A Tool-Supported Approach for Modeling and Verifying Hybrid Systems Using EVENT-B and the Differential Equation Solver SAGEMATH

Meryem Afendi[1], Amel Mammar[2] and Régine Laleau[1]
[1]Univ. Paris Est Creteil LACL, F-94010 Creteil, France
[2]SAMOVAR, Institut Polytechnique de Paris Télécom SudParis, Evry, France

Keywords: Cyber-Physical System, EVENT-B, Refinement, Correctness Proof, Ordinary Differential Equation, Differential Equation Solver.

Abstract: The common mathematical model for cyber-physical systems is that of hybrid systems that enable combining both discrete and continuous behaviors represented by differential equations. In this paper, we introduce a formal approach, using EVENT-B and its refinement strategy, for specifying and verifying cyber-physical systems whose behavior is described by ordinary differential equations. To deal with the resolution of ordinary differential equations in Event-B, the approach is based on interfacing the differential equation solver SAGEMATH (System for Algebra and Geometry Experimentation) with the RODIN tool, a platform for EVENT-B projects development. For this purpose, we modeled and implemented the interface to the solver in EVENT-B using a RODIN plugin. This enables to reason on the EVENT-B specification and prove safety properties. The proposed approach was successfully applied on a frequently used cyber-physical system case studies.

1 INTRODUCTION

Hybrid systems involve explicitly and simultaneously continuous and discrete behaviors. In general, the state of a hybrid system is defined by the values of the continuous variables and the discrete mode of the controller. The formal modeling, verification and overall design of hybrid systems give rise to serious challenges. The development of techniques and tools to effectively design and verify hybrid systems has drawn the attention of many researchers. These approaches can be grouped into two categories: *model-checking-based* approaches and *proof-based* approaches.

The work presented in this paper is achieved in the context of the DISCONT project (DISCONT ANR Project, 2017) that aims at developing formal approaches for building correct cyber-physical systems (CPSs). The physical parts behavior of such systems is often described by ordinary differential equations (ODEs) (Perko, 2013) that involve an unknown function depending on a single variable. Following this objective, this paper introduces a proof-based approach to model CPSs using EVENT-B (Abrial, 2010). Our development is inspired by the approach introduced in (Kopetz, 1991) that consists in specifying an abstract model, *EventTriggered* model, where the controller interrupts the physical part when a particular event occurs, and then introducing a more concrete model, *TimeTriggered* model, where the controller interrupts periodically the physical part.

The *EventTriggered* model describes an ideal behavior where the time is continuous and the sensors have continuous access to continuous measurements. *TimeTriggered* model represents a more realistic behavior where the sensors take periodic measurements. In our approach, the EVENT-B refinement mechanism allows to formalize and prove the refinement link between *EventTriggered* and *TimeTriggered* models. The EVENT-B method is designed for modeling discrete systems, it does not support the resolution of ODEs. To deal with this limitation, we interface the RODIN tool, a platform for the EVENT-B modeling language, with a differential equation solver, SAGEMATH (System for Algebra and Geometry Experimentation) (Zimmermann et al.,

Afendi, M., Mammar, A. and Laleau, R.
A Tool-Supported Approach for Modeling and Verifying Hybrid Systems using EVENT-B and the Differential Equation Solver SAGEMATH.
DOI: 10.5220/0012080900003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 71-83
ISBN: 978-989-758-665-1; ISSN: 2184-2833
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

2018) in our case. This is achieved by implementing a plugin to RODIN that enables to call SAGEMATH.

The paper is organized as follows. The next section briefly describes the formal EVENT-B method and the ODE solvers. Section 3 is dedicated to a state of the art on proof-based approaches to deal with continuous aspects of CPSs and the approaches that inter- face theorem provers with computer algebra systems. Sections 4 and 6 introduce our approach for combining a differential equation solver with EVENT-B and illustrate it on a frequently used CPS case study. Section 5 describes the tool developed to support the approach. Finally, Section 7 concludes and presents some future work.

2 BACKGROUND

2.1 EVENT-B

EVENT-B is a formal method introduced by J. Raymond Abrial to describe discrete systems using events (Abrial, 2010). It is based on set theory and predicate logic. An EVENT-B model is composed of machines and contexts. EVENT-B context represents the static part and includes constants C, abstract and enumerated sets S, and their properties specified as axioms A. A context may be extended by adding new elements. EVENT-B machine defines variables V which describe the machine state, invariants Inv which define the properties that must be satisfied whatever the state of the system. The behavior of the system is described by a set of events of the form (**ANY** X **WHERE** G **THEN** S **END**), which update the state variables using the substitution S when its guard or condition G are satisfied. A machine may see contexts to have access to their contents.

An EVENT-B model gives rise to a set of Proof Obligations (POs) that aim at verifying its correctness. Basically, we have to verify that:

- the initialisation establishes the invariant, that means that the invariant is correct after the initialisation: [*INITIALISATION*] *Inv*, where the expression [S]*Inv* denotes the substitution S applied to the formula *Inv*; it denotes the weakest constraint on the before state such that the execution of S leads to an after-state satisfying *Inv*.
- for each event of the form (**ANY** X **WHERE** G **THEN** S **END**), we have to prove that the invariant Inv is preserved by the execution of the event: $\forall(S,C,V,X).(A \wedge G \wedge Inv \Rightarrow [S]Inv)$

EVENT-B is supported by the open-source and free RODIN platform which is an Eclipse-based IDE. New features can be added to RODIN as plug-ins. For example, the theory plug-in (Butler and Maamria, 2010) is a RODIN extension that allows one to define new data types like *REAL*, new operators, etc.

The key feature of EVENT-B to master system complexity consists in using abstract modeling to represent the abstract behavior of a given system and the refinement to introduce details by establishing the compliance between the abstract and the concrete models. The EVENT-B refinement technique preserves all the properties introduced and proved in the abstract model. To demonstrate that a concrete model *Conc* refines an abstract one *Abs*, it must be proved that all behaviors of *Conc* are included in those of *Abs*. For each event (**ANY** X **WHERE** G **THEN** S **END**) refined by the event (**ANY** X_r **WHERE** G_r **THEN** S_r **END**), we have to establish the following two proof obligations:

- *Guard refinement:* the guard of the refined event should be stronger than the guard of the abstract one:
  $\forall(S,C,S_r,C_r,V,V_r,X,X_r).$
  $(A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$
- *Simulation:* the effect of the refined substitution should be stronger than the effect of the abstract one:
  $\forall(S,C,S_r,C_r,V,V_r,X,X_r).$
  $(A \wedge A_r \wedge Inv \wedge Inv_r \wedge [S_r]Inv_r \Rightarrow [S]Inv)$

2.2 Differential Equation Solvers

In this paper, we focus on ordinary differential equations (ODEs). An ODE involves an unknown function that depends on a single variable. The most common form of the ODEs that describes the evolution of hybrid systems is: $a_n(t)y^n(t) + ... + a_2(t)y''(t) + a_1(t)y'(t) + a_0(t)y(t) = b(t)$, where the dependent variable is y and the independent variable is t. There are many computer algebra systems for solving ODEs, such as SAGEMATH (Zimmermann et al., 2018) and Mathematica (Wolfram, 2003).

