method* (see, e.g., [12]). The original method was developed for non-coupled ODEs. However, our compositional effort/flow-based models of hybrid systems may involve *coupled ODEs*. We therefore first adapt the Runge-Kutta method of orders 2 and 4 to approximate the continuous behavior of such models, and describe the formalization of the adapted method in Real-Time Maude. Furthermore, we compare the results and execution times of both simulation and model checking between the Euler method and the Runge-Kutta-based methods on *thermal* systems with realistic parameters.

Another main challenge in hybrid systems analysis is to determine the *time point* at which a *discrete* event must take place. For example, if we heat water, its phase changes from liquid to evaporating when its temperature reaches $100°$. Using a *fixed time step size* in the approximation of the water's dynamics, we may miss the moment when the temperature is $100°$ by a significant amount. The second main contribution of this paper is the integration of a technique that approximates those time points at which a discrete event must take place more precisely by *dynamically* adjusting the time step size. We present the adaptive step size technique for the Euler and

the Runge-Kutta 2nd order approximation methods. We show the applicability of our techniques on three thermal systems with realistic parameters.

There are several simulation tools for hybrid systems, such as MATLAB/Simulink [13], HyVisual [14], and CHARON [15], that are based on numerical methods. In contrast to these tools, our approach supports, besides modeling and simulation, also the *formal analysis*, such as reachability analysis and linear temporal logic model checking, of hybrid systems. Of course, the results of such model checking must be seen in light of the approximation inaccuracies of the continuous behaviors. Our approach also differs from model checkers for hybrid systems, such as CheckMate [16], PHAVer [17], d/dt [18], and HYPERTECH [19], in that we do not use abstraction or over-approximation, but still support the modeling, reachability analysis, and LTL model checking of the full class of hybrid systems, describing the continuous dynamics by (possibly non-linear) ODEs. Whereas other formal tools use hybrid automata, chart or block models, or formulas for modeling, a main advantage of our approach is that it is based on the intuitive yet expressive *rewriting logic* formalism as the underlying modeling formalism, and therefore supports advanced object-oriented features as well as the definition of any computable data type. Finally, in [20] we show how the time step size can be adjusted for the Runge-Kutta method according to a user-defined error tolerance for the continuous behavior; that paper neither gives implementation details for the Runge-Kutta method nor deals with estimating the time of the next discrete event.

To summarize, the work presented in [9] and in this paper together provide:

1) a modeling framework for complex hybrid systems that is (i) intuitive and expressive, (ii) object-oriented, with support for advanced features such as inheritance and dynamic creation and deletion of objects, (iii) algebraic, and that (iv) makes it easy to specify the continuous dynamics in a simple and compositional way; and
2) a simulation, reachability analysis, and LTL model checking tool for such models based on fairly precise approximations of the system's continuous behavior.

The paper is structured as follows: Section II gives an overview of Real-Time Maude. Section III briefly explains our approach in [9] for modeling hybrid systems in rewriting logic that we extend. Section IV presents the adaptation of the Runge-Kutta method for our purposes, and Section V explains the discrete event detection approach. Section VI describes the Real-Time Maude implementation. The case studies are summarized in Section VII and some concluding remarks are given in Section VIII.

II. Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [3] theory with Σ a signature[1] and E a set of confluent and terminating *conditional equations*. (Σ, E) specifies the state space as an algebraic data type, and contains a specification of a sort `Time` modeling the time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) one-step transitions. The rules are applied *modulo* the equations E.[2]
- TR is a set of (possibly conditional) *tick rewrite rules* that model time elapse, written with syntax

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

  where τ is a term of sort `Time` denoting the *duration* of the rewrite.

The global states of a system are terms of the form $\{t\}$; the form of the tick rules then ensures that time advances uniformly in a system. The Real-Time Maude syntax is fairly intuitive (see [3]). For example, a function f with arguments of sorts $s_1 \ldots s_n$ and value of sort s is declared `op` f `:` $s_1 \ldots s_n$ `->` s. Equations are written `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. Variables are declared with the keywords `var` and `vars`.

A *class* declaration `class` C `|` att_1 `:` $s_1, \ldots,$ att_n `:` s_n . declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. A *subclass* inherits all attributes and rules of its superclasses. An *object* of class C 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 val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, a state is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and possibly *messages*. 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 in Real-Time Maude.

Real-Time Maude specifications are *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command `(trew` t `in time <=` τ `.)` simulates *one* fair behavior of the system *up to duration* τ, where t is the initial state and τ is a term of sort `Time`. The *search* command uses breadth-first search to analyze all possible behaviors of the system and checks whether a state matching a *pattern* can be reached from the initial state such that a given *condition* is satisfied. 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. Finally, the `find earliest` command determines the shortest time needed to reach a desired state.

