A correct by construction approach for the modeling and the verification of cyber-physical systems in Event-B

Meryem Afendi

A Correct by Construction Approach for the Modeling and the Verification of Cyber-Physical Systems in Event-B

---

par

Meryem Afendi

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE PARIS-EST CRÉTEIL

Laboratoire d'Algorithmique Complexité et Logique
Université Paris Est Créteil, Créteil, France

Composition du Jury:

Président:

Nom du Président

Rapporteurs:

Richard Banach, Senior Lecturer, University of Manchester

Frédéric Mallet, Professeur, Université Côte d'Azur

Examinateurs:

Dominique Méry, Professeur, Université de Lorraine

Burkhart Wolff, Professeur, Université Paris-Saclay

Co-directrices:

Régine Laleau, Professeure, Université Paris-Est Créteil

Amel Mammar, Professeure, Télécom-Sud Paris

Summary

Hybrid systems are one of the most common mathematical models for Cyber-Physical Systems (CPSs). They combine discrete dynamics represented by state machines or finite automata with continuous behaviors represented by differential equations. The measurement of continuous behaviors is performed by sensors. When these sensors have a continuous access to these measurements, this kind of models is called *Event-Triggered* models. The properties of such models are easier to prove, while their implementation are difficult in practice. Therefore, it is preferable to introduce a more concrete kind of models, called *Time-Triggered* models, where the sensors take periodic measurements. Contrary to *Event-Triggered* models, *Time-Triggered* models are much easier to implement, but much more difficult to verify. Based on the differential refinement logic (dR$\mathcal{L}$), a dynamic logic for refinement relations on hybrid systems, it is possible to prove that a *Time-Triggered* model refines an *Event-Triggered* model. However, being done by hand, this proof is error-prone since no prover is available to support this logic. 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. We use the Rodin platform to develop Event-B models and its associated provers (automatic and interactive) to ensure their correctness. The obtained Event-B models are generic and can be then instantiated to model and prove any specific CPS. The proposed approach is illustrated by two frequently used CPS case studies. Moreover, this approach implements an interface between the differential equation solver SageMath (System for Algebra and Geometry Experimentation) and the Rodin tool to deal with the resolution of ordinary differential equations in Event-B. The proposed approach was successfully applied on a frequently used cyber-physical system case study. We used our approach on various problems taken from control theory, including the stop sign, the water tank, thermostat.

Résumé

Les systèmes hybrides sont l'un des modèles mathématiques les plus courants pour la modélisation des systèmes cyber-physiques (SCP). Ils combinent des dynamiques discrètes représentées par des machines à états ou des automates finis avec des comportements continus représentés par des équations différentielles. Les comportements continus des systèmes cyber-physiques sont collectés à l'aide des capteurs. Lorsque ces capteurs ont un accès continu à ces mesures, ce type de modèles est appelé modèles déclenchés par événements *Event-Triggered*. Les propriétés de tels modèles sont plus faciles à prouver, tandis que leur mise en œuvre est difficile en pratique. Par conséquence, il est préférable d'introduire un type de modèles plus concrets, appelés modèles déclenchés par le temps *Time-Triggered*, où les capteurs prennent des mesures périodiques. Contrairement aux modèles *Event-Triggered*, les modèles *Time-Triggered* sont beaucoup plus faciles à mettre en œuvre, mais beaucoup plus difficiles à vérifier. En utilisant dR$\mathcal{L}$, une logique dynamique de raffinement pour les systèmes hybrides, il est possible de prouver qu'un modèle *Event-Triggered*, les modèles *Time-Triggered* raffine un modèle *Event-Triggered*. Cependant, étant faites à la main, les preuves dans dR$\mathcal{L}$ sont sujette aux erreurs puisqu'aucun outil n'est disponible pour supporter cette logique. Pour surmonter cette limite, nous proposons une nouvelle approche pour prouver ce raffinement, basée sur Event-B pour tirer parti de son processus de raffinement bien défini et de ses outils de support. Nous utilisons la plateforme Rodin pour développer des modèles Event-B et ses outils de preuves associés (automatiques et interactifs) pour assurer l'exactitude de ces modèles. Les modèles Event-B obtenus sont génériques et peuvent ensuite être instanciés pour modéliser et prouver n'importe quel système cyber-physique. L'approche proposée est illustrée par deux études de cas fréquemment utilisées. De plus, cette approche implémente une interface entre le solveur d'équations différentielles SageMath (System for Algebra and Geometry Experimentation) et l'outil Rodin pour traiter la résolution des équations différentielles ordinaires dans Event-B. L'approche proposée a été également appliquée avec succès sur une étude de cas de système cyber-physique fréquemment utilisé. Nous avons utilisé notre approche sur divers problèmes issus de la théorie du contrôle, notamment le panneau d'arrêt, le réservoir d'eau, le thermostat.

Acknowledgement

This work is supported by the DISCONT Project (https://discont.loria.fr) of the French National Research Agency (ANR), Grant Number ANR-17-CE25-0005. Many thanks to the members of this project for their great help in polishing my work and publishing it.

I would like to thank my supervisor Prof. Régine Laleau and Prof. Amel Mammar for their consistent support and guidance during the running of this thesis. They continuously provided encouragement and was always willing and enthusiastic to assist in any way she could throughout the research project. Finally, many thanks to all participants that took part in the study and enabled this research to be possible.

Finally, i would like to thank my parents, my brother, and my friends. It would have been impossible to finish my studies without their unwavering support over the past few years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Context: Recent progress in the industrial sector has allowed the development of a new production model based on digital network architectures to give birth to a fourth industrial revolution ("industry 4.0" or "industry of the future"). Cyber-physical systems (CPSs) [2] are one of the main technologies in this industry and form the basis of future technologies. The domain of these systems has rapidly become a source of innovation with applications in many sectors: health, transport, smart grid, etc. This type of systems allows the discrete virtual world and the continuous physical world to be connected via a network of sensors and actuators.

A common mathematical model for CPSs is that of hybrid systems that combine discrete behavior represented by state machines or finite automata with continuous behavior described by differential equations. In hybrid systems, the measurement of continuous behaviors is performed by sensors. Ideally, sensors have a continuous access to these measurements, this can be captured by an abstract model of CPSs, called *Event-Triggered* system by Kopetz [3]. However, implementing such models is difficult in practice. Therefore, it is preferable to introduce a more concrete model, called *Time-Triggered* system [3] in, where the sensors take periodic measurements. Platzer et al. [4, 5] use *Event* and *Time-Triggered* models to design and verify hybrid systems. They have proved that a *Time-Triggered* model is a refinement of an *Event-Triggered* model, by using an extension of differential dynamic logic (d$\mathcal{L}$), called differential refinement logic (dR$\mathcal{L}$).

Challenges: The continuous behavior of hybrid systems is often described by ordinary differential equations (ODEs) that involve an unknown function depending on a single variable. There are two types of methods for solving ordinary differential equations: analytical (symbolic) methods and numerical methods. Analytical methods use a set of theorems to obtain an exact solution for a given differential equation. For example, the computer algebra SageMath (System for Algebra and Geometry Experimentation) [6] provides a predefined function that uses analytical methods to find analytical solutions for ODEs. However most differential equations cannot be solved exactly. Therefore, we must rely on numerical methods to obtain approximate solutions or use approximation techniques to transform an equation into an equivalent equation with an exact solution. For example, linearization techniques can be used to transform a nonlinear differential equation into a linear differential equation and then apply analytical methods for linear differential equations. The obtained solution is thus an approximate solution for the original one.

The interaction between the software part and the physical world makes the verification of hybrid systems an intellectual challenge. The development of techniques and tools to effectively design hybrid systems has drawn the attention of many researchers. Traditional approaches are based on simulation tools like Matlab/Simulink [7] or Stateflow [8] which are however time-consuming and produce results tainted with uncertainty, this is why hybrid systems with critical safety properties involve the use of formal methods. For this purpose, several formal approaches have been proposed. These approaches can be grouped 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. They are based on the calculation of the set of reachable states for hybrid automata. These approaches suffer from the classical problems related to state space explosion and boundedness of considered variables.

- Proof-based approaches use deductive verification to prove the safety properties of hybrid systems. One of the strong points of these approaches is that they support large hybrid system specifications of any kinds, such as linear hybrid systems, non linear hybrid systems, etc. However, they require significant effort and a high expertise during the modeling and proof phases.