To find symbolic solutions for a given ODE, SAGEMATH mainly offers two functions: *desolve()* and *desolve rk4*. *desolve()* that takes as parameters: (*de,dvar,ivar,ics*), where: *de* represents the differential equation, *dvar* represents the unknown function, *ivar* represents the independent variable and *ics* is an optional argument denoting the initial conditions. *desolve rk4*(de,dvar,ivar,ics) is very similar to *desolve*, it returns an approximate solution of the ODE in the form of tuple (*t, y*).

3 STATE OF THE ART

Today, rigorous development methodologies based on mathematical and logical foundations are mature enough to support the development of hybrid systems. Formal approaches for modeling and verifying hybrid systems can be divided into two categories: model checking-based approaches and proof-based approaches. Model-checking-based approaches use hybrid automata to model hybrid systems and algorithmic analysis methods to prove their safety properties (Henzinger et al., 1997; Frehse et al., 2011). On the other hand, proof-based approaches use deductive verification to model and prove the properties of hybrid systems. In this paper, we are interested in proof-based approaches that can handle differential equations in hybrid systems modeling.

3.1 EVENT-B Modeling of Hybrid Systems

Hybrid EVENT-B, introduced in (Banach et al., 2015), extends EVENT-B to model both discrete and physical parts of CPSs. It defines two kinds of events: *mode events* that represent the traditional discrete EVENT-B events and the *pliant events* that describe the continuous behavior of the physical part by specifying the corresponding differential equation using piecewise absolutely continuous functions. On the domain of reals (**IR**), such functions are continuous over intervals, with possibly different values on the edges. The correctness of Hybrid EVENT-B models is ensured using a set of customised proof obligations patterns, defined in a way similar to classical EVENT-B. The method was successfully applied on many concrete examples and has been a source of inspiration for several approaches (Su et al., 2014; Butler et al., 2016; Dupont et al., 2018) including ours. Nevertheless, its major limitation is that it is not supported by any automatic tool. Therefore, the proof obligations must be generated and discharged manually, which makes it difficult to apply on critical systems since the proof phase is error-prone. Moreover, Hybrid EVENT-B does not provide any mechanism for solving differential equations in EVENT-B, which is a strength of our approach.

The approach, introduced by G. Dupont et al. in (Dupont et al., 2018), proposes the use of the plug-in Theory of EVENT-B to handle continuous aspects of CPSs. The main idea of this approach is to describe the continuous evolution of time t and the generic continuous measurements, represented by the variable *plantV*. *plantV* is indexed by TIME ($plantV \in TIME^1 \rightarrow S$) where S is a constant defined in the associated context to be equal to: $\mathbf{IR}^n$ with n is the number of the system continuous variables. The progression of time t is modeled by an event named *Progress*. It states that the new value of time t_1 will become greater than its previous value, ($t < t_1$).

This approach introduces a theory named *DiffEq* that provides several abstract operators to model ODEs in EVENT-B. To prove the safety properties of hybrid systems, it defines properties on these ODEs using operators like the operator *Solvable* which ensures that the ODEs are solvable. Nevertheless, no mechanism is provided to solve the concrete ODEs. Therefore, this approach remains quite abstract regarding the resolution of differential equations. In this paper, we extend this abstract model and use some operators of the theory *DiffEq* to solve ordinary differential equations using EVENT-B and SAGEMATH. Solving ODEs allows proving the safety properties of hybrid systems in a *Time-Triggered* system which is the most concrete model of CPSs that we could achieve. Moreover, these solutions will serve in developing an automation tool to simulate the continuous behaviors of hybrid systems in EVENT-B.

3.2 Differential Logic (dL)

In (Platzer, 2008), Platzer introduces a proof-based approach to model and verify hybrid systems. It is based on a first-order differential logic in the domain of reals (**IR**), named dynamic differential logic dL, and its associated proof calculus. Both are supported by the theorem prover KeYmaera (Platzer and Quesel, 2008) and its successor KeYmaera X (Fulton et al., 2015). The advantage of this approach is its ability to model and verify the evolution of continuous measurements, represented by differential equations, using dL formulas. Hybrid systems are described in dL using hybrid programs, whose hybrid dynamics arise from combining a specific set of ordered discrete programming constructs with continuous ODEs. Most hybrid programs are modeled by the formula, $(ctrl;\ plant)^1$, where ctrl denotes the execution of the controller (discrete evolution), followed by the physical part plant (continuous evolution), with a non-deterministic repetition. dRL (differential Refinement Logic) (Loos and Platzer, 2016) extends dL by introducing the notion of refinement on hybrid systems. Interestingly,

[1] Defined on $\mathbf{IR}^+$

Event- and *Time- Triggered* systems are specified with d$\mathcal{RL}$ and proof obligations are defined to check that *TimeTriggered* is a refinement of *EventTriggered*. d$\mathcal{RL}$ is not supported by any prover, it cannot be used for critical systems since the proof obligations are manually generated and discharged. To overcome this limit, we proposes a new correct-by construction approach to prove this refinement, based on EVENT-B to take advantage of its well-defined refinement process and its support tools. Unlike d$\mathcal{RL}$, developing hybrid systems with EVENT-B enables to deal with the complexity of the system by incrementally introducing the properties. Moreover, EVENT-B enables to have a good view on the proof activity and its different steps that helps us to have a better understanding of the system.

4 A GENERIC FORMAL APPROACH FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS IN EVENT-B

Besides the proposed formal approaches, Kopetz (Kopetz, 1991) introduces an approach that we have found interesting because it considers a CPS at different levels of abstraction that allows to deal with the complexity of such systems. The proposed approach consists in specifying an abstract model, *Event-Triggered* model, in which the controller interrupts the physical part when certain events occur. Then defining a more concrete model, *Time-Triggered* model, in which the controller interrupts periodically the physical part.

This section describes our contribution in using the EVENT-B method and SAGEMATH for modeling and verifying hybrid systems. Our proposal consists in modeling the (*ctrl; plant*)[2] idiom of *Event* and *Time-triggered* models with the event-based paradigm of EVENT-B. The link between these models are expressed as a refinement relationship. Our approach reuses the abstract model and the theories introduced in (Dupont et al., 2018), such as *DiffEq*, to handle continuous aspects of hybrid systems. This allows us to model and verify the compliance between *Time* and *Event-Triggered* systems. The approach follows the development schema depicted in Figure 1. Using the EVENT-B refinement technique, we start by an abstract model that specifies the continuous aspects of CPSs. Then we enrich the model step by step to introduce a concrete model that includes the solutions of ODEs. Therefore, the approach consists of four generic models: model *System_M* denotes the continuous part of CPSs, model *EventTriggered_M* describes the interaction between the continuous and the discrete parts of CPSs, model *TimeTriggered_M* represents the discrete time and *TimeTriggeredDesolve*_M introduces the resolution of ODEs in EVENT-B by calling the SAGEMATH functions. The whole models can be downloaded from https://github.com/CPSsWithEventB/Main/blob/main/README.md.