III. Modeling Interacting Hybrid Systems with the Effort/Flow Approach

Our goal is to develop a methodology for the modeling and analysis of complex hybrid systems containing multiple

[1] i.e., declarations of *sorts*, *subsorts*, and *function symbols*

[2] E is a union $E' \cup A$, where A is a set of equational axioms (associativity, commutativity, identity, etc.). Deduction is performed *modulo* A. A term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

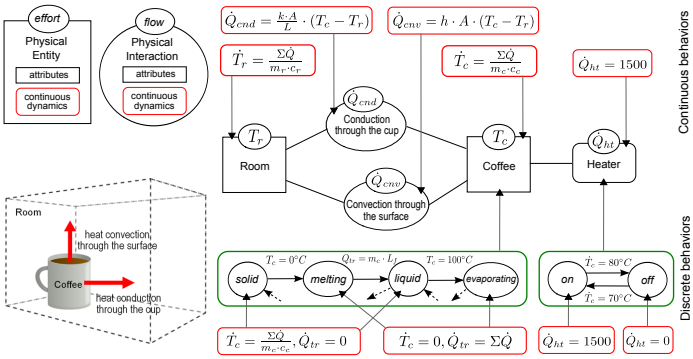

Fig. 1. Physical system components and their interactions in a simple thermal system.

physical components, where the continuous behavior of a component may depend on the continuous behavior of other components. In [9] we present our *modeling* approach. We adapt the *effort/flow* method [10], where also the *(physical) interactions* between *(physical) entities* are considered as first-class citizens. A physical system is therefore modeled as a network of physical entities and physical interactions. This makes the models *modular* in the sense that it is sufficient to define the continuous dynamics for each (class of) component(s) to define the dynamics of the entire system.

As shown in Fig. 1, a *physical entity* is described by a real-valued *effort*, a set of *attribute* values, and the entity's *continuous dynamics*. The attribute values describe discrete properties, e.g., the mass or the phase of a material, that are either constant or can be changed by discrete events only. The effort variable represents a dynamic physical quantity, such as temperature, that evolves over time as given by the continuous dynamics in the form of an ordinary differential equation (ODE). This dynamics may also depend on physical interactions connected to it.

A *physical interaction* between two physical entities is described by a real-valued *flow*, a set of *attribute* values, and a *continuous dynamics*. As for entities, the attributes represent discrete properties, whereas the flow value describes the dynamic interaction between two entities, e.g., the heat flow rate in a thermal interaction, whose evolution over time is specified by the continuous dynamics. Note that the explicit modeling of interactions avoids the direct coupling of the ODEs of physical entities.

Given a set of entities connected by some interactions, in [9] we use time discretization to approximate the entities' effort values after a given amount of time has elapsed: The time step is approximated by a sequence of small discrete time steps. For each small time step, we use the interactions' dynamics to determine the flow of energy, i.e., the current flow values of the interactions. These flow values together with the dynamics of the entities are used to compute the effort values of the physical entities after the small time step.

As we do not require that the ODEs involved in the above computation are linear, they are in general not solvable analytically. In [9] we use the Euler method for approximation: For a variable y with time derivative $y'(t) = f(t, y(t))$ and initial value y_n at time t_n, we approximate the value of y at time $t_{n+1} = t_n + h$, $h \in \mathbb{R}_{>0}$, by $y_{n+1} = y_n + hf(t_n, y_n)$ (see Fig. 2). An effort variable v_{effort} is described by a continuous dynamics $v'_{effort}(t) = f(t, \overrightarrow{v}_{attr}, \overrightarrow{v}_{flow}(t))$. Given the value $v_{effort,n}$ of v_{effort} at time t_n, first the values $\overrightarrow{v}_{flow,n}$ of the involved flow variables at time t_n are computed, and used to determine the derivative $d = f(t_n, \overrightarrow{v}_{attr}, \overrightarrow{v}_{flow,n})$ of the effort variable at time t_n. Finally, the approximation of the effort at $t_{n+1} = t_n + h$ is given by $v_{effort,n+1} = v_{effort,n} + hd$.

Besides the continuous behavior, hybrid systems also exhibit *discrete events*, corresponding to discrete changes such as, e.g., phase transitions or reactions to external stimuli. When a discrete event is enabled, time cannot advance until the event has been performed. Otherwise, the continuous behavior is performed as long as no discrete events are possible.

Fig. 1 shows a *thermal system* representing a cup of coffee in a room. In thermal systems, a physical entity is a *thermal entity*, whose effort variable (T) denotes the temperature of the entity and whose continuous dynamics defines the heat gained or lost by the entity as time evolves and its temperature changes. Likewise, a physical interaction is a *thermal interaction* whose flow variable ($\dot{Q}$) denotes the heat flow rate. Examples of thermal interactions are conduction, convection, and radiation. Their continuous dynamics are given by equations for the heat transfer rates.

IV. Higher Order Numerical Methods

There are two sources of errors in the above discrete-time approximation of a continuous behavior. *Round-off errors* originate from the limitations of computers in representing numbers. Discrete-time *approximation errors* originate from the fact that the approximations $y_1, y_2, ..., y_N$ of the values of y at time points $t_1, t_2, \ldots, t_N$ deviate from the exact values $y(t_1), y(t_2), ..., y(t_N)$. If we assume that an exact arithmetic is used, as in Maude, and thus there are no round-off errors, the deviation $|y(t_n) - y_n|$ is called the *global approximation error* at the nth step. This error stems from the global approximation error at the previous step and the approximation error of computing y_n based on y_{n-1} which we call the *local approximation error*.