The definition of generic approaches, like that presented in this thesis using the Event-B formal method, for modeling and verifying hybrid systems may promote the use of proof-based approaches for industrial applications. The use of Event-B and its Rodin platform, a tool for Event-B project development, permits us to assist the developers in editing, checking but also proving correctness using the automatic and interactive provers included in the platform. In addition, interfacing a computer algebra system such as SageMath with an interactive theorem prover permits to deal with the resolution of ordinary differential equations when modeling hybrid system using a discrete formal method.

Contributions: In this thesis, we are interested in modeling and verifying the safety properties of a cyber-physical system. Our objective, as a part of the DISCONT project [9], is to develop formal approaches for modeling and verifying hybrid systems that combine discrete and continuous worlds. For this purpose, we have developed an approach to model and prove *Event-Triggered* systems and *Time-Triggered* systems in Event-B by taking advantage of its well-defined refinement process and its automatic/interactive provers used to verify the correctness of the models.

Since Event-B is designed for modeling discrete systems, it does not support the resolution of ordinary differential equations. To deal with this limit, we interface the Rodin tool with a differential equation solver, SageMath in our case, using the notion of plug-in. The main contributions of the present thesis are as follows:

- a generic formal proved approach for designing correct cyber-physical systems by considering any number of safety properties. This approach consists in defining three generic models in Event-B starting with an abstract model of cyber-physical systems and then using the refinement strategy to introduce more concrete details. These models are verified under Rodin using a set of theories introduced in [10]. This generic approach models and proves the relationship between *Event-Triggered* and *Time-Triggered* systems in Event-B. We reuse the approach proposed by Dupond et al. in [10] that defines a set of theories needed to model continuous aspects of CPSs in Event-B.

- a set of instantiation rules that are defined to systematically build the model of a specific application. These rules make it possible to deal with more complex safety properties (a conjunction of atomic ones) and make the approach more general. Moreover, we provide a set of generic invariants which have been identified from the case studies to prove the safety properties. They just need to be instantiated for proving any specific application.

- an extension of the generic approach to interface Event-B with the differential equation solver SageMath. A new generic model is defined. It refines the *Time-Triggered* model by introducing a function to model calls to the solver. A tool has been implemented as a new Rodin plug-in. This plug-in permits to call SageMath during the proof phase.

- a set of case studies to validate our approach. They have been chosen so that they represent different kinds of CPSs: hybrid systems with one or several continous variables, one or several safety properties, a non linear hybrid system.

Organisation of the Manuscript: The thesis is organised as follows:

Chapter 2: presents the context of the thesis by providing definitions of the main elements used in the development of our approach. In particular, it describes the mathematical model of cyber-physical systems, that of hybrid systems which specify both continuous and discrete dynamics of CPSs, the formal method Event-B used to model and prove our approach. In addition, the chapter presents the computer algebra system SageMath for the resolution of differential equations.

Chapter 3: presents the state of art of the most relevant formal approaches for cyber-physical systems modeling: model checking-based approaches, proof-based approaches. It also presents some approaches that integrate formal methods with computer algebra systems. In addition, the chapter discusses the main advantages and limitations of each approach which allows us to express the requirements needed to develop our approach.

Chapter 4: presents the case studies we have chosen to illustrate our generic approach. It describes their discrete and continuous behaviors using two different methods for modeling cyber-physical systems. These case studies are linear and admit exact solutions. While remaining simple, these case studies are didactic and quite representative of linear hybrid systems that admit exact solutions. The chapter also presents a nonlinear case study.

Chapter 5: introduces our main contribution, a correct-by-construction approach for modeling and verifying cyber-physical systems using Event-B. The approach proposes to model and prove a *Time-Triggered* system in Event-B by providing a link with an *Event-Triggered* system, as described by Kopetz. It uses the refinement strategy of Event-B to model the relation between these two systems.

Chapter 6: presents a set of rules for instantiating the generic approach following two different strategies. The first strategy consists in starting by an abstract model of the specification and then introducing more concrete details to instantiate *Event* and *Time-Triggered* models. The second strategy consists in directly modeling an *Event-Triggered* model and then introducing by refinement a *Time-Triggered* model.

Chapter 7: describes the interface between the differential equation solver SageMath and the Rodin tool to deal with the resolution of ordinary differential equations in Event-B. This is achieved by implementing a plugin to Rodin using the Eclipse platform which permits to add new plugins by providing a set of interfaces and predefined JAVA classes.

Chapter 8: applies our approach on a set of chosen case studies. The application is done by using the set of rules defined in Chapter 6. For each case study, we apply a different strategy to show the main differences between both strategies.

Chapter 9: gives a summary of our contributions, and discusses future work related to our work.

Chapter 2

Context

Contents

This chapter presents the main aspects relevant to modeling cyber-physical systems, presented in Section 2.1. The most common model for cyber-physical systems, that of hybrid systems, is presented in Section 2.2. Then, Section 2.3 presents ordinary differential equations used to specify the continuous part of hybrid systems. In Sections 2.4 and 2.5, we describe two well known approaches for modelling hybrid systems, hybrid automata which are a specific type of state machines used to model hybrid systems, and hybrid programs that represent the programming language for hybrid systems. The main notions of the EVENT-B formal method are described in Section 2.6. Last, Section 2.7 presents the computer algebra systems used to find analytical and numerical solutions of ordinary differential equations.

2.1 Cyber-Physical Systems (CPSs)

The context of this thesis is the domain of cyber-physical systems (CPSs) that integrate computation, networking, and physical processing. This type of system connects the discrete virtual world and the continuous physical world via a network of sensors and actuators. One of the most common architectures in cyber-physical systems is a separate software controller that represents the discrete part and controls the physical part through a loop with sensors and actuators as depicted by Figure 2.1. The term cyber-physical system first appeared in the mid-2000s, and the original definitions of these systems were provided by Edward A. Lee [2] as part of a collaboration with the National Science Foundation (NSF):

"*Cyber-Physical Systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. In the physical world, the passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions*". In other words, CPSs use computation and embedded communications to interact with physical processes to create new system functions.

Cyber-physical systems can be viewed as embedded systems. An embedded system is a combination of computer hardware and software that performs a specific task within the device in which it is integrated. In contrast to traditional embedded systems, which typically rely only on homogeneous communication structures, cyber-physical systems typically do not interact with individual devices, but rather interacting discrete systems with physical inputs and outputs. Cyber-physical systems are designed as a network of steering elements.

In addition, cyber-physical systems can communicate with other systems and exchange data with remote systems using Internet communication technology, a fundamental building block of the Internet of Things. The concept of cyber-physical systems therefore expands the definition of the Internet of Things, as networked devices which not only communicate with each other, but are also autonomous entities and must communicate and control each other.

The domain of cyber physical systems has emerged as an active domain of research in recent years that attracts the attention of many researchers which has given rise to a new category of system called Cyber-physical systems of systems (CPSoS) [11]. CPSoS involve a distributed and net-worked computing elements and human users that controlled a large physical elements. Therefore, the modeling and the verification of such complex systems requires the evolution of techniques and tools designed for standard CPSs such as presented in [12].

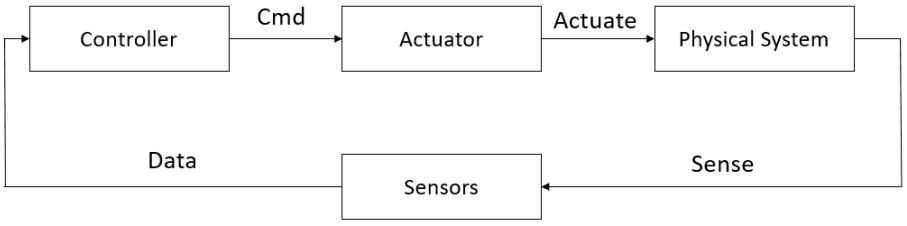

Figure 2.1: Cyber-Physical Systems Global Architecture.

A cyber physical system consists of many heterogeneous subsystems that interact to perform different functions of the system. The main components of a cyber physical system are the controller and the physical part. A cyber physical system may consist of a controller and a physical part, a single controller controlling a single plant, or a controller and multiple plants, a single controller controlling multiple plants, or multiple controllers and multiple plants. This thesis covers cyber physical case studies involving one controller and one plant as depicted by Figure 2.1, but this does not prevent the developed model from being used to cover other architectures.