[2] Defined on $\mathbf{IR}^+$

4.1 Generic System Model

Generic System model consists of the context *System_Ctx* that defines a set of constants and axioms required to model the continuous part of CPSs, and the machine *System_M* that introduces two variables: the independent variable $t \in \mathbf{IR}^+$ and the continuous variable *plantV*, defined as a partial function, *plant*V $\in [0,t] \nrightarrow \mathbf{IR}$. Variable t represents the current time and evolves according to the event *Progress* (see Section 3.1) that we have slightly modified to avoid the Zeno problem, where the time interval continually gets smaller and smaller which prevents the controller from reacting exactly at the right moment. For this purpose, we have defined in the context *System_Ctx* a constant parameter sigma whose value depends on

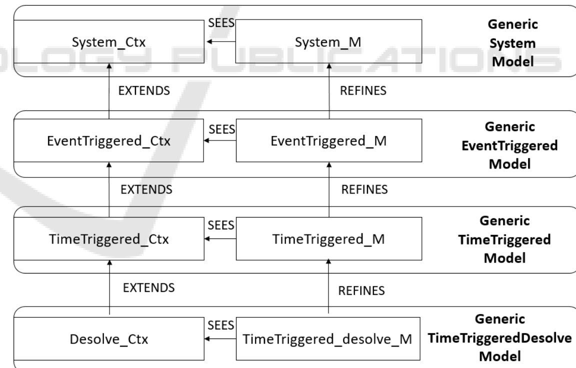

Figure 1: Generic EVENT-B specification with the *B_desolve* function.

$Plant \triangleq$ **ANY** e, $plant1$
WHERE grd1: $e \in DE(\mathrm{IR})$
grd2: $Solvable([0,t] - dom(plantV), e)$
grd3: $plant1 \in [0,t] - dom(plantV) \rightarrow \mathrm{IR} \land AppendSolutionBAP(e,[0,t] - dom(plantV), [0,t] - dom(plantV), plant1)$
THEN act1: $plantV := plantV \mathbin{\lhd\!\!-} plant1$
END

Figure 2: EVENT-B modeling of the plant event.

the properties of the specific system, $sigma \in \mathbf{IR}+$. We added in the event Progress the constraint: $t_1 - t \geq sigma$ to guarantee that the time progression is always greater than *sigma*, where t_1 is a parameter of the event Progress that specifies the new value of time *t*. t_1 is determined according to the properties of the modeled system.

The behavior of the physical part is modeled by the event *Plant* (see Figure 2). Event *Plant* uses the operator *AppendSolutionBAP* defined in *DiffEq*, where *e* represents the abstract ODE and belongs to *DE(S)* which is a set of differential equations built on *S*. *AppendSolutionBAP* enables to update the value of *plantV* (*plantV* $\lhd\!-$ *plant*1) between the last time and the current value of *t* ([0,*t*]−*dom*(*plantV*)).

```
Ctrl ≙ ANY value
WHERE grd1: exec = ctrl
  grd2: value ∈ IR
  grd3: ∀p.p ∈ PROP ∧ value ∉ prop_evade_values(p)
      ⇒ (prop_safe(p))(plantV(t), value) = TRUE
THEN act1: ctrlV := value     act2: exec := prg
END
```

Figure 3: EVENT-B modeling of the controller behaviour.

4.2 Generic EventTriggered Model

At this level, we express the properties desired for the system. To do this, *EventTriggereed_Ctx* extends *System_Ctx* by adding a set of properties *PROP*. Moreover, we define the constant *prop_safe* (resp. *prop_evt_trig*) that maps each property to its safety envelope (resp. the boundary of the safety envelope). Note that the safety envelope is calculated from the safety requirements that the system must satisfy to guarantee that the controller will react exactly at the right moment.

EventTriggered_M model refines *System_M* model to specify the idiom (*ctrl;plant*)[3] whose semantics is as follows: the physical part (*Plant*) evolves in parallel with the time (*Progress*) and both are interrupted just before crossing the boundary of the safety envelope, represented by the formula *prop_evt_trig*. Nevertheless in EVENT-B, it is not possible to state that two events should be executed simultaneously. A first solution to this limitation is to merge the events *Progress* and *Plant* in order to make both evolve at the same time. This option has proved to be unsatisfactory for systems with several physical parts that are modelled then by several *Plant* events. Moreover, making the time progress before the control may lead to a deadlock if the time progresses too fast and the controller cannot take any action to

```
Progress REFINES Progress ≙ ANY t1
WHERE grd1: t1 ∈ TIME ∧ t1 + t ≥ sigma ∧ t < t1
  grd2: exec = prg
  grd3: ∀p.p ∈ PROP ∧ ctrlV ∉ prop_evade_values(p)
⇒ (prop_evt_trig(p))(plantV(t), t1 − t, ctrlV) = TRUE
THEN act1: t := t1     act2: exec := plant
END
```

Figure 4: Refinement of the Progess event in the EventTriggered level.

```
Plant REFINES Plant ≙ ANY plant1
WHERE grd1: exec = plant
grd2: plant1 ∈ [0,t] − dom(plantV) → IR
grd3:ode(f_evol_plantV(ctrlV), plant1(t), t) ∈ DE(IR)
grd4: Solvable([0,t] − dom(plantV),
  ode(f_evol_plantV(ctrlV), plant1(t), t))
grd5: AppendSolutionBAP(ode(f_evol_plantV(ctrlV),
  plant1(t), t), [0,t] − dom(plantV),
  [0,t] − dom(plantV), plant1)
WITH e : e = ode(f_evol_plantV(ctrlV), plant1(t), t)
THEN act1: plantV := plantV ⊲ plant1
  act2: exec := ctrl
END
```

Figure 5: Refinement of the *Plant* event in the *EventTriggered* level.

guarantee the safety of the system. So, we have chosen to keep the time progression as a separate event that is enabled between the *controller* and the *Plant*. For this purpose, we have introduced two new variables *ctrlV* and *exec*. Variable *ctrlV* represents the control variable and belongs to **IR**, exec takes its value in {*prg,ctrl,plant*}; it is a flag used to model the alternation between the progression of time, the control and physical parts as follows: (*ctrl; prg; plant*)[3]. Moreover, we associate with each property a set of values for variable *ctrlV*, called *prop_evade_values* that ensure that the boundary of the safety envelope is never crossed. Thus, the controller is modeled by the event *Ctrl* (see Figure 3) that checks, for each property *p*, that the safety envelope is true if the chosen value value for the control variable *ctrlV* does not belong to the evade values of *p* (Guard *grd3*).

Similarly, the event *Progress* is refined to specify that: (1) the event is enabled when it is the turn of time to progress (Guard *grd2*) and (2) time must not evolve beyond a value that makes the physical part cross the boundary of a safety property (Guard *grd3*) to guarantee that the time will not progress too much and make the safety properties false if the controller fails to react in time. In other words, the new value of time t_1 should be chosen such that the physical part reacts safely during the period from the last progression of time *t* and t_1 ($t_1 - t$). The parts added by refinement are

[3] Defined on $\mathbf{IR}^+$

written in blue (see Figure 4). The physical part, represented by the event *Plant*, is refined by replacing the abstract differential equation e with that defined for a function denoted *f_evol_plantV* to describe the evolution of the state variable *plantV* according to the system discrete state (clause **WITH** of Figure 5).