The Euler method is an elementary technique to approximate solutions of first-order initial-value problems, and assumes that the integral curve can be approximated by a sequence of straight-line segments [21]. The error of the discrete-time approximation can be assessed by a comparison with a Taylor expansion [12]. Relative to the Taylor series, the Euler method has a local approximation error proportional to h^2 and a global approximation error proportional to h for a small time-step $0 < h << 1$. The main reason for this large error is that the approximation from t_n to t_{n+1} considers only the derivative at the beginning of the time interval. In general, if a numerical method has local approximation error h^p then its global approximation error is h^{p-1}. Numerical methods with a larger p yield higher accuracy at the cost of higher computational effort.

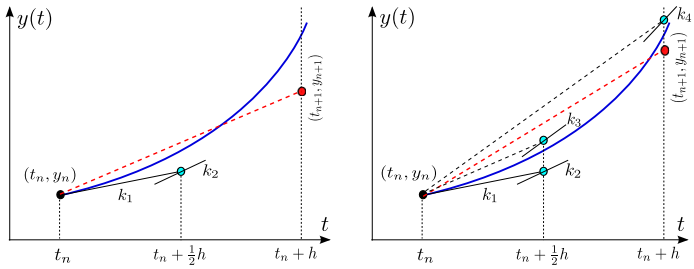

Fig. 2. The Runge-Kutta 2nd order and 4th order methods.

The *Runge-Kutta 2nd order method* (RK2), also known as the *midpoint method* (see Fig. 2), uses the derivative k_1 at the initial point (t_n, y_n) to find a *trial point* $p_1 = y_n + hk_1$ for y at time $t_n + h$, as done by the Euler method. Then it computes the derivative k_2 at the *trial midpoint* between (t_n, y_n) and $(t_n + h, p_1)$, and uses this *midpoint derivative* to determine the approximation y_{n+1} as follows:

$$\begin{array}{ll} k_1 = f(t_n, y_n) & k_2 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1) \\ y_{n+1} = y_n + hk_2 & \end{array} \quad (1)$$

The local approximation error is proportional to h^3, which makes this method 2nd order.

The *Runge-Kutta 4th order method* (RK4), illustrated in Fig. 2, is also known as the *classical Runge-Kutta* and computes four derivatives: one at the initial point, two at trial midpoints, and one at a trial endpoint. These derivatives are used to compute the approximation as follows:

$$\begin{array}{ll} k_1 = f(t_n, y_n) & k_2 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1) \\ k_3 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2) & k_4 = f(t_n + h, y_n + hk_3) \\ y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) & \end{array}$$

where k_1 is the slope at (t_n, y_n), k_2 at the midpoint between (t_n, y_n) and $(t_n + h, y_n + hk_1)$, k_3 at the midpoint between (t_n, y_n) and $(t_n + h, y_n + hk_2)$, and k_4 at the trial endpoint $(t_n + h, y_n + hk_3)$.

V. Discrete-Event-Detection-Based Adaptive Time Increments

Using fixed time increments in the approximations may lead to "long" delays in the execution of discrete events. Let y again be a continuous variable with time derivative $y'(t) = f(t, y(t))$, let h be the fixed time increment in the approximation, and let d be a discrete event that is enabled when the continuous variable y reaches the value δ, and so that y_n does not enable d whereas y_{n+1} enables d. In a fixed-increment approach, the discrete event d would be executed at time $t_n + h$, even though there might be "much" smaller time increments $h' < h$ such that the approximated value of y at time $t_n + h'$ enables d.

We want to dynamically compute "good" time increment values, by trying to detect whether some discrete event could be enabled between the current time t_n and $t_n + h$; if so, we compute a smaller step size $\tilde{h}_n$, so that d is enabled at time $t_n + \tilde{h}_n$, and advance time from t_n to $t_n + \tilde{h}_n$.

For the Euler method, $y_{n+1} = y_n + h \cdot f(t_n, y_n)$ is used to approximate the next value of y. Let δ be the switch point value of y which enables a discrete event d. The adaptive step size $\tilde{h}_n$ is then defined by $\tilde{h}_n = \frac{\delta - y_n}{f(t_n, y_n)}$.

The computations become more complex for the higher-order methods. For RK2, we redefine Equation 1 as a one-variable function $F(\tilde{h}) = \tilde{h} f\left(t_n + \frac{1}{2}\tilde{h},\ y_n + \frac{1}{2}\tilde{h} f(t_n, y_n)\right) + y_n - \delta$. We compute the value of $\tilde{h}$ by defining $F(\tilde{h}) = 0$, and then use Newton's method to compute the approximation of $\tilde{h}$ iteratively by $\tilde{h}_{m+1} = \tilde{h}_m - \frac{F(\tilde{h}_m)}{F'(\tilde{h}_m)}$. Using this method we need to define an initial guess $\tilde{h}_0$ to compute $\tilde{h}_1, \tilde{h}_2, ...$, which stops when $|\tilde{h}_{m+1} - \tilde{h}_m| < \varepsilon$, where ε is a tolerance value. We do not show here how to derive the function $F'(\tilde{h})$. However, at the end we get $F'(\tilde{h}) = f(t_n + \frac{1}{2}\tilde{h}_m,\ y_n + \frac{1}{2}\tilde{h}_m f(t_n, y_n)) + \frac{1}{2}\left[\frac{\partial f}{\partial t}(\tilde{h}_m) + f(t_n, y_n)\frac{\partial f}{\partial y}(\tilde{h}_m)\right]$.