2.2 Hybrid Systems

Hybrid systems [13] are frequently used to model CPS. They are dynamic systems that combine both flow described by ordinary differential equations and jumps described by state machines or automata. In general, the states of a hybrid system are defined by the values of continuous variables and discrete modes of the controller. States can change continuously according to the physical state or discretely according to the controller state. Continuous flow is allowed as long as the invariants hold, but discrete transitions can occur when

certain jump conditions are met. Formal modeling, verification, and overall design of hybrid systems are significant challenges. The development of techniques and tools for effectively designing and verifying hybrid systems has attracted the attention of many researchers [14]. Traditional approaches are based on simulation tools such as Matlab/Simulink and Stateflow, which are time consuming and yield results that are susceptible to uncertainty. To overcome these limitations, several formal approaches for hybrid system modeling have been proposed, we can quote: *Hybrid Automata* (Sec. 2.4) and *Hybrid Programs* (Sec. 2.5). We can also mention the approaches that combine formal design and verification of hybrid systems: *Hybrid CSP, Hybrid Hoare Logic* (Sec. 3.2.4) and *Hybrid* Event-B (Sec. 3.3.5).

An example of hybrid systems is that of a *bouncing ball*, a well-known hybrid system for controlling the motion of a rubber ball, as shown in Figure 2.2. This system consists in dropping a ball from a predefined height H. The ball undergoes elastic deformation, hits the ground, loses energy, bounces into the air and starts falling again. The continuous behavior of this system is modeled by the current position of the ball x and its current velocity v. This continuous behavior evolves according to two linear ordinary differential equations, $\frac{dx}{dt} = v(t)$ and $\frac{dv}{dt} = -g$, where g is the acceleration due to gravity. This system can be described by two different behaviors: *falling* and *bouncing*. The ball continues to fall as long as its current position x is greater than or equal to 0 ($x \geq 0$). A bounce state is achieved when the ball hits the ground ($x = 0$). In either state, the safety property, $0 \leq x(t) \leq H$, of the system must be satisfied to allow the ball never bounces higher than the initial height H.

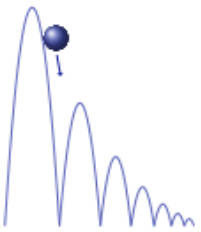

Figure 2.2: The Bouncing Ball

The Event-Time-Triggered Approach Besides the proposed formal approaches, Kopetz [3] 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 [3]. The *Event-Triggered* model represents an ideal behavior in which time is continuous and the sensors have continuous access to continuous measurements. The *Time-Triggered* model represents more specific behaviors where sensors take periodic measurements. Therefore, the controller of a *Time-Triggered* system must make choices that should be safe until the next sensors update. This makes proofs for these types of systems more complex compared to those of *Event-Triggered* systems.

2.3 Ordinary Differential Equations (ODEs)

The evolution of physical systems is often described by ordinary differential equations (ODEs) [15]. An ordinary differential equation of order n is the relationship between a single independent variable $x \in \mathbb{R}$, an unknown function y, and its derivative at a point

x. The most common form of the ODEs that describes the evolution of hybrid systems is: $a_n(x)y^n(x) + ... + a_2(x)y''(x) + a_1(x)y'(x) + a_0(x)y(x) = b(x)$. We distinguish two types of ordinary differential equations: linear and nonlinear ODEs. Linear ordinary differential equations are a special case of ODEs in which the unknown function and its derivative occur only in first order and do not multiply with each other. A linear ordinary differential equation is written in the following form (where y represents the dependent variable and x represents the independent variable):

$$a_n(x)y^n + ... + a_2(x)y'' + a_1(x)y' + a_0(x)y = b(x)$$

The theory of solving linear equations is very well developed because linear equations are simple enough to be solved. However, most physical systems are represented by nonlinear ordinary differential equations, which cannot usually be solved exactly, and are approximated by linear differential equations. There are two ways to solve ODEs: analytical (symbolic) methods and numerical methods. Analytical techniques use a set of theorems to obtain exact solutions (in the form of integrals) of certain differential equations. There are many computer algebra systems for solving ODEs, such as SageMath [6]. These solvers cover a wide range of mathematics, including algebra, calculus, number theory, formal linear algebra, and more. However, most differential equations cannot be solved exactly. Therefore, we must resort to numerical techniques to obtain approximate solutions, or use approximation techniques to convert the equations to another type of equivalent equations. For example, linearization techniques convert nonlinear ordinary differential equations to linear ordinary differential equations and apply linear differential equation analysis techniques to them.

A theorem can be used to prove whether a first-order ordinary differential equation, an ODE that only uses the first derivative y', has a solution, and whether its solution is unique. Note that any n^{th}-order ordinary differential equation can be transformed into a system of n first-order ODEs. The two main theorems are Peano's existence theorem and the Cauchy-Lipschitz (Picard-Lindelöf) theorem. Both assume the existence of initial conditions. An ordinary differential equation with initial conditions is called a Cauchy problem. Cauchy problem is made up of a first order ordinary differential equation for which we are looking for a solution satisfying given initial conditions.

2.4 Hybrid Automata

Hybrid automata are widely used as models for hybrid systems. They associate each discrete state with an ordinary differential equation that describes the evolution of a set of continuous variables over time, and an invariant that imposes additional properties on these continuous variables. The state of an automaton can change instantaneously by discrete transitions (mode changes) consisting of discrete steps, or by continuous activity (continuous variable changes without mode changes). In this thesis, we use a simple state transition system to represent hybrid automata such as used by Platzer in [16] to just have a graphic representation of hybrid programs described in the next section.

Figure 2.3 shows the hybrid automaton associated with the *Bouncing Ball* case study. The initial conditions are represented by the constants *xinit* and *vinit*. The states of the automaton are associated with differential equations describing the expansion of the ball state variables x and v. The expression, $0 \leq x(t)$, specifies the local invariant, which is the condition for the controller to react correctly at the right time.

As shown in Figure 2.3, the system can be in one of the following discrete states:

- *Init state*: initial values for position and velocity are represented by the constants *xinit* and *vinit* respectively. These constants should be chosen such that $xinit \geq 0$ and $vinit \geq 0$.
- *Falling state*: in this state, the ball falls with velocity v. This happens according to the positive weight g, $\frac{dv}{dt} = -g$. As soon as the formula $x(t) \geq 0$ is no longer satisfied,

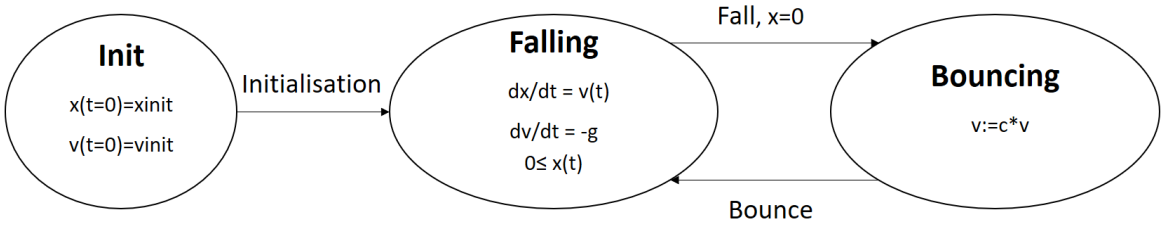

Figure 2.3: Hybrid Automaton of the Bouncing Ball System

the system should switch to the *Bouncing* state.

- *Bouncing state*: in this state, the ball position is 0. Therefore, the controller updates the ball velocity by $-c \times v$ to allow it to bounce again, where c represents the damping coefficient $(0 \leq c < 1)$.

Discrete dynamics are represented as transitions between states in a hybrid automaton. For example, if the expression $x(t) \geq 0$ is true, the ball can fall with $\frac{dx}{dt} = v$ (*Falling* state). If this formula is no longer valid i.e when $x = 0$, the ball must enter the *Bouncing* state if it is not already in it.

2.5 Hybrid Programs (HPs)

Hybrid Programs (HPs) [16] stand for programming languages for hybrid systems. They describe both discrete and continuous behavior of the hybrid system using sequential composition (;), non-deterministic choice ($\cup$), non-deterministic repetition (*), discrete assignments ($:=$), continuous evolution ($'$). Most hybrid programs are defined using the notation, $(ctrl; plant)^*$, where *ctrl* denotes the execution of the controller (discrete evolution), followed by the physical part *plant* (continuous evolution). This sequence is non-deterministically repeated, this is indicated by the star (*).