4.3 Generic TimeTriggered Model

The sensors of the *TimeTriggered* model take periodic measurements of physical parts and its controller executes for each sensors update. The main difference between the *Event*- and the *Time- Triggered* models is in the modeling of the progression of time. The longest time between *TimeTriggered* sensors updates is bounded by a symbolic duration named epsilon and which must be defined by the designer of the specific hybrid system. Thus, the controller is executed at least every *epsilon* units of time. So, we refine event Progress by adding the formula $t_1 - t \leq epsilon$ to state that the time cannot progress by more than *epsilon* units of time. Moreover, since the controller of a *TimeTriggered* model must make a choice that will be safe for up to *epsilon* units of time, we define a new safety envelope named *prop_safeEpsilon*. Event *Ctrl* is refined by adding a guard to ensure that *prop_safeEpsilon* is true for the new value of *ctrlV* when this later does not belong to the evade values of a property (*grd4* of Figure 6).

```
Ctrl REFINES Ctrl ≜ ANY value
WHERE grd1: exec = ctrl
grd2: value ∈ IR
grd3: ∀p.p ∈ PROP ∧ value ∉ prop_evade_values(p)
    ⇒ (prop_safe(p))(plantV(t), value) = TRUE
grd4: ∀p.p ∈ PROP ∧ value ∉ prop_evade_values(P)
⇒ (prop_safeEpsilon(p))(plantV(t), value) = TRUE
THEN act1: ctrlV := value    act2: exec := prg
END
```

Figure 6: Refinement of the *Ctrl* event at *TimeTriggered* level.

Let us remark that the guard related to the formula *safe* is kept because we are in a generic model and do not have yet the concrete expression of *safe*. This guard will be removed when modelling a specific application giving rise to a proof obligation to establish that the other guards imply it.

4.4 Generic TimeTriggeredDesolve Model

Generic *TimeTriggeredDesolve* model refines the generic *TimeTriggered* model to introduce the resolution of ODEs. Depending on the linearity of ODE, a specific SAGEMATH *Desolve* function is used. Thus, we have to distinguish two cases:

- **case of** linear **ODE:** in that case, we use the function *B_desolve* that is defined to model analytical solutions of ODEs in EVENT-B. It returns a function of type $IR \rightarrow IR$ that represents the values of the continuous variables of a given hybrid system. Introducing this function in our generic approach allows us to prove the safety properties of hybrid systems in a *TimeTriggeredDesolve* system, which was not possible with the approach introduced in (Afendi et al., 2020). Moreover, this function serves to establish the link between our EVENT-B models and the differential equation solver SAGEMATH.

  $$B_desolve \in (IN \times IR \times (IR^{+} \rightarrow IR) \times IR^{+} \times (IR^{+} \times IR)) \rightarrow (IR \rightarrow IR)$$

  - the first and the second parameters denote the order and the right term of the considered ODE.
  - the third parameter denotes *the unknown function*, represented by a continuous variable.

```
Plant REFINES Plant ≜ ANY plant1, lastTime
WHERE grd1 : exec = plant
grd2 : lastTime ∈ IR+ ∧ dom(plantV) = [0, lastTime]
grd3 : plant1 = B_desolve(1 ↦ ctrlV ↦ plantV
   ↦ t ↦ (lastTime ↦ plantV(lastTime)))
grd4: plant1 ∈ [0,t] − dom(plantV) → IR
grd5: ode(f_evol_plantV(ctrlV), plant1(t), t) ∈ DE(IR)
grd6: Solvable([0,t] − dom(plantV),
   ode(f_evol_plantV(ctrlV), plant1(t), t))
grd7 : AppendSolutionBAP(ode(f_evol_plantV(ctrlV),
   plant1(t), t), [0,t] − dom(plantV),
   [0,t] − dom(plantV), plant1)
THEN act1: plantV := plantV ◁ plant1
   act2: exec := ctrl
END
```

Figure 7: Refinement of the *Plant* event by calling the DE solver.

  - the fourth parameter denotes the independent *variable*, represented by a discrete variable and typed by $\mathbf{IR}^{+}$.
  - the last parameter denotes the initial values of both the independent variable and the unknown function.

  Event *Plant* is refined to calculate the value of *plant1* during the period from *lastTime* to t using the function *B_desolve*, which is specified by the guard *grd3* (see Figure 7). This guard is used to link the abstract event to its refinement. It enhances the guards grd6 and grd7 that aim at modeling the differential equation solution

using the operators of the theory DiffEq. It serves to verify the properties assumed by the operators Solvable and AppendSolutionBAP about the solutions of the given ODEs in order to establish a link between our approach and the theory DiffEq. The parameters dvar, ivar and ics (see Section 2.2) are represented respectively by the dependent variable plantV and the independent variable t as well as the initial values of these variables. The parameter lastTime is introduced to represent the last progression of time at which plantV has been calculated. The solution of a given differential equation is calculated from lastTime to t in order not to overwrite the old values of the continuous variable plantV.

- **case of non linear ODE:** if the ODE is linearisable, we apply the same refinement using the Desolve function on the linear form while proving such a linearisation using the approach defined in (Dupont et al., 2021). Otherwise, we use the SAGEMATH *desolve_rk4()* function that returns an approximate solution for ODEs. This function is defined by:

  $B_desolve\ rk4 \in \mathbb{R}\times(\mathbb{R}^+ \mapsto \mathbb{R})\times\mathbb{R}^+\times(\mathbb{R}^+ \times\mathbb{R})\times(\mathbb{R}^+ \times\mathbb{R}^+) \rightarrow (\mathbb{R}^+ \mapsto\mathbb{R})$

  B_desolve_rk4 returns a function of type $\mathbf{IR}^+ \mapsto \mathbf{IR}$ that represents the values of the continuous variables. The first four parameters are the same parameters as those of *B_desolve*. The last parameter is used to specify the interval denoted [*lastTime,t*] for which the values of *plantV* are calculated. The refinement of the event *Plant* in case of non linear ODE is similar to that for linear ODE with the guard grd3 replaced with:

  $plant1 = B_desolve_rk4(f_evol(ctrlV) \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)) \mapsto (lastTime \mapsto t))$

4.5 Modeling the Safety Properties

The main goal of the discrete part represented by the controller is to ensure the safety properties of a specific hybrid system. To model these safety properties in EVENT-B, a constant function $prop \in \mathbf{IR}^n \rightarrow BOOL$ is defined in the context *Desolve* where *n* denotes the number of variables occurring in the property. Then an invariant is added in the machine *TimeTriggered_desolve_M*, *inv*[4]: $\forall x \cdot x \in dom(plantV) \Rightarrow prop(plantV(x)) = TRUE$, where *plantV* will be replaced by the specific continuous variables. To discharge the PO generated for this invariant, we added to the event Plant the following guard grd8: $\forall xx \cdot xx \in dom(plant1) \Rightarrow prop(plant1(xx)) = TRUE$. This guard will be removed on a specific case to generate a proof obligation that aims at proving that this guard is actually satisfied.