VI. Integration in Real-Time Maude

In this section we describe how we have integrated the techniques for applying higher-order numerical methods and discrete event detection in our hybrid system modeling and execution framework introduced in [9]. The full Real-Time Maude specification, as well as longer technical reports describing this formalization in more detail, is available at http://folk.uio.no/mohamf/Hybrid.

In our object-oriented modeling framework, each physical entity and each physical interaction is modeled by an object. For example, a physical entity with the temperature as effort (a so-called thermal entity) can be modeled as an instance of the following (partially listed) class:

```
class ThermalEntity | temp : Rat, mass : PosRat, hCap : PosRat, ...
```

The effort variable `temp` represents the entity's temperature. The attributes `mass` and `hCap` denote the mass and the heat capacity of the entity, respectively. The water substance can then be defined as a subclass:

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

ops liquid solid gas melting evaporating
    condensing freezing : -> MatterState .
```

The attribute `phase` represents the phase of the water, which can be a main phase (solid, liquid, gas) or a phase transition (melting, freezing, evaporating, condensing). The change from a main phase to a phase transition occurs at a certain temperature. During a phase transition the temperature is constant. A change from a phase transition to a main phase happens when the heat `heatTrans` accumulated during the phase transition divided by the mass of the entity reaches the *latent heat*. The different kinds of physical interactions are also specified by classes. A system configuration contains objects modeling physical entities and interactions. For example, a water substance with temperature 20° can be represented by an object

```
< w : WaterEntity | temp : 20, phase : liquid, heatTrans : 0, ... >
```

There are two possible execution steps: discrete events and time steps. *Discrete events*, e.g., phase changes, are modeled by instantaneous rewrite rules. For example, the phase changes from solid to melting and from melting to liquid can be modeled by the following rules:

```
crl [solid-to-melting] :
    < WE : WaterEntity | phase : solid, temp : T  >
 => < WE : WaterEntity | phase : melting,  heatTrans : 0 >
    if T >= 0 .
crl [melting-to-liquid] :
    < WE : WaterEntity | phase : melting, heatTrans : QT, mass : M >
 => < WE : WaterEntity | phase : liquid  >
    if QT >= M * latentHeatFus .
```

For *time steps*, we can specify (1) which numerical approximation to use, and (2) whether fixed or adaptive step sizes should be used. The chosen numerical method and step size modes are managed by an instance of the `SysMan` class:

```
ops euler rk2 rk4 : -> NumMethod [ctor] .
class SysMan | numMethod : NumMethod, SSMode : SSModeType,
   SSDef : Rat, SSUsed : Rat, ...
```

The value of `numMethod` is one of the numerical methods `euler`, `rk2`, and `rk4`. The attribute `SSMode` for the step size mode has the value `static` for fixed step size and `dynamic` for an adaptive step size. `SSDef` is used to store the step size for the next time step as determined by the step size mode, and `SSUsed` the step size that was used in the last time step.

We have also integrated the adaptive-step-size technique described in [20] which adjusts the step size according to a user-given error tolerance in the approximation of the continuous behaviors; indeed, that feature is used in case study 3 below, but will not be further explained in this paper.

Time steps may be executed only as long as some invariants are satisfied. E.g., the temperature of ice may increase only as long as the temperature is below $0°$, since at that point a phase change must happen. Such invariants are specified by a function `timeCanAdvance`. For the phases solid and melting the function is defined as follows:

```
eq timeCanAdvance(
   < WE : WaterEntity | phase : solid, temp : T > )
   = T < 0 .
eq timeCanAdvance(
   < WE : WaterEntity | phase : melting, mass : M, heatTrans : QT>)
   = QT < M * latentHeatFus .
```

The *tick rules* modeling time elapse use some functions explained below.

The `computeEffort` function computes the approximation for a small time step. The function `stepSize` returns a pair of values, the first one being the size for the current time step and the second one the size for the next time step, without considering discrete event detection. Finally, `adaptiveStepSizeDE` adapts that step size for the detection of discrete events.

In the following tick rule, time advances using a fixed step size:

```
crl [static] :
    {< SM : SysMan | SSDef : D, SSMode : static > REST}
 => {computeEffort (< SM : SysMan | SSUsed : D > REST) } in time D
    if timeCanAdvance(< SM : SysMan | > REST) .
```

The following tick rule, adaptive step size is enabled.

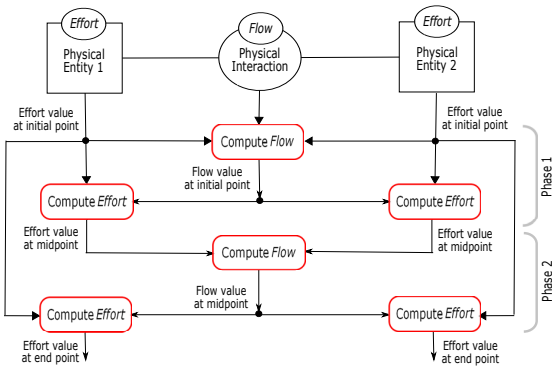

Fig. 3. The implementation of the Runge-Kutta 2nd order method in Effort-Flow framework.

The function `adaptiveStepSizeDE` is used to define the discrete-events-detection-based adaptive step size:

```
crl [dynamic] :
    {< SM : SysMan | SSDef : D, SSMode : dynamic > REST}
 => {computeEffort (< SM : SysMan | SSUsed : U > REST) } in time U
    if timeCanAdvance(< SM : SysMan | > REST)
       /\ U := adaptiveStepSizeDE(
                  < SM : SysMan | SSUsed : D > REST) .
```

A. Higher-Order Numerical Methods

To apply the RK2 method for the approximation of the effort values, we compute, for each time step, the slopes k_1 and k_2 for the effort variables. However, the continuous dynamics of thermal entities may refer to the flows of connected thermal interactions. In order to compute the slopes for the efforts, we need the values of the flow variables at the initial point and at the midpoint. We must therefore adapt the RK2 method to work interleaved with the flow computation.

Fig. 3 illustrates how the RK2 method is applied in our effort-flow framework. Assume a thermal interaction and two connected thermal entities with given initial effort variable values at time t_n. We first compute the initial flow variable value at time t_n using the initial effort variable values. Using the result, we compute the slope k_1 at the initial point for both involved effort variables. These slopes allow us to compute the two midpoints. In the second phase we determine the flow variable value at time $t_n + \frac{1}{2}h$ using the midpoint values of the effort variables. The resulting flow variable value allows us to compute the slopes k_2 of the effort variables at the midpoints. Finally, those midpoint slopes are used to compute the approximation of the effort variable values at the endpoint at t_{n+1}.

To implement the RK2 method, we define several functions for computing the flow and effort variables. We do not show the details of the functions here since they have similar patterns with the ones of our previous works in [9], [20]. To execute the continuous behaviors using the RK2 method the following function used in the tick rule:

```
op computeEffort : Configuration  -> Configuration .
eq computeEffort(
   < SM : SysMan | numMethod : rk2 > REST) =
   computeEffort-FI(computeFlow-MP(
     computeEffort-MP(computeFlow-IP(< SM : SysMan | > REST)))) .
```

As explained in Section IV, to compute the approximated value of an effort variable y using the RK4 method, we need to compute at each time step the slopes k_1, k_2, k_3, and k_4. For that we need, besides the initial value, three values of y: two at the midpoints and one at the trial endpoint. Furthermore, we need the approximated value at the end of the interval. Again, before we can compute those four y values, we need to compute the related flow variable values. This method has been formalized in a similar way as the midpoint method. To execute the continuous behaviors using the RK4 method the following function used in the tick rule:

```
eq computeEffort(
   < SM : SysMan | numMethod : rk4 > REST) =
   computeEffort-FI(computeFlow-P3(
     computeEffort-P3(computeFlow-P2(
      computeEffort-P2(computeFlow-P1(
       computeEffort-P1(computeFlow-IP(
        < SM : SysMan | > REST)))) .
```

The implementation of the different `computeEffort` functions above involve the computation of slopes, based on the system's dynamics. The functions `Tdot` and `HTdot` are used to specify the continuous dynamics of the temperature and the heat transition variables of the water entity, respectively. Their definitions distinguish between main and transition phases of the material, and on the computation of the numerical method in that they are invoked. The two equations below define these functions for the main phases and for the initial points (IP) in the numerical methods:

```
ops Tdot-IP HTdot-IP : Oid Configuration -> Rat .
ceq Tdot-IP(TE, < TE : ThermalEntity | hCap : C, mass : M,
           phase : P > REST)
   = TdotFrm(sumFlows-IP(TE, REST), M, C)
   if P == solid or P == liquid or P == gas .
ceq HTdot-IP(TE, < TE : ThermalEntity | phase : P > REST ) = 0
   if P == solid or P == liquid or P == gas .
```

The above functions represent the slope at the initial point in the form of $f(t_n, y_n)$. For the other slopes we have similar definitions. E.g., `Tdot-MP` and `HTdot-MP` specify the dynamics for the midpoints in RK2 in the form $f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hf(t_n, y_n))$.

B. Discrete Event Detection

To detect discrete events, we make the time step size adaptive as follows. Before time advances with some step size value we check whether there are discrete events that would get enabled during the time step. If yes, we decrease the time step size such that we get near to the first point in time at which a discrete event gets enabled. If no, time will advance with the predefined step size.

The function `adaptiveStepSizeDE`, which is used in the tick rules, computes the adaptation of the step size for the detection of discrete events.