Formal Definition

A hybrid program is defined by the following grammar: where α, β are HPs, x a variable, θ is a term that may contain x, and F is a formula.

$$\alpha, \beta ::= \alpha; \beta | \alpha \cup \beta | \alpha^* | x := \theta | x := * | ?F$$

- $\alpha; \beta$: sequential synthesis, first performs α then β afterwards.
- $\alpha \cup \beta$: non-deterministic choice of α or β.
- α^*: non-deterministic iterations, repeating α $n \geq 0$ iterations.
- $x := \theta$: discrete assignment (jump) of the value of the term θ to the variable x.
- $x := *$: non-deterministic assignment to an arbitrary real number to x.
- $?F$: checks if the expression F is valid in the current state, abort if not.

Example of Hybrid Programs

Model 2.1: Bouncing Ball Hybrid Program

$$init \rightarrow [(ctrlV; plantV)^*](req) \quad \textbf{(2.1.1)}$$
$$init \equiv g > 0 \land x \geq 0 \land H \geq 0 \land c \geq 0 \land x < H \quad \textbf{(2.1.2)}$$
$$ctrl \equiv? \ (x = 0)(v := -c \times v) \quad \textbf{(2.1.3)}$$
$$plant \equiv (x' = v, v' = -g \ \& \ x \geq 0) \quad \textbf{(2.1.4)}$$
$$req \equiv 0 \leq x \leq H \quad \textbf{(2.1.5)}$$

Model 2.1 describes a hybrid program represented by (2.1.1) and related to the *Bouncing Ball* case study. The initial conditions are formally given by equation (2.1.2): weight g, position x and initial energy level H must be positive at the initial state. A related hybrid program is specified by sequential assembly of the controller (2.1.3) and the physical part (2.1.4). The control part uses the operator ? to update the velocity v when the ball hits the ground. The physical part uses the ODEs $(x' = v, v' = -g)$ to describe the continuous evolution of the system. In fact, hybrid programs model the notion of transitions between discrete states of a hybrid system by adding constraints to the system evolution space specified by the expression $x \leq 0$. Finally, equation (2.1.5) expresses the safety requirements of the system. This indicates that the ball position x never bounces higher than its initial height.

2.6 The Event-B Method

This thesis is aimed at modeling and verifying hybrid systems using the formal Event-B method and the Rodin platform. This method emerged as a further development of the classical B-method [17]. Introduced by J. Raymond Abrial [18], Event-B is a formal way to describe discrete systems in terms of events. The Event-B model brings with it a series of proof obligations (POs) aimed at verifying its correctness. Even if verifying specifications in Event-B, i.e. discharging the POs, is often hard, such a difficulty and complexity may depend on how the specification was built. In fact, in general, several solutions can be considered to model the same system. In this case, one criterion for choosing a particular solution is to minimize the complexity of the generated models.

Event-B is supported by the open-source and free Rodin platform, an Eclipse-based IDE, that allows modeling and verifying Event-B systems. Rodin makes it possible to create contexts, machines, generate proof obligations corresponding to properties, prove these proof obligations automatically or interactively, etc. It can also be coupled with tools for animating models such as ProB [19], which can be very useful to check if the specification produces the intended behaviors. New features can be added in Rodin as Eclipse plugins. For example, the Theory plug-in [20] is a Rodin extension that permits users to define their own new data types and operators.

2.6.1 Refinement

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 refinement to introduce details and demonstrate compliance between the abstract and the concrete models. The refinement of a formal model allows us to enrich this model step by step, using an incremental approach. Refinement is the foundation of the correction-by-construction paradigm.

There are two types of refinement in Event-B: horizontal refinement and vertical refinement. Horizontal refinement, also known as superposition, can be used to subsequently add complexity to the model. Such a refinement process makes it possible to add step by

step requirements from the specifications of the system to be modeled. Vertical refinement, also known as data refinement, makes it possible to refine a model resulting from a horizontal refinement process. Indeed, it allows a step-by-step implementation of the specification resulting from a horizontal refinement process. This type of refinement does not add functionality to the model, such as horizontal refinement, but it does refine the model to become closer to an executable model.

2.6.2 Modeling

The basic element of the development accomplished in the EVENT-B method is the model. The EVENT-B model consists of several components, *Context* and *Machine*. It can include a set of contexts and represents a purely mathematical structure consisting of sets, constants, axioms and theorems. An unparameterized model is composed only with machines. An EVENT-B model is parameterized by contexts if it is composed with both, contexts and machines. An EVENT-B context represents the static part of the EVENT-B model. Machine EVENT-B represents the dynamic behavior of the system. It can have access to one or more contexts.

2.6.2.1 Context EVENT-B

An EVENT-B context defines the mathematical structure associated with a system. Context can include support sets, constants, axioms, and theorems (see Figure 2.4). These elements are enclosed in clauses as shown in the list below.

```
CONTEXT Ctx
EXTENDS Ctx1, ...
SETS S, ...
CONSTANTS C, ...
AXIOMS A, ...
THEOREMS T, ...
END
```

Figure 2.4: Structure of an EVENT-B Context

- **EXTENDS Clause**: declares the context(s) extended by the described context.
- **CARRIER SETS Clause**: specifies a user-defined type. It describes a set of abstract and enumerated types.
- **CONSTANTS Clause**: represents the constants used in the model to set parameters for development.
- **AXIOMS Clause**: specifies types and restrictions of constants and carrier sets.
- **THEOREMS Clause**: describes the properties expected to be derived from the axioms.

2.6.2.2 Machine EVENT-B

An EVENT-B machine specifies the dynamic behavior of the modeled system. A machine models a system using state variables and a sequence of events that update those variables. It consists primarily of a set of modeling elements that mainly define state variables, invariants and events (see Figure 2.5).

- **REFINES Clause**: declares the machine refined by the described machine.

```
MACHINE M
REFINES M1
SEES Ctx, ...
VARIABLES v, ...
INVARIANTS I(v)
EVENTS
INITIALISATION BAP(v,v'), ...
...
END
```

Figure 2.5: Structure of an EVENT-B Machine

- **SEES Clause**: declares the contexts seen by the machine being described. This clause permits to get access to elements defined in the contexts during the modeling and the proof phase.
- **VARIABLES Clause**: declares variables for the modeled system. New variables can be introduced using the refinement strategy in order to enrich the modeled system.
- **INVARIANT Clause**: describes the properties of the state variables defined in the *VARIABLES* clause and the properties of the system being modeled. These properties shall be preserved by the initialisation and events.
- **INITIALISATION Clause**: allows giving initial values to the variables of the corresponding clause. They define the initial states of the underlying treated system.

EVENTS Clause

The *EVENTS* clause defines all the events that may occur in a given model. The structure of an EVENT-B event is shown by Figure 2.6. An EVENT-B event is triggered when the properties specified in the *WHERE* clause evaluate to true. Any event in EVENT-B models a discrete transition and can be defined by a *before-after* predicate denoted $BAP(v, v')$, where v and v' respectively denote the value of the variables before and after the execution of the actions associated with the event.

```
EVENT E
REFINES E1
ANY p, ...
WHERE G(v,p), ...
WITH w,..
THEN act,..
END
```

Figure 2.6: Structure of an EVENT-B EVENT

- **REFINES Clause**: lists the abstract event (s) that the current event refines (if any).
- **ANY Clause**: lists the parameters of the event.
- **WHERE Clause**: contains the various event guards. These guards are necessary conditions to trigger the event. This clause specifies also the types of the parameters.
- **WITH Clause**: when a parameter in an abstract event disappears in the concrete version of that event, it is essential to define a witness on the existence of this parameter.

- **THEN Clause**: describes the list of actions of the event.