[4] 2 x and xx enable to cover all the moments from the beginning until the current time

4.6 Correctness of the Specification

Table 1: RODIN Proof statistics for the generic models.

Specific_Models	Tot.	Aut.	Man.
System	8	1	7
EventTriggered	19	11	8
TimeTriggered	2	1	1
TimeTriggeredDesolve	5	3	2

Table 1 gives the statistics of the POs generated for the correctness of our generic models where. It is noticeable that 47% of them were automatically discharged. These POs include the correctness of the events that model the behavior of the physical parts and the controller, as well as the correctness of their refinement. The POs related to the well-definedness have been interactively discharged under RODIN using the properties of *Reals* and *DiffEq* theories. To prove the compliance between *TimeTriggered_M* and *EventTriggered_M* machines, Rodin has generated a set of proof obligations that we have discharged in the *TimeTriggered_M* machine. In these generic models, as we have kept the guard related to the formula safe and *prop_evt_trig* in the event *Progress* (see Figure 4), the refinement proofs are rather simple and related mainly to the type checking of the different variables and the feasibility of the abstract and the concrete models.

To prove the correctness of the *TimeTriggeredDesolve* model, RODIN has generated five proof obligations, three of them were automatically discharged. The remaining proof obligations are as follows:

- *PO1* is a well-definedness proof obligation which aims at proving that the guard *grd3* of the event *Plant* (see Figure 7), added to model the solution of the generic ODEs using the function *B_desolve*, is well defined. This guard assigns to

the parameter *plant1* the solution of the generic ordinary differential equation obtained using the function *B_desolve*. To discharge this proof obligation, we must prove that the set of the results returned by *B_desolve* is equal to the set of definition of *plant1*. This proof obligation was discharged using some rewriting rules, the properties of the *Reals* theory and some invariants defined in refined machines.

- *PO2* is generated to prove that the event *Plant* preserves the system safety property, specified using the constant *prop*. This proof obligation was discharged by replacing the value of *plant1* by the result returned by *B_desolve*.

PO1: $lastTime \in dom(plantV) \land plantV \in \mathbb{R} \rightarrow \mathbb{R} \land 1 \mapsto ctrlV \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)) \in dom(B_desolve) \land B_desolve \in \mathbb{N} \times \mathbb{R} \times P(\mathbb{R} \times \mathbb{R}) \times \mathbb{R} \times (\mathbb{R} \times \mathbb{R}) \rightarrow P(\mathbb{R} \times \mathbb{R})$

PO2: $\forall x \cdot x \in dom(plantV \Leftarrow plant1) \implies prop((plantV \Leftarrow plant1)(x)) = TRUE$

4.7 Instantiating the Generic Approach

To design a specific case study following our generic approach, we instantiate the generic *TimeTriggered-Desolve* model by replacing the generic continuous variable *plantV* by that or those associated with the case study. If the case study includes several continuous variables, it is necessary to define a set of parameters with the same cardinality as the set of continuous variables. The function *B_desolve* is then instantiated by the specific parameters of the modeled case study. We make the assumption that the safety property is in a conjunctive normal form ($\land_{i=1..n}\ p_i$) and that, for each formula p_i, the following elements are specified: $event_trig_i$, $safe_i$, $safeEpsilon_i$ and a set of evade values $evade_values_i$ for *ctrlV*. In that case, the instantiation consists in valuing the set *PROP* and the different constants as follows where $X \in \{event_trig, safe, safeEpsilon, evade_values\}$:

$PROP = \cup_{i=1..n}\{p_i\}$, $prop_X = \cup_{i=1..n}\{p_i \mapsto val_i\}$

The safety property represented by the formula prop also needs to be instantiated in the specific model (see Section 6).

5 A TOOL FOR SUPPORTING THE APPROACH

To make our approach workable, we built a new RODIN plug-in, called SAGEMATH plug-in, that interfaces the RODIN platform with SAGEMATH solver to calculate the solutions of ODEs. Solving ODEs is needed for proving the safety property and the satisfiability of a guard removed in a refinement. In other words, during the proof of a PO, SAGEMATH needs to be called on each term *B_desolve(...)* or B *desolve_rk4(...)* in order to replace it by the solution of the corresponding ODE. The use the SAGEMATH plug-in follows the 3- steps general process: (1) calling SAGEMATH from RODIN, (2) solving differential equations and (3) using the result returned in RODIN (see Figure 8). In the first step, an input field that allows calling SAGEMATH from RODIN appears automatically when the current PO contains the terms *B_desolve* (resp. *B_desolve rk4(...))*. The second step consists in calling manually a predefined script generated systematically from the function *B_desolve(...)* (resp. *B_desolve_rk4(...)*). The user must generate a specific SAGEMATH script according to the structure and the nature of the differential equation to be solved. The last step consists in manually translating the result of SAGEMATH according to EVENT-B syntax. The user must use the theory of reals to translate the result and add it as an hypothesis to prove the current PO. The description of the different stages that constitute the main scenario use of the SAGEMATH plug-in can be downloaded from https://github.com/CPSsWith EventB/Main/blob/main/README.md. Hereafter, we describe each step.

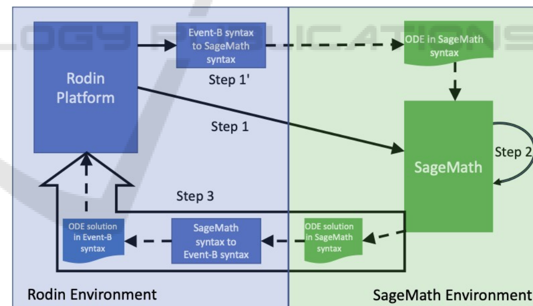

Figure 8: The general process.

5.1 Calling SAGEMATH from RODIN (Step1)

To call SAGEMATH from RODIN, a button called sage has been added in the proof window using an Eclipse plug-in. The button is made available on a hypothesis/goal that contains the term *B_desolve* (resp. *B_desolve_rk4*(...)). To develop a RODIN plug-in, Eclipse provides a set of Java interfaces. These interfaces are intended to be implemented according to the goal of the plugin. To implement the *SAGEMATH* plug-in using Eclipse IDE, we define the

following Java classes: *SageTacticProvider, SageApplication and SageTactic*.

5.1.1 SageTacticProvider Class

This Java class implements the method *getPossibleApplications* to check the presence of the term *B_desolve* (resp. *B_desolve:rk4(...)*) in each proof obligation and returns an instance of the *SageApplication* class. Function *getPossibleApplications* uses two main predicates, *pred*, a local variable, and *hyp*, a parameter of the function. *pred* takes as value *hyp* if this latter is not null or the current proof obligation otherwise. If the tags, the left and right parts of the formula pred, are equal to those of the predicate that contains *B_desolve* (resp. *B_desolve_rk4(...)*), we return the list of tactics that can be applied as a list of instances of *SageApplication* or null otherwise. This is repeated for each node of the proof tree.