```
op adaptiveStepSizeDE : Configuration -> Rat .

eq adaptiveStepSizeDE(< SM : SysMan | SSUsed : U > REST) =
   if DEWillOccur(computeEffort(< SM : SysMan | > REST))
   then stepSizeDE(computeFlow-IP(< SM : SysMan | > REST))
   else U fi .
```

The function `DEWillOccur` checks discrete event occurrences recursively in each entity. We show the cases for the solid and melting phases:

```
op DEWillOccur : Configuration -> Bool .
eq DEWillOccur(< WE : WaterEntity | phase : solid, temp : T >) =
   T >= 0 .
eq DEWillOccur(< WE : WaterEntity | phase : melting, mass: M,
                                    heatTrans : H >) =
   H >= M * latentHeatFus .
```

If any discrete event would get enabled during a time step, we decrease the time step size. We do so by approximating for each such discrete event the duration after that it gets enabled, and take the minimum[3]. This way we assure that we do not miss any discrete event, i.e., that the reduction happens for the discrete event that gets enabled first. The function `stepSizeDE` performs this computation depending on the numerical method used:

```
eq stepSizeDE(
   < WE : WaterEntity | >
   < SM : SysMan | numMethod : euler, SSUsed : U > REST)
   = min(min(U, stepSizeEuler(< WE : WaterEntity | > REST)),
         stepSizeDE(< SM : SysMan | > REST)) .

eq stepSizeDE(
   < WE : WaterEntity | >
   < SM : SysMan | numMethod : rk2, SSUsed : U > REST)
   = min(min(U, stepSizeRK2(< WE : WaterEntity | >
                            < SM : SysMan | > REST)),
         stepSizeDE(< SM : SysMan | > REST)) .

eq stepSizeDE(< SM : SysMan | SSUsed : U > REST) = U [owise] .
```

The function `stepSizeEuler` computes the step size based on the Euler method. The computation, represented by the function `adaptEuler`, is based on the equation explained in Section V. The equation below defines the case for the solid phase of a water entity. Similar cases cover the other phases.

```
op stepSizeEuler : Configuration -> Rat .
eq stepSizeEuler(< WE : WaterEntity | phase : solid, temp : T > REST)
  = adaptEuler(T, 0, Tdot-IP(WE, < WE : WaterEntity | > REST)) .

op adaptEuler : Rat Rat Rat -> Rat .
eq adaptEuler(Y1, Y2, F) = (Y2 - Y1) / F .
```

The function `stepSizeRK2` computes the step size based on the RK2 method. It uses the function `adaptNewton`, the application of Newton's method (explained in Section V). Below is the equation defining the function for the water entity in the solid phase. The other cases are analogous.

```
op stepSizeRK2 : Configuration -> Rat .

ceq stepSizeRK2(
    < WE : WaterEntity | phase : solid, temp : T >
    < SM : SysMan | SSUsed : U, diffTol : DIFFTOL > REST)
    =   if abs(U - HM1) <= DIFFTOL
        then HM1 else
        stepSizeRK2(computeFlow(< WE : WaterEntity | >
           < SM: SysMan | SSUsed : HM1 > REST)) fi
   if HM1 := adaptNewton(U, T, 0,
                Tdot-P1(WE, < WE : WaterEntity | > REST),
                Tdot-IP(WE, < WE : WaterEntity | > REST),
                dfdt(WE, < WE : WaterEntity | > REST),
                dfdy(WE, < WE : WaterEntity | > REST)) .

op adaptNewton : Rat Rat Rat Rat Rat Rat Rat -> Rat .
eq adaptNewton(U, Y1, Y2, F1, F2, DFDT, DFDY)
   = U - (U * F1 + (Y2 - Y1)) / (F1 + 1/2 * (DFDT + F2 * DFDY)) .
```

Because of space limitation, we do not show the im-

[3]Thereby we ignore negative values, for example when ice is getting colder then we would approximate the duration until melting by a negative number.

plementation of `dfdt` and `dfdy` which correspond to the computation of $\frac{\partial f}{\partial t}$ and $\frac{\partial f}{\partial y}$ in Section V.

VII. Case Studies

In [9], we use the Euler-based method with fixed sized time increments to model and analyze thermal systems in Real-Time Maude. This section investigates how using higher-order numerical methods and adaptive step size technique affects the accuracy and performance of the simulation and model checking of those systems. We start with a cup of hot coffee in a room. We then start with iced coffee and add a heater that provides constant heat to the coffee (to analyze melting and evaporating). Finally, we add a controller which should turn the heater on or off in order to keep the temperature of the coffee in a desired interval. The experiments are performed on an Intel(R) Pentium(R) 4 CPU 3.00GHz with 3GB of RAM.

A. Case Study 1: A Cup of Coffee in a Room

We first model a cup of 70° hot coffee in a 20° room, as shown in Fig. 1, with *conduction* and *convection* as thermal interactions, and with realistic physical parameters. The initial state consists of a `SysMan` object managing the numerical computation, thermal entity objects `coffee` and `room`, and two thermal interaction objects that model the heat flow:

```
op cs1 : -> GlobalSystem .
eq cs1 =
{< coffee : WaterEntity | temperature : 70, mode : default,
                          heatCap : coffeeHC, mass : coffeeMass,
                          heatTrans : 0,  phase : liquid >
 < room : ThermalEntity | temperature : 20, mode : default,
                          mass : roomMass, heatCap : roomHC >
 < cond : Conduction | entity1 : coffee, entity2 : room, hfr : 0,
                       thermCond : k, area : condArea,
                       thickness : cupThick >
 < conv : Convection | entity1 : coffee, entity2 : room, hfr : 0,
                       convCoeff : h, area : convArea >
 < sysMan : SysMan   | numMethod : rk4, stepSize : STEP-SIZE >} .
```

The behavior of the system until time 1000 can be simulated using the following timed rewriting command:

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

Fig. 4 (left) shows the simulation results using Euler, RK2, and RK4 methods. Since the ODE system defining the dynamics of *this* system *is* analytically solvable, we have solved it, and have computed the exact coffee temperature at each time point. Fig. 4 (left, zoomed) shows the accuracy of the different methods compared to the exact solutions. Fig. 4 (right) shows the relative approximation error percentage of different methods during in each time point of simulation compared to the exact solutions.[4]

The following table shows the final temperature values of the coffee (T_c) and the room (T_r) at time 1000, as well as the the average of relative approximation error and computation times, for the three numerical methods with time step $h = 1$:

[4]The relative approximation error is $\frac{|\text{val}_{\text{exact}} - \text{val}_{\text{approx}}|}{|\text{val}_{\text{exact}}|}$. Its percentage error is 100 times the relative error.

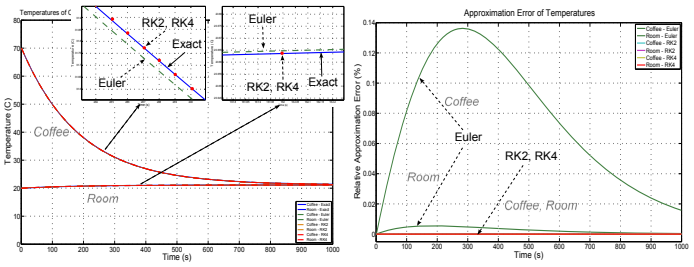

Fig. 4. The simulation results of case study 1.

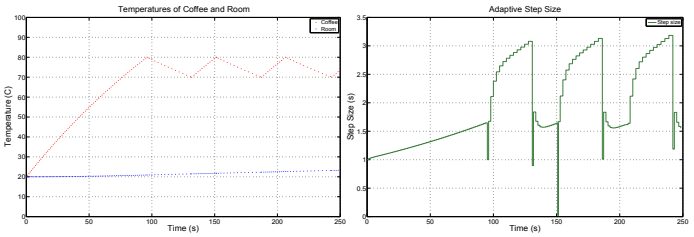

Fig. 5. The simulation results of case study 3.

		Euler	RK2	RK4
Value at time 1000	T_c	21.3849029833	21.3882633204	21.3882573106
	T_r	21.1459272868	21.1458480788	21.1458482205
Average relative error	T_c	0.0767	1.3684e-04	1.4615e-10
	T_r	0.0027	4.7831e-06	0
CPU time (s)		49	233	1950

B. Case Study 2: Heating the Coffee

To illustrate hybrid behaviors, we add a heater providing a constant heat flow to the cup of coffee with starting temperature $-10°C$. We use the *find earliest* command to find out when our coffee starts melting (and similarly for evaporation):

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

The following table compares the results and execution time of the above command for fixed and adaptive step size techniques (with default step size 1). For RK2 with adaptive step size we compare the results using different tolerances (10^{-3} and 10^{-6}), which relate to the use of Newton's method when computing the step size:

Step size	Num. method	Melting starts			Evaporation starts		
		temperature at discrete switch	time point	CPU time (s)	temperature at discrete switch	time point	CPU time (s)
Fixed	Euler	0.1326328017	11	0.28	100.2297897362	269	5.1
	RK2	0.1068407713	11	0.31	100.0573413049	268	19
	RK4	0.1068865758	11	0.66	100.0590566081	269	104
Adapt.	Euler	0.0000000000	10.8521692720	0.47	100.0000000000	267.4762327594	18
	RK2 (tol 10^{-3})	0.0000022124	10.8803479312	0.58	100.0000007376	267.7335530093	650
	RK2 (tol 10^{-6})	0.0000000000	10.8803454543	0.64	100.0000000000	267.7335504855	1116

C. Case Study 3: Controlling the Heater

Finally, we add a controller to keep the temperature of the coffee between 70° and 80° by turning the heater on and off. Figure 5 shows the simulation results using the Euler method with adaptive step size, with default step size 1. The diagram on the right shows the step size, which gets smaller when a discrete change (turning the heater on or off) happens.

Simulation cannot analyze whether a complex property holds in a system. Many complex system requirements can be

expressed as *linear temporal logic* (LTL) formulas, and Real-Time Maude's LTL model checker can check whether each possible behavior, up to a certain duration, of a system satisfies a given LTL formula. The results of such model checking must of course be understood in light of the approximation of the continuous dynamics. We use LTL model checking to analyze the stability property that once the temperature of the coffee has reached $70°$, it will remain between $70°$ and $80°$. Because of approximations, we have defined an atomic proposition `temp-ok` to hold in all states where the coffee temperature is between $69.9°$ and $80.1°$. The following time-bounded model checking command then checks whether our property holds for all behaviors up to time 500:

```
Maude> (mc cs3 |=t [] (temp-ok -> [] temp-ok) in time <= 500 .)
```

For *fixed* step size 1, this command returns `true` for the Euler, RK2, and RK4 methods after executing for, respectively, 10, 38, and 281 seconds. The commands fail to terminate within a couple of hours when using *adaptive* step sizes.

VIII. Concluding Remarks

In this paper we describe (i) how the 2nd and 4th order Runge-Kutta methods can be adapted to effort/flow-based compositional modeling of *interacting* hybrid systems, and how the adapted methods are formalized in Real-Time Maude; and (ii) how the time increment can be dynamically adjusted to better approximate the time when a discrete transition should be performed. We compare the precision and execution times on three thermal systems, and show that the better accuracy of the Runge-Kutta methods comes at a cost in terms of computational effort. Adaptive step sizes also come at a cost, and while this cost is acceptable for simulation, it might make LTL model checking of large systems unfeasible. It is also worth noting that the Euler method with adaptive step sizes can sometimes compute the precise point in time when the (approximated) value of a "continuous variable" reaches a certain value.

Making these methods, and the modeling framework, available within the rewriting logic tool Real-Time Maude should make Real-Time Maude a suitable candidate for the object-based formal modeling, simulation, and model checking of advanced hybrid systems due to the tool's expressiveness, support for concurrent objects, user-definable data types, different communication models, etc.; this of course has to be validated on more advanced hybrid systems.

Acknowledgments

We gratefully acknowledge financial support for this work by The Research Council of Norway through the Rhytm project and by The Research Council of Norway and the German Academic Exchange Service through the DAAD ppp project HySmart.

References

[1] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.

[2] ——, "The Real-Time Maude tool," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 332–336.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, LNCS. Springer, 2007, vol. 4350.

[4] P. C. Ölveczky, J. Meseguer, and C. L. Talcott, "Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude," *Formal Methods in System Design*, vol. 29, no. 3, 2006.

[5] E. Lien and P. C. Ölveczky, "Formal modeling and analysis of an IETF multicast protocol," in *Proc. SEFM '09*. IEEE, 2009.

[6] M. Katelman, J. Meseguer, and J. Hou, "Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking," in *Proc. FMOODS'08*, LNCS, vol. 5051. Springer, 2008.

[7] P. C. Ölveczky and S. Thorvaldsen, "Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude," *Theoretical Computer Science*, vol. 410, no. 2-3, 2009.

[8] P. C. Ölveczky and M. Caccamo, "Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude," in *Proc. FASE '06*, LNCS, vol. 3922. Springer, 2006.

[9] M. Fadlisyah, E. Ábrahám, D. Lepri, and P. C. Ölveczky, "A rewriting-logic-based technique for modeling thermal systems," in *Proc. RTRTS'10*, Electronic Proceedings in Theoretical Computer Science, vol. 36, 2010.

[10] P. E. Wellstead, *Introduction to physical system modelling*. Academic Press, 1979.

[11] J. Hoffman, *Numerical Methods for Engineers and Scientists*. Marcel Dekker, Inc, 2001.

[12] W. Cheney and D. Kincaid, *Numerical Mathematics and Computing*. Brooks & Cole Publishing Co., 1994.

[13] "*Simulink home page*," http://www.mathworks.com/products/simulink.

[14] E. Lee and H. Zheng, "HyVisual: A hybrid system modeling framework based on Ptolemy II," in *IFAC Conference on Analysis and Design of Hybrid Systems*, 2006.

[15] J. Esposito, V. Kumar, and G. Pappas, "Accurate event detection for simulating hybrid systems," in *Proc. of HSCC'01*, LNCS, vol. 2034. Springer, 2001.

[16] E. Clarke, A. Fehnker, Z. Han, B. Krogh, O. Stursberg, and M. Theobald, "Verification of hybrid systems based on counterexample-guided abstraction refinement," in *Proc. TACAS'03*, LNCS, vol. 2619. Springer, 2003.

[17] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Proc. HSCC'05*, LNCS, vol. 3414. Springer, 2005.

[18] T. Dang, "Verification and synthesis of hybrid systems," Ph.D. dissertation, INPG, 2000.

[19] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-toi, "Beyond HYTECH: Hybrid systems analysis using interval numerical methods," in *Proc. of HSCC'00*, LNCS, vol. 1790. Springer, 2000.

[20] M. Fadlisyah, E. Ábrahám, and P. C. Ölveczky, "Adaptive-step-size numerical methods in rewriting-logic-based formal analysis of interacting hybrid systems," in *Proc. TTSS'10*, 2010, to appear in *ENTCS*.

[21] B. Rice and J. Strange, *Ordinary Differential Equations with Applications*. Brooks/Cole Publishing Company, 1989.