Each event is made up of one or more so-called substitution actions. EVENT-B offers two kinds of substitution actions:

- Deterministic action: is expressed using the operator :=. It is made of a variable identifier *var*, followed by :=, followed by an expression *exp*.

$$var := exp$$

- Non-deterministic action: is expressed using the operator : | and a before-after predicate which specifies the corresponding value just before the action takes place. It is made of a variable identifier *var*, followed by : |, followed by a before-after predicate characterized by the symbol $'$, var'.

$$var :| var'$$

  A special case of non-deterministic actions is expressed using the symbol :∈. It is made of a variable identifier *var*, followed by :∈, followed by a set expression S.

$$var :\in S \equiv var :| var' \in S$$

2.6.2.3 Example

We illustrate the use of the elements presented in this section on a simple example. The goal is to model the discrete states of the behavior of a car. We model the evolution of the discrete state of the car which is represented by the evolution of its acceleration a. Figure 2.7 gives the context for this development. The system can be into three states: *Accelerating*, *Braking* and *Stopped*. These states are modeled using an abstract carrier set named $STATE$. The context *Car_Ctx* contains all the parameters of the modeled system and their properties such as the maximum of braking A and the maximum limit of braking B that must be positive. *Car_Ctx* defines also the constant ai that represents the initial value of the acceleration of the car and which must be defined between $-B$ and A.

```
CONTEXT Car_Ctx
STATES STATE
CONSTANTS Accelerating , Braking , Stopped , A, B, ai
AXIOMS
axm1: partition(STATE {Accelerating},{Braking},{Stopped})
axm2: A ∈ ℕ
axm3: B ∈ ℕ
axm4: ai ∈ ℤ ∧ ai ≤ A ∧ ai ≥ −B
END
```

Figure 2.7: Context Car_Ctx

The *Car_M* machine (see Figure 2.8) sees the context *Car_Ctx* and uses its constants and axioms. It defines two variables, variable a representing the acceleration of the car and variable *state* representing the current state of the system. The value assigned to the variable a must be defined in the interval $[-B, A]$ given by the invariant $inv3$. Initially, the value of a is ai that is defined in $[-B, A]$. The value of a is updated according to the current value of the variable *state*, so three events are defined: *Accelerate*, *Brake* and *Stop*.

```
MACHINE Car_M
SEES Car_Ctx
VARIABLES a, state
INVARIANTS
 inv1: a ∈ ℤ
 inv2: state ∈ STATE
 inv3: a ≤ A ∧ a ≥ −B
EVENTS
 INITIALISATION
 THEN
  act1:   a:=ai
  act2:   state:=Accelerating
 END

 Accelerate
 WHERE
  grd1: state= Accelerating
 THEN
  act1: a:= A
  act2: state:= Braking
 END
 Brake
 WHERE
  grd1: state= Braking
 THEN
  act1: a:= −B
  act2: state:= Stopped
 END

 Stop
 WHERE
  grd1: state= Stopped
 THEN
  act1: a:=0
  act2: state:= Accelerating
 END
```

Figure 2.8: Machine Car_M

2.6.3 Proof obligations

To ensure the correctness of an Event-B machine, a set of proof obligations (POs) are generated. These POs fall into two categories:

- *event feasibility*: for each event, of the form (**ANY** X **WHERE** G **THEN** Act **END**), we have to prove that there does exist at least a value for X that verifies G: $\forall S, C.\ (A \wedge Inv \Rightarrow \exists\ X.G)$.
- *event correctness*: we have to establish that the invariant Inv is fulfilled after the initialisation $INITIALISATION$ and that each event, of the form (**ANY** X **WHERE** G **THEN** Act **END**), re-establishes the invariant:

$$[INITIALISATION]\ Inv$$
$$\forall(S, C, V, X).\ (A \wedge G \wedge Inv \Rightarrow [Act]Inv)$$

The expression $[Act]Inv$ denotes the actions Act applied as a substitution to the formula Inv; it denotes the weakest constraint on the before state such that the execution of Act leads to an after state satisfying Inv.

2.7 Computer Algebra Systems (CASs)

A computer algebra system (CAS) [21] is a mathematical software that provides the ability to manipulate mathematical formulas in a manner similar to the traditional manual calculation of mathematicians and scientists. A key part of this system is the manipulation of mathematical formulas in symbolic form. It can handle literal expressions and use symbolic arithmetic where possible to perform exact calculations. Symbolic or formal calculus consists in making a computer algebra system perform exact mathematical calculations (expansion, transformation, simplification of expressions). It is typical of algebra that symbols in formulas are not necessarily replaced by specific numerical values, but are retained in the process of calculation. They are many tools that support formal calculations such as: *Mathematica* [22] , SageMath [6], *Maple* [23], *Matlab* [24] etc.

SageMath (System for Algebra and Geometry Experimentation) [6] is a free computer algebra system that combines the functionalities of many free programs into a common Python based interface. Its main goal is to create a free and open source alternative to Magma [25], Maple [26], Mathematica [22] and Matlab [24]. As such, it uses several existing open source libraries from other projects. It has two modes of use: notebook mode and command line mode. SageMath covers a wide range of mathematics, including algebra, calculus, number theory, cryptography, numerical calculus, commutative algebra, group theory, graph theory, and formal linear algebra.

Analytical Solutions in SageMath

To find the symbolic solution of a given ordinary differential equation, SageMath provides a function called *desolve()*. It computes general solutions to ordinary first- or second-order differential equations via *Maxima*, which is a system for the manipulation of symbolic and numerical expressions, including differentiation, integration, Taylor series, Laplace transforms, ODEs, systems of linear equations, polynomials, sets, lists, vectors, matrices and tensors. *desolve()* is defined by:

$$desolve(de, dvar, ics, ivar, show\_method)$$

- *de*: represents a differential equation.
- *dvar*: represents the unknown function (dependent variable).
- *ics*: represents an optional argument used to specify initial conditions. For linear equations, specify the list [$x0$, $y0$].
- *ivar*: represents an optional argument that specifies the independent variable.
- *show_method*: is an optional argument, by default set to False. Otherwise, SageMath will ask for the solution method to be used.

Numerical Solutions in SageMath

SageMath includes several functions that allow finding approximate solutions for first-order ODEs using numerical methods such as the Runge-Kutta methods, including the well-known method called the Euler Method, represented in SageMath by the *euler_method*() function. It provides also the 4th order Runge-Kutta method represented in SageMath by the *desolve_rk4*() function. In our development, we decided to implement the result of *desolve_rk4* in Event-B, since it is more complete and easier to use than the *euler_method*() function.

Function desolve_rk4 numerically solves first-order ordinary differential equations using the 4th order Runge-Kutta numerical method. The method is based on the principle of iteration. It uses the first guess of the solution to compute the second, etc. For example, using the value of the continuous variable y at time $t0$, at time $t1$ calculate the next value of such. The signature of *desolve_rk4* is:

$$desolve\_rk4(equation, variable, ics, [options : ivar, end - points, step, output])$$

- *equation*: this argument is either the right-hand side of the equation or the complete symbolic equation. For example, consider the first-order ordinary differential equation $y' = ycos(x) + ysin(x)$, where y is the dependent variable and x is the independent variable. In this case the argument *equation* is either $y * cos(x) + y * sin(x)$ or the full symbolic equation $diff(y, x, 1) == y * cos(x) + y * sin(x)$ where $diff$ will be used to

represent the left-hand side of an ordinary differential equation in SageMath and takes as parameters the unknown function y, the independent variable x, and the order of the ordinary differential equation 1.

- *variable*: this argument represents the dependent variable y. The dependent variable y should be defined as the symbolic equation $diff(y,t) == ycos(x)+ysin(x)$, otherwise $var('y')$.
- *ics*: this argument is given as a list of the initial conditions for the independent variable x and the dependent variable y, in the order $ics = [x0, y0]$. If the equation uses discrete variables a, b, their initial conditions are given in the order in which they are declared in SageMath: $[x_0, y_0, a_0...b_0]$ where a_0 and b_0 represent the initial values of a and b.
- *ivar*: this argument represents the independent variable of a given ODE. It is an optional argument that takes *None* as a default value.
- *end_points*: this argument represents the lower and upper bounds for the interval over which the numerical values of the dependent variable y are calculated. It is an optional argument that takes as a default value, $end\_points = ics[0] + 10$. For example, if $ics[0] = 30$, *desolve_rk4* returns the values of y between 30 and 40. Otherwise, this argument can take the following values:
  - *val*: integrate between $min(ics[0], val)$ and $max(ics[0], val)$, where $ics[0]$ represents the initial value of the independent variable x at time 0. For example, if we set $val = 20$ and $ics[0] = 5$, *desolve_rk4* will return y values between 5 and 20.
  - $[m, M]$: integrate between $min(ics[0], m)$ and $max(ics[0], M)$. If we consider $m = 5$ and $M = 20$ and $ics[0] = 0$, *desolve_rk4* returns the values of y between $min(ics[0], m) = 0$ and $max(ics[0], M) = 20$.

2.8 Conclusion

In this chapter, we introduced the concepts of cyber-physical systems and hybrid systems, different kinds of differential equations used to model the continuous behavior of cyber-physical systems, and the Event-B formal method. Then we presented two classical formal models for hybrid systems, namely hybrid automata and hybrid programs. To verify that an hybrid system satisfies its properties, the relevant differential equations need to be solved, computer algebra system such as SageMath can then be used. The next chapter is devoted to the presentation of formal approaches used to verify the correctness of hybrid systems.

Chapter 3

State Of Art

Contents

Hybrid systems are often safety critical systems. A fundamental step in the design of these systems is their modeling and verification. 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.