5.1.2 SageTactic

Class This Java class implements the *apply()* method that creates a process for calling SAGEMATH. Function *apply()* contains all the instructions that will be applied when calling SAGEMATH. The process for calling SAGEMATH is created using the predefined Java class *Process* and provides the path of the executable file of SAGEMATH to the predefined Java class *ProcessBuilder*. The Java class *ProcessBuilder* can be used to call external applications thanks to the *start()* method and the Java class *Process* can be used to create new system processes.

5.1.3 SageApplication Class

This Java class establishes the link between the checking of the presence of the function *B_desolve* (resp. *B_desolve_rk4(...)*) in the current proof obligation and the call to SAGEMATH. It implements in particular two methods:

- *getHyperlinkLabel()*: allows to display the button sage in the proof window.
- *getTactic()*: allows to create an instance of the class *SageTactic* to execute the *apply()* method.

5.2 Solving Odes in SAGEMATH (Step1' and Step2)

In Steps 1′ and 2, the ODE is resolved. Step 1′ is viewed as a preliminary step of 1. It consists in systematically generating a SAGEMATH script from the EVENT-B functions *B_desolve* and *B_desolve_rk4(...)*, with all the parameters necessary

```
1: ctrlV = var('ctrlV')     2: t = var('t')
3: plantV = function('plantV')(t)
4: lastTime = var('lastTime')
5: eq = desolve(diff(plantV, t, 1) == ctrlV,
dvar = plantV, ivar = t,
ics = [lastTime, plantV(lastTime)])
6: o = open('sageresult.txt', 'w')
7: o.write(str(eq))     8: o.close()
```

Figure 9: B_desolve script.

to execute the SAGEMATH predefined functions *desolve* and *desolve_rk4*. In such a script, the differential equation must be expressed depending on the controlled variable *ctrlV* that links the continuous and the discrete parts of an hybrid system. A script is executed in SAGEMATH using the following command: *load('scriptName.sage')*. The script below is generated from the formula $B_desolve(1 \rightarrow ctrlV \rightarrow plantV \rightarrow t \rightarrow (lastTime \rightarrow plantV(lastTime)))(x)$. It solves a differential equation plantV′ = ctrlV, where (see Figure 9):

- Statement 1 is generated using the second parameter of B *desolve* and it specifies the right part of the ODE *plantV′ = ctrlV*.
- Statement 2 is generated using the forth parameter of *B_desolve* and it specifies the definition of the independent variable *t*.
- Statement 3 is generated using the third parameter of *B_desolve* and it specifies the definition of the continuous variable represented by *plantV*. The definition of this variable must always be after the definition of the independent variable.
- Statement 4 is generated using the first part of the parameter that specifies the initial conditions and it represents the last progression of time from which we calculate the values of the continuous variable.
- Statement 5 represents the call to the SAGEMATH predefined function *desolve*. The first parameter 1: ctrlV = var(′ ctrlV′) 2: t = var(′ t ′) 3: plantV = f unction(′plantV′)(t) 4: lastTime = var(′ lastTime′) 5: eq = desolve(di f f(plantV,t,1) == ctrlV, dvar = plantV,ivar = t, ics = [lastTime, plantV(lastTime)]) 6: o = open(′ sageresult.txt′ , ′w ′) 7: o.write(str(eq)) 8: o.close() Figure 9: B desolve script. of this function is generated using the first, second and third parameters of B desolve. The second, third and forth parameters are generated respectively using the third, forth and last parameters of *B_desolve*.

- Statements 6-8 generate a text file, named ”sageresult.txt”, which stores the result of the differential equation specified by *eq* in Statement 5.

5.3 Using SAGEMATH Results in RODIN (Step3)

In Step 3, the term *B desolve()* is replaced by the result returned by SAGEMATH and stored in a text file. For that purpose, the predicate (*B_desolve() = sol'*) is added as an additional hypothesis in the current PO, where *sol'* is a rewritten of *sol* according to the syntax of the theory of reals. Basically, this theory adopts a prefix style by defining a keyword for each operator on the reals like *plus* for addition, *times* for multiplication, etc. So for instance, the formula *ctrlV* × *lastTime* + *plantV*(*lastTime*) is rewritten into *plus*(*times*(*ctrlV*→ *lastTime*)→ *plantV*(*lastTime*)).

6 APPLICATION

To demonstrate the feasabilty of our approach we have applied it on several frequently CPS case studies like the following ones:

- The *Stop Sign* System whose objective is to stop a car before a stop signal *SP*. The control strategy is to adjust the velocity of the car by accelerating or braking. The continuous behavior of this system is modeled by the position *p* and the velocity *v* of the car, as well as its acceleration *a*. This continuous behavior evolves according to two linear ODEs, $\frac{dp}{dt} = v(t)$ and $\frac{dv}{dt} = a$.
- The Hybrid *Water Tank* System whose objective is to maintain the water level between a high level *V_high* and a low level *V_low* with *0 < V low < V high*. The system includes a tank, a pump to fill the tank and a sensor to get the level of the water. The water level is specified by the variable *V* that evolves according to the flowing linear ODEs, $\frac{dV}{dt} = f_in$ when the pump is activated and $\frac{dV}{dt} = -f_out$ otherwise. The pump is activated (resp. disabled) to fill (resp. empty) the water tank by f in (*resp. –f_out*) as long as the property *V < V_high (resp. V > V_low*) is true.

These case studies are didactic and quite representative of linear hybrid systems that admit exact solutions (polynomial ODE solutions). The continuous behavior of the *Stop Sign* case study is represented by two state variables *p* and *v* while that of the *Water Tank* is represented by a single state variable *V*. Moreover, the *Stop Sign* case study is represented by three different modes of control, *Accelerating, Braking* and *Stopped*, that require a single safety envelope. The system can enter state *Accelerating* when the car is very far from the stop signal *SP*. State *Braking* is entered when the car is very close to the stop signal *SP*. The state *Stopped* is entered when the car is stopped i.e *v* = 0 (consequently *a* = 0) presumably right before signal *SP*. In the other hand, the *Water Tank* case study is composed of two modes and when its controller enter one of these two modes the other one is considered as an evade mode which requires the use of two safety envelops. This diversity will allow us to properly illustrate the use of our generic approach. For the *Water Tank* case study, we chose to directly instantiate the generic model *TimeTriggeredDesolve* in order to use the interface between RODIN and SAGEMATH to solve the associated ODEs. For the Stop Sign case study, we chose to start by modeling the controlled system by refining the generic *EventTriggered* model. The associated Stop Sign *EventTriggered* model will then be refined by a Stop Sign *TimeTriggered* model in which we directly model the solutions of the associated ODEs without using the function B desolve. The whole EVENT-B specification can be downloaded from https://github.com/ CPSsWithEventB/Main/blob/main/README.md. In this paper, we present in details the modeling of the Water Tank.