This chapter presents the current state of formal approaches that have been developed for the design and verification of hybrid systems. Section 3.1 gives an overview of some model checking-based approaches. Proof-based approaches to specifying and verifying the continuous part of CPSs using differential equations are described in Section 3.2. Section 3.3 focuses on the Event-B-based approaches. Section 3.4 presents an approach to integrate theorem provers with computer algebra systems to prove the safety properties of CPSs. Finally, Section 3.5 describes issues that can arise when using each approach.

3.1 Model Checking-Based Approaches

Model checking-based approaches, also known as algorithmic approaches, require constructing a finite transitions system through a discrete abstraction such as a hybrid automaton. These approaches are based on computing a set of reachable states to automatically verify that the system satisfies a set of expected properties. Depending on the nature of the hybrid system to be dealt with, various approaches have been proposed. For linear hybrid systems, the reachability determination is decidable, then tools such as $HyTech$ [27], $PHaVer$ [28], d/dt [29] or $SpaceEx$ [30] are used. Since the reachability of nonlinear systems cannot be determined, tools such as $Flow^*$ [31] or $iSAT$ [32] $dReal/dReach$ [33] use bounded model checking for reachability analysis to prove safety properties on these systems.

3.1.1 Hytech

HyTech [27] is an automatic and symbolic model checker for hybrid linear automata [34]. *Hytech* is the first model checker to implement reachability analysis for hybrid linear automata. A key feature of *HyTech* is its ability to perform parametric analysis, i.e. to determine the values of the design parameters that will allow the hybrid linear state machine to meet the timing requirements. Hybrid systems in *Hytech* are specified as a collection of discrete and continuous component automata, and timing requirements are verified by symbolic model checking. If the verification fails, *HyTech* will generate a diagnostic error trace.

3.1.2 SpaceEx

The *SpaceEx* [30] platform implements the reachability and safety verification algorithms for linear hybrid systems. It is a successor to *PHAVer* [28] for computing reachable states set of continuous and hybrid systems. This solves the main problem of *PHAVer*, which abstracts linear continuous dynamics by a constant domain associated with the derivative, and poses a scalability problem due to the large number of domains required. *SpaceEx* enhances tools for verifying existing hybrid systems and consists of three components: a command line program, a powerful analysis kernel, a configuration file that defines initial conditions, and other options. Among the accessibility computation algorithms implemented in this platform, we can mention scalable reachability algorithms. *SpaceEx* combines a polyhedral representation with a continuous set of support functions to compute an over-approximation of the states reachable by the system.

3.1.3 FLOW*

*FLOW** [31] is a verification tool for nonlinear hybrid systems. It focuses on reachability-based verification of hybrid automata. Reachability problems cannot be determined in a hybrid automata, so the tool computes an over-approximation of the reachable states set. Approximation theorems are presented as a finite set of Taylor models [35]. These models support functional operations such as addition, multiplication, division and derivation.

3.1.4 dReach/dReal

dReach [33] is a tool for verifying the safety of hybrid systems with nonlinear continuous dynamics. It can handle general hybrid systems containing nonlinear differential equations. This tool is based on its SMT solver *dReal* [25] for nonlinear theory on real numbers. *dReal* handles problems involving a wide range of real nonlinear functions such as polynomials, exponentials, etc.

3.1.5 Discussion

An hybrid model checker depends on the type of the hybrid automaton it handles, its dynamics, and the properties of its guards and invariants. Most model checking-based approaches are either limited to verifiable properties or to simplified classes of systems. Unfortunately, the problem of model checking is computationally very difficult. Moreover, as already mentioned, model checking-based approaches suffer from classical problems related to state-space explosion of the variables considered. Unfortunately, this problem is computationally very difficult. In fact, this problem cannot be solved even with a simple property or system.

3.2 Proof-Based Approaches

This section introduces proof-based approaches that can handle differential equations in hybrid systems modeling such as d$\mathcal{L}$ and its extension dR$\mathcal{L}$, first order differential logics supported by the theorem prover KeYmaera [36] and its successor KeYmaera X [37].

3.2.1 Differential Dynamic Logic d$\mathcal{L}$

This section describes the real-domain ($\mathbb{R}$) first-order differential dynamic logic, introduced by A. Platzer, to express safety and liveness properties of hybrid systems, and its related proof calculus used to determine their exactness. d$\mathcal{L}$ formulas are built using logical symbols of first-order logic and the modalities [ ] (box modality) and (diamond modality) [38] according to the following grammar (where α is a hybrid program (HP); φ, ψ, θ_1 and θ_2 are formulas and x is a variable):

$$[\alpha]\varphi \mid \langle\alpha\rangle\varphi \mid \varphi ::= \theta_1 \sim \theta_2 \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid \forall x\varphi \mid \exists x\varphi$$

- $[\alpha]\varphi$: is true if after each run of HP α, formula φ holds.
- $\langle\alpha\rangle\varphi$: is true if φ is true after at least one run of HP α.
- $\varphi ::= \theta_1 \sim \theta_2$: is true *iff* $\theta_1 \sim \theta_2$ is true with $\sim \in \{=, >, \geq, <, \leq\}$.
- $\neg\varphi$: is true if φ is false.
- $\varphi \wedge \psi$: is true if both φ and ψ are true.

- $\varphi \vee \psi$: is true if φ is true or ψ is true.
- $\varphi \rightarrow \psi$: is true if φ is false or ψ is true.
- $\varphi \leftrightarrow \psi$: is true if φ and ψ are both true or both false.
- $\forall x \varphi$: is true if φ is true for all values of variable x.
- $\exists x \varphi$: is true if φ is true for some values of variable x.

To prove the safety of hybrid systems, d$\mathcal{L}$ provides a proof calculus supported by two formal verification tools, KeYmaera and its successor KeYmaera X. The main advantage of d$\mathcal{L}$ is its ability to handle differential equations with non-polynomial solutions. When a system of differential equations does not have a polynomial solution, many mechanisms of differential induction (induction of differential equations) are available, such as differential invariants and differential cuts. On the other hand, if the solution is polynomial, it can be easily replaced with a discrete assignment at any point once the time variable is introduced. Finally, in order to establish a safety property for a system, *safeReq*, a formula expressing safety relative to initial conditions needs to be proved, $\mathit{init} \rightarrow [(\mathit{ctrl}; \mathit{plant})^*](\mathit{safeReq})$ that means: if the initial conditions (*init*) hold, then, after each run of the hybrid program, *safeReq* remains true.

KeYmaera is an automated/interactive formal verification tool for hybrid systems that supports d$\mathcal{L}$ and its associated proof calculus. KeYmaera is a combination of deductive, real algebraic and computational algebraic proof techniques. To automate the verification process, KeYmaera implements an auto-proof strategy that symbolically decomposes hybrid system specifications using d$\mathcal{L}$ proof rules defined in a rule base. KeYmaera interacts with several solvers such as Mathematica and Orbital, a Java math library. KeYmaera uses both tools to obtain symbolic solutions of differential equations that can be used to treat continuous dynamics. In addition, KeYmaera provides a set of proof rules for handling differential induction mechanisms, allowing differential equations with non-polynomial solutions to be treated. The most interesting mechanism is the differential invariant, which provides induction of differential equations. The main advantage of such invariants is the ability to prove properties of differential equations without solving the equations. To prove the safety properties of hybrid systems using KeYmaera, we must define a KeYmaera file which specifies both, the model of the hybrid system and the correctness property to verify. KeYmaera files are defined by the following syntax:

- $\alpha ::= \alpha; \beta$: the symbol ; specifies a sequential composition of the hybrid programs α and β. It executes all the instructions of α and then executes the program β.
- $x := t$: the symbol := specifies a discrete assignment that assigns the value of term t to x.
- $x := *$: assigns non-deterministically any real to x.
- $?H$: the symbol ? checks if the formula H is true or not.
- $\alpha + +\beta$: represents a non-deterministic choice between the hyprid programs α and β.
- α^*: it non-deterministically repeats the hybrid program α.
- $\{x' = t, y' = s, H\}$: specifies the differential equations that describe the continuous behaviors of the state variables x and y, with evolution domain constraint H which needs to be true during the evolution.

KeYmaera X is the successor to the KeYmaera tool and also supports differential dynamic logic and hybrid programs. The most important feature of KeYmaera X is the ability to allow users to specify custom proof search techniques as tactics using its proof programming language called Bellerophon. A tactic is a program that combines a set of proof rules to define a proof discovery strategy.

3.2.2 Differential Refinement Logic dR$\mathcal{L}$

dR$\mathcal{L}$ [5] is a generalized refinement computation of d$\mathcal{L}$. dR$\mathcal{L}$ extends d$\mathcal{L}$ by introducing hyprid programs refinement operator ($\leq$). In addition to the d$\mathcal{L}$ expressions, dR$\mathcal{L}$ defines expressions of the form $\alpha \leq \beta$, read α refines β, with α and β denoting HPs. According to [5], the expression $\alpha \leq \beta$ is true in state s if and only if all states reachable from s by following transitions from α are also reachable from state s by following transitions from β.

dR$\mathcal{L}$ preserves the safety properties of refined hybrid programs by showing that if $\alpha \leq \beta$ and $[\beta]\varphi$, then the formula φ is true in all states reachable from s by following the transitions of α ($[\alpha]\varphi$). There is a similar rule for diamond modalities ($\langle\ \rangle$), which states that if α refines β, and there is at least one transition from α to a state s where φ is true, then $\langle\beta\rangle\varphi$ is true. This is represented by the following two proof rules:

$$\frac{\Gamma \vdash [\alpha]\varphi, \Delta \quad \Gamma \vdash \beta \leq \alpha, \Delta}{\Gamma \vdash [\beta]\varphi, \Delta}([\leq])$$

$$\frac{\Gamma \vdash \langle\alpha\rangle\varphi, \Delta \quad \Gamma \vdash \beta \leq \alpha, \Delta}{\Gamma \vdash \langle\beta\rangle\varphi, \Delta}(\langle\leq\rangle)$$

A proof calculus associated with a logical language such as d$\mathcal{L}$ or dR$\mathcal{L}$ is a set of syntactic transformations that are each proved sound. By combining many of these transformations on a complicated formula, we may simplify and break apart the formula until we are left with formulas that can be proved true using quantifier elimination, in which case we have a proof of our original complicated formula. The proof calculs for dR$\mathcal{L}$ is composed of three types of proof rules:

- Proof rules based on the axioms of the KAT algebra [39].

  $$\frac{\Gamma \vdash [\alpha^*](\alpha;\gamma) \leq \gamma, \Delta \quad \Gamma \vdash [\alpha^*]\beta \leq \gamma, \Delta}{\Gamma \vdash \alpha^*;\beta \leq \gamma, \Delta}(loop_l)$$

  $$\frac{\Gamma \vdash \beta \leq \gamma, \Delta \quad \Gamma \vdash (\gamma;\alpha) \leq \gamma, \Delta}{\Gamma \vdash \beta;\alpha^* \leq \gamma, \Delta}(loop_r)$$

  these two proof rules are used to handle loops in a refinement proof. When we have $\alpha^*;\beta$, we add a $[\alpha^*]$ to both the left and right premise, in contrast to $loop_r$ that does not require modalities in the premises.

- Structural proof rules, which exploit the similarities between hybrid programs.

  $$\frac{\Gamma \vdash \alpha_1 \leq \alpha_2, \Delta \quad \Gamma \vdash [\alpha_1](\beta_1 \leq \beta_2), \Delta}{\Gamma \vdash (\alpha_1;\beta_1) \leq (\alpha_2;\beta_2), \Delta}(;)$$

  this proof rule is used to handle the refinement of sequential hybrid programs.

- Proof rules to handle the refinement of differential equations:

$$\frac{\Gamma \vdash \forall x(H_1 \rightarrow H_2), \Delta}{\Gamma \vdash (x' = \theta \ \& \ H_1) \leq (x' = \theta \ \& \ H_2), \Delta}(DR)$$

  this proof rule shows that if two differential equations differ only in their evolution domain, then a refinement relationship can be only satisfied if the evolution domain of the smaller program is subset of the evolution domain of the larger program.

3.2.2.1 Event and Time-Triggered Systems in dR$\mathcal{L}$

dR$\mathcal{L}$ introduces two generic templates [5], *Model 3.1* and *Model 3.2*, to design and prove *Event-* and *Time-Triggered* systems presented in Section 2.2. The control part of these two generic templates has only two modes: the *normal* mode is triggered when the system safety envelope, denoted by *safe*, is satisfied, otherwise the system enters into the *evade* mode. Note that the operator $\sim$ is used to designate the topological closure of the negation. *Model 3.1* represents the generic model associated with a controller triggered by events:

Event-Triggered Model

Model 3.1: Event-triggered Generic Model

$$event^* \equiv (ctrl_{Ev}; plant_{Ev})^* \quad \textbf{(3.1.1)}$$

$$ctrl_{Ev} \equiv (ctrlV := evade\_value) \cup (ctrlV := *; ?safe(plantV)) \quad \textbf{(3.1.2)}$$

$$plant_{Ev} \equiv t := 0; \ plantV_0 := plantV; (plantV' = f\_evol(ctrlV), t' = 1 \& \ evt\_trig(plantV) \wedge dom\_evol(plantV)) \quad \textbf{(3.1.3)}$$

$$\cup (plantV' = f\_evol(ctrlV), t' = 1 \ \& \ \sim evt\_trig(plantV) \wedge dom\_evol(plantV)) \quad \textbf{(3.1.4)}$$

Where:

- $ctrlV$: the control variable (acceleration in the case of a car).
- $plantV$: the state variable of the system (position and velocity in the case of a car).
- $plantV'$: represents the evolution of $plantV$ over time $d(plantV)/dt$. Mention that in d$\mathcal{L}$, the notation $variable'$ denotes a derivative while, in Event-B, it is used to denote the new value of $variable$ after triggering the event that updates it.
- $safe(plantV)$: defines the system safety envelope. It is calculated from the safety requirement that the system must satisfy.
- $plantV' = f\_evol(ctrlV)$: defines the ordinary differential equation that describes the continuous evolution of the system.
- $evt\_trig(plantV)$: the predicate that defines the boundary of the safety envelope. When the system becomes closed to this boundary, the controller triggers the *evade* mode. It must define a closed domain.
- $\sim evt\_trig(plantV)$: topological closure of the complement of evt_trig.
- $dom\_evol(plantV)$: defines the evolution domain of the system. It is a set of constraints on the state variable. For example, the evolution domain of the *Bouncing Ball* case study is $x \geq 0$ which denotes that the car is not allowed to back down.

- $plantV_0$: represents the initial value of $plantV$.

When the formula *safe* is satisfied, the system can evolve continuously according to the formula **(3.1.3)** until it reaches the boundary of the domain $evt\_trig(plantV)$. In that case, the controller must then switch to the *evade* mode by affecting a deterministic value $evade\_value$ to the control variable ($ctrlV$), it will behave according to the formula **(3.1.4)**. Note that the variables t and $plantV_0$ have no effect on the state of this model; they will be used in the second model. To prove the safety of this model, dR$\mathcal{L}$ provides the following proof obligation where ζ is the context of the system that may contain any property needed to facilitate the proof:

$$evt\_trig(plantV) \wedge \zeta \vdash [event](evt\_trig(plantV) \wedge \zeta)$$

This proof obligation states that *Model 3.1* is safe if its associated hybrid program *event* always satisfies the loop invariant $evt\_trig(plantV)$ and ζ.

Time-Triggered Model

Model 3.2: Time-triggered Generic Model

$$time^* \equiv (ctrl_t; plant_t)^* \quad \textbf{(3.2.1)}$$

$$ctrl_t \equiv (ctrlV := evade\_value) \cup (ctrlV := *; ?safe_\epsilon(plantV, ctrlV)) \quad \textbf{(3.2.2)}$$

$$plant_t \equiv t := 0;\ plantV_0 := plantV; (plantV' = f\_evol(ctrlV), t' = 1\ \&\ t \leq \epsilon \wedge dom\_evol(plantV)) \quad \textbf{(3.2.3)}$$

where

ϵ: maximum time between two sensor updates.
t: allows to know if the duration ϵ is reached or not.

Model 3.2 represents the generic model associated to a *Time-Triggered* system. The controller of such system reacts at least every ϵ seconds, where the formulas **(3.1.3)** and **(3.1.4)** are replaced by the formula **(3.2.3)**. Formula *safe* is also replaced by formula $safe_\epsilon$, which depends on both the current choice of $ctrlV$ and the time duration ϵ, in addition to the current state $plantV$, in order to guarantee that the controller will make a choice that will be safe for up to ϵ time. To prove that *Model 3.2* satisfies a safety property φ, dR$\mathcal{L}$ has introduced the following proof obligation ([$\leq$]) where Δ denotes a set of formulas like invariant properties.

$$\frac{\zeta \vdash [event^*]\varphi, \Delta \quad \zeta \vdash (time^* \leq event^*), \Delta}{\zeta \vdash [time^*]\varphi, \Delta}[\leq]$$

This proof obligation consists of two sub-proof obligations: the first one proves that *Model 3.1* satisfies the system safety property φ, and the second one aims at verifying that *Model 3.2* refines *Model 3.1*.