6.1 The Modeling of the Water Tank System

To model the *Water Tank* case study using our approach, we follow the schema depicted by Figure 10. The instantiation starts by directly refining the generic model *TimeTriggeredDesolve_M* to obtain the model *WaterTank_M* that sees (Abrial, 2010) the context *WaterTank_Ctx*. The safety property is expressed in a conjunctive normal form ($V_{low} \leq V \wedge V \leq V_{high}$). So the context *WaterTank_Ctx* contains the following elements[5]:

[5] *bool(P)* gives the boolean value of the predicate *P*.

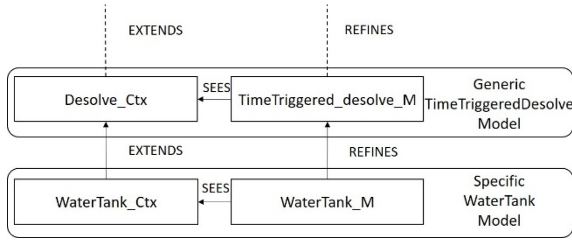

Figure 10: Architecture of the EVENT-B model of the water tank system.

$$
\begin{aligned}
&PROP=\{p_1, p_2\}\\
&prop_event_trig=\{p_1 \mapsto (\lambda\, V \mapsto t_1 \mapsto ctrlV \cdot V \in \mathbb{R} \land\\
&\qquad ctrlV \times t_1 \in \mathbb{R} \mid bool(V + ctrlV \times t_1 \geq V_{low})),\\
&p_2 \mapsto (\lambda\, V \mapsto t_1 \mapsto ctrlV \cdot V \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid\\
&\qquad bool(V + ctrlV \times t_1 \leq V_{high}))\}\\
&prop_safe=\{p_1 \mapsto (\lambda\, V \mapsto ctrlV \cdot V \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid\\
&\qquad bool(V > V_low)),\\
&\qquad p_2 \mapsto (\lambda\, V \mapsto ctrlV \cdot V \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid\\
&\qquad bool(V < V_{high}))\}\\
&prop_safeEpsilon=\{p_1 \mapsto (\lambda\, V \mapsto ctrlV \cdot V \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid\\
&\quad bool(V + (ctrlV \times epsilon) > V_{low})),\\
&\quad p_2 \mapsto (\lambda\, V \mapsto ctrlV \cdot V \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid\\
&\qquad bool(V + (ctrlV \times epsilon) < V_{high}))\}\\
&prop_evade_values=\{p_1 \mapsto \{f_{in}\}, p_2 \mapsto \{-f_{out}\}\}
\end{aligned}
$$

Machine *WaterTank_M* defines the invariants of Figure 11. Invariant **inv1** is defined to replace the generic continuous variable *plantV* by the specific one represented by the water level V. Invariant **inv2** specifies the possible values of the variable *ctrlV*. Invariant **inv3** models the safety property using the function $prop \in \mathbf{IR} \rightarrow BOOL$ where the formula *prop(V(x))* is defined in the context *WaterTank_Ctx* by $V(x) \leq V_high \land V(x) \geq V_low$.

$$
\begin{aligned}
&\textbf{inv1}\!: plantV = V \land ran(V) \subseteq \mathrm{IR}\\
&\textbf{inv2}\!: ctrlV \in \{f_{in}, -f_{out}\}\\
&\textbf{inv3}\!: \forall x \cdot x \in dom(V) \Rightarrow prop(V(x)) = TRUE
\end{aligned}
$$

Figure 11: Invariant of the water tank system.

The event *Plant*, of the generic machine *TimeTriggered_desolve_M* is refined by replacing the generic parameters of the function *B_desolve* by those related to the ODEs: $\frac{dV}{dt} = f_{in}$ if the pump is *On* and $\frac{dV}{dt} = -f_{out}$ otherwise (see Figure 12). These ODEs are expressed in the function *B_desolve* as follows: *B_desolve(1→ ctrlV→V→ t → (lastTime → V(lastTime)))*, where the integer 1 denotes the degree of both ODEs, *ctrlV* is the controlled variable, V rep resents the dependent variable for which we calculate the values over the time t that denotes the independent variable, and *lastTime* and *V*(*lastTime*) represent respectively the initial values of the time t and the value of continuous variable V at the instant *lastTime*.

$$
\begin{aligned}
&Plant\ \textbf{REFINES}\ Plant\ \textbf{ANY}\ lastTime,\ plant1\\
&\textbf{WHERE grd1}\!: exec = plant\\
&\textbf{grd2}\!: lastTime \in \mathrm{IR}^{+} \land dom(V) = [0, lastTime]\\
&\textbf{grd3}\!: plant1 = B_desolve(1 \mapsto ctrlV \mapsto V \mapsto t\\
&\quad \mapsto (lastTime \mapsto V(lastTime)))\\
&\textbf{grd4}\!: ode(f_evol_plantV(ctrlV), plant1(t), t) \in DE(\mathrm{IR})\\
&\textbf{grd5}\!: Solvable([0,\ t]\\
&\quad - dom(V, ode(f_evol_plantV(ctrlV), plant1(t), t))\\
&\textbf{grd6}\!: AppendSolutionBAP(ode(f_evol_plantV(ctrlV),\\
&\quad plant1(t), t), [0, t] - dom(V), [0, t] - dom(V), plant1)\\
&\textbf{THEN act1}\!: V := V \lhd plant1 \qquad \textbf{act2}\!: exec := ctrl\\
&\textbf{END}
\end{aligned}
$$

Figure 12: The Plant event associated to the water tank system.

$$
\begin{aligned}
&\textbf{inv4}\!: \forall x \cdot x \in PROP \land\\
&ctrlV \notin prop_evade_values(x) \land exec = prg \Rightarrow\\
&(prop_safeEpsilon(x))(V(t) \mapsto ctrlV) = TRUE
\end{aligned}
$$

Figure 13: Implicit property on the system.

6.2 Correctness of the Specification of the Water Tank

Machine *WaterTank_M* generates 102 POs, 39% of them are automatically discharged. Like for the generic models, the POs related to the guard feasibility and well-definedness have been interactively discharged under RODIN thanks to several provers like SMT and AtelierB provers but also inference rules described in the RODIN theory that implements reals. The use of these inference rules made the proof activity longer since they are not automatically applied even on simple examples like the transitivity rule.

To discharge the POs related to the guards feasibility and well-definedness, we needed to add invariants that translate implicit properties on the system (see Figure 13). This invariant specifies that the system is safe if the controller has chosen a value for *ctrlV* that does not belong to the sets *prop_evade_values(x)*. Moreover, we added an invariant which ensures that before executing the physical part, the safety property is satisfied during the period *epsilon* (see Figure 14).

The most important part in the proof phase is the one concerning the safety property specified using the invariant *inv3*. RODIN generated for this invariant the PO of Figure 15 for removing the *grd8* defined in Section 4.5. This PO is obtained by replacing *plant1* in V := V ⩤ plant1 by *B_desolve(1→ ctrlV→ V→ t → (lastTime→ V(lastTime)))* (see *grd3* of event *Plant*). To discharge this PO, we call SAGEMATH to obtain the exact solution of the related ODE.

$$
\begin{array}{l}
\textbf{inv5} : \exists t1 \cdot t1 \in RRealPlus \land \\
minus(t \mapsto t1) \mapsto epsilon \in leq \\
dom(V) = Closed2Closed(Rzero, t1) \land \\
\land (exec = plant \Rightarrow t1 = t) \land \\
(exec = plant \Rightarrow t \mapsto t1 \in gt) \land (\forall x \cdot x \in PROP \land \\
ctrlV \notin prop_evade_values(x) \land exec = plant \\
\Rightarrow (prop_safeEpsilon(x))(V(t1) \mapsto ctrlV) = TRUE)
\end{array}
$$

Figure 14: Preservation of the property during a cycle of ε units of time.

$$
\begin{array}{l}
B_desolve(1 \mapsto ctrlV \mapsto V \mapsto t \mapsto \\
\qquad (lastTime \mapsto V(lastTime)))(x) \leq V_high \land \\
B_desolve(1 \mapsto ctrlV \mapsto V \mapsto t \mapsto \\
\qquad (lastTime \mapsto V(lastTime)))(x) \geq V_low
\end{array}
$$

Figure 15: The PO related to the safety property.

We must then replace the function *B_desolve(...)* with the solution returned by the call to SAGEMATH from RODIN. This solution is obtained by replacing the generic variable *plantV* by *V* in the script of Figure 9.

Let us remark that for the non-linear ODE, the proof of the safety property is achieved by assuming the monotonicity of the function returned by *desolve_rk4* on the interval [*lastTime,t*]. For that purpose, we have to prove the following property on the returned function to state that it is increasing or decreasing:

$$
\begin{array}{c}
\forall tt \cdot tt \in [lastTime,t] \Rightarrow \\
(plant1(tt) \geq plant1(lastTime) \land plant1(tt) \leq plant1(t)) \\
\lor (plant1(tt) \leq plant1(lastTime) \land plant1(tt) \geq \\
plant1(t))
\end{array}
$$

Having this property as verified, the proof of a safety property comes down to prove it for the lower and/or the upper bounds.

6.3 Discussion on the Proof Activity

From both case studies that we have modeled and verified to prove the feasibility of our approach, the following conclusion can be drawn. The complexity of the application-dependent proofs is proportionate to the number of the terms of the ordinary differential equation solution. In other words, the higher the degree of the ordinary differential equation, the higher the complexity of the proofs: the proofs of the *Stop Sign* case study took more than one week while 2 days were enough for the *Water Tank* case study. We think that the development of an inference engine for the theory that implements the reals would help speed up the proof activity. Such an inference rule would automate the application of some inference rules like reflexivity, transitivity, etc.

7 CONCLUSION AND FUTURE WORKS

This paper has presented a proof-based approach that uses the EVENT-B refinement technique to model and verify the correctness of CPSs whose behavior is described using ODEs. This approach combines the EVENT-B formal method with the differential equation solver SAGEMATH by modeling and implementing the call to the solver. The approach is supported by a tool, built as a RODIN plug-in, that establishes the bridge between EVENT-B and SAGEMATH.

To cope with the complexity of the system, the built EVENT-B specification consists of four generic models: *System* model that represents the continuous aspects of CPSs *EventTriggered* model that specifies the interactions between the discrete and the continuous parts of CPSs, *TimeTriggered* model that specifies the discrete time of the discrete part of CPSs and *TimeTriggeredDesolve* that introduces a function to model the call to a DE solver, called either B desolve when treating linear ODEs and *B_desolve_rk4* when treating nonlinear ODEs.

The proposed approach was successfully applied on several case studies like the water tank system presented in this paper but also those with multiple continuous variables such as the *Stop Sign* case study https://github.com/CPSsWithEventB/Main/blob/main/README.md. We admit that the chosen case study is a simple hybrid system with a linear ODE but it served us to describe the different steps for applying our generic approach. Using SAGEMATH, we can deal with more complex ODEs as we showed by modeling the function *desolve_rk4* which solves nonlinear ODEs. Without solving ODEs, our models were abstract and did not allow proving the safety properties of hybrid systems. As future work, we plan to apply our approach on more complex case studies.

REFERENCES

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

Afendi, M., Laleau, R., and Mammar, A. (2020). Modelling hybrid programs with event-b. In *Rigorous StateBased Methods: 7th International Conference, ABZ 2020, Ulm, Germany, May 27–29, 2020, Proceedings*, pages 139–154. Springer.

Banach, R., Butler, M., Qin, S., Verma, N., and Zhu, H. (2015). Core Hybrid Event-B I: Single Hybrid EventB Machines. *Science of Computer Programming*, 105:92–123.

Butler, M., Abrial, J.-R., and Banach, R. (2016). Modelling and Refining Hybrid Systems in Event-B and Rodin.

Butler, M. and Maamria, I. (2010). Mathematical Extension in Event-B through the Rodin Theory Component.

Dupont, G., Aït-Ameur, Y., Pantel, M., and Singh, N. K. (2018). Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B. In Butler, M., Raschke, A., Hoang, T. S., and Reichl, K., editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 155–170, Cham. Springer International Publishing.

Dupont, G., Ameur, Y. A., Pantel, M., and Singh, N. K. (2021). Event-B Refinement for Continuous Behaviours Approximation. In Hou, Z. and Ganesh, V., editors, *Automated Technology for Verification and Analysis 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 320–336. Springer.

Frehse, G., Guernic, C. L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O. (2011). Spaceex: Scalable verification of hybrid systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer.

Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., and Platzer, A. (2015). KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *International Conference on Automated Deduction*, pages 527–538. Springer.

Henzinger, T. A., Ho, P., and Wong-Toi, H. (1997). HYTECH: A model checker for hybrid systems. *Int. J. Softw. Tools Technol. Transf.*, 1(1-2):110–122. Kopetz, H. (1991). Event-Triggered Versus Time-Triggered Real-Time Systems. In *Operating Systems of the 90s and Beyond*, pages 86–101. Springer.

Loos, S. M. and Platzer, A. (2016). Differential Refinement Logic. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–10. IEEE.

DISCONT ANR Project (2017). https://discont.loria.fr.

Perko, L. (2013). *Differential Equations and Dynamical Systems*, volume 7. Springer Science & Business Media.

Platzer, A. (2008). Differential Dynamic Logic for Hybrid Systems. *Journal of Automated Reasoning*, 41(2):143–189.

Platzer, A. and Quesel, J.-D. (2008). KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In *International Joint Conference on Automated Reasoning*, pages 171–178. Springer.

Su, W., Abrial, J.-R., and Zhu, H. (2014). Formalizing Hybrid Systems with Event-B and the Rodin Platform. *Science of Computer Programming*, 94:164–202.

Wolfram, S. (2003). The Mathematica Book, 5th edn. Wolfram Media, Champaign (2003).

Zimmermann, P., Casamayou, A., Cohen, N., Connan, G., Dumont, T., Fousse, L., Maltey, F., Meulien, M., Mezzarobba, M., Pernet, C., et al. (2018). *Computational Mathematics with SageMath*. SIAM.