3.2.2.2 Time-Triggered Model Refines Event-Triggered Model

To prove that a *Time-Triggered* system refines an *Event-Triggered* system, dR$\mathcal{L}$ provides three proof obligations:

- **PO1_dR$\mathcal{L}$** : $evt\_trig(plantV) \wedge \zeta \wedge safe_\epsilon(plantV, \bar{ctrlV}) \vdash safe(plantV)$

where: $\bar{ctrlV}$: represents a non-deterministic choice of the control variable. This proof obligation expresses that the safety envelope of *Model 3.2* implies that of *Model 3.1*, which means that the discrete controller refines the continuous one.

- **PO2_dR$\mathcal{L}$:** $\bar{evt\_trig(plantV_0)} \wedge \zeta \wedge \bar{safe_\epsilon(plantV_0, ctrlV)} \wedge 0 \leq t \leq \epsilon \wedge \bar{dom\_evol(plantV)} \wedge \bar{plantV} = S_{\bar{plantV_0}, \bar{ctrlV}}(t) \vdash \bar{evt\_trig(plantV)}$

  where:
  $\bar{plantV_0}$: set of physical state variables values at instant $t = 0$.
  $\bar{plantV}$: set of physical state variables values at instant t.
  $S_{\bar{plantV_0}, \bar{ctrlV}}(t)$: solutions of the ordinary differential equation associated with $plantV_0$, given $\bar{ctrlV}$.

  This proof obligation expresses that the non-deterministic choice of $ctrlV := *$ expressed by $\bar{ctrlV}$ guarantees that the system will not cross the boundary of $evt\_trig(plantV)$ within time ϵ.

- **PO3_dR$\mathcal{L}$:** $\bar{evt\_trig(plantV_0)} \wedge \zeta \wedge 0 \leq t \leq \epsilon \wedge \bar{dom\_evol(plantV)} \wedge \bar{plantV} = S_{\bar{plantV_0}, \bar{evade\_value}}(t) \vdash \bar{evt\_trig(plantV)}$

  This proof obligation is similar to the previous one for the *evade* mode: $ctrlV := evade\_value$.

It is worth noting that such proof obligations are achieved on the instantiated models since they cannot be discharged on the generic ones without having concrete expressions for the different formulae. The major limitation of dR$\mathcal{L}$ is that it is not supported by any prover, thus these proof obligations are manually discharged. This limitation represents a strong restriction on its application to more complex hybrid systems since a such proof activity is very tedious and error-prone.

3.2.3 Parallelism and Modular Proof in Differential Dynamic Logic

Refinement allows building hybrid systems gradually by starting with an abstract system easy to understand and verify until we get the concrete system. This development approach simplifies the challenge of large-scale verification of hybrid systems. There is another method called the component-based method that simplifies the complexity of hybrid systems. Basically, the hybrid system is disassembled into parts (components) to easily check the safety, and these components are assembled using the assembly mechanism to form the entire system. Among the approaches proposed to support this method, we can cite the one presented in [40]. It is based on differential dynamic logic and defines parallel composition operator $\circ$ for building a system from its parts. This operator is the first composition operator in differential dynamic logic that is modular, commutative and associative.

The author of [40] introduces two parallel composition operators. The first is called the parallel continuous compositing operator and is denoted by $\circ_c$. This is defined as the purely continuous behavior of the component, i.e. ODEs to construct. As a reminder, the continuous part of a hybrid system is represented by an ordinary differential equation of the form $y' = \theta_y \,\&\, dom\_evol_y$ where y is a vector of state variables $(y_1, ..., y_n)$, presented in the previous sections by the variable $plantV$, and θ_x is a vector $(\theta_1, ..., \theta_n)$ of terms of real arithmetic, presented in the previous sections by the term $f\_evol(ctrlV)$. The second operator is used to compose behaviors that combine both continuous and discrete parts. To facilitate the proof process, the approach extends the proof system of d$\mathcal{L}$ in order to be able to decompose the proof of the global system. In Event-B, we can decompose a

complex system into multiple sub-systems but there is no mechanism to recompose these sub-systems. Composing sub-systems in Event-B can be an interesting subject of research for future work.

3.2.4 Hybrid CSP and Hybrid Chi

The formal language Hybrid Communicating Sequential Processes (HCSP) [41,42] is an extension of Communicating Sequential Processes (CSP) [43] that allows modeling the sequential dynamics of hybrid systems. With support for continuous variables and differential equations, HCSP can be used to model real-time and continuous behaviors in message-based communications. The approach developed in [44] verifies the safety of HCSP processes by using differential invariants to reason about differential equations and using logic to handle communication, parallelism interruptions, timing, etc. Moreover, Hybrid Hoare Logic [45] is supported by an interactive theorem prover based on Isabelle/HOL that allows verification of HCSP models. A set of refinement rules is defined to refine the HCSP abstract specification to lower-level implementations. The work presented in [46] proposes another hybrid extension of CSP, a new formal language called Hybrid Chi. It integrates the concepts of dynamics and control theory with those of computer science, especially those of process algebra and hybrid automata. The HCSP approach differs from Hybrid Chi in that it does not share common variables essential for modular specification of continuous and hybrid systems. With Hybrid CSP and Hybrid Chi, formal methods can be applied to continuous processes.

3.2.5 Modeling and Verifying Hybrid Systems with Isabelle/HOL

The authors of [47] present a new proof-based approach that introduces a new differential Hoare logic dH and a new differential refinement calculus dR using the higher-order logic proof assistant Isabelle/HOL. Using this approach, a complex property, like ordinary differential equation liveness or program correctness, should be modeled using dH, broken down into (simpler) step-by-step refinements using dR and proved in Isabelle. The differential Hoare Logics dH implements the differential dynamic logic in Isabelle by simply adding a single Hoare-style axiom. The approach uses the Kleene algebras [48] with tests and the Morgan-style approach [49] to derive rules for verification condition generation and refinement laws of dR. The authors have developed new methods and Isabelle components [50] to support the modeling and the verification of hybrid programs using dR and dH in Isabelle/HOL. dR and dH are implemented in Isabelle/HOL, a proof assistant that combines a high level of automation with a unique big and coherent library of theorems about differential equations. Furthermore, Isabelle/HOL brings the advantage of generality, i.e the proof completed for a given hybrid system could be reused in a longer proof for a complex system.

3.2.6 Modeling and Verifying Hybrid Systems with Coq and Coquelicot

The authors of [51] present a new approach for modeling and verifying hybrid systems using the interactive theorem prover Coq and its library Coquelicot for real analysis [52]. The Coq system is based on recursive calculus that combines both higher-order logic and a richly typed functional programming language. Programs can be extracted from proofs into external programming languages such as OCaml or Haskell. The approach of [51] proposes to encode in the C programming language a discrete representation of a continuous differential equation that describes the behavior of a 1D sound wave system. This C program has two different sets of annotations. The first one relates to continuous definitions (derivation, approximation by Taylor series, etc.) and the second one relates to discrete aspects of the program (loop invariants, pre-conditions and post-conditions of the used functions, etc.). Frama-C is used to extract these annotations and projects them into Jessie [53] or Why [54]

that generates proof obligations. These POs are discharged automatically or interactively using the SMT solver or interactively using Coquelicot.

3.2.7 Discussion

In this section, we presented the differential refinement logic dR$\mathcal{L}$ that extends the differential logic d$\mathcal{L}$ in order to define a relation of refinement between hybrid programs. The proof obligations in dR$\mathcal{L}$ are difficult to discharge since there is no tool that supports this logic. Moreover, performing proofs using KeYmaeraX, the automatic theorem prover for d$\mathcal{L}$, requires us to guess the relevant invariants which is not always possible. For this purpose, the approach introduced in [40] proposes to replace the refinement strategy designed for d$\mathcal{L}$ by a method called the component-based that also simplifies the complexity of hybrid systems. Unlike dR$\mathcal{L}$, developing hybrid systems with Event-B permits to deal with the complexity of the system by incrementally introducing the properties. Moreover, Event-B permits to have a good view on the proof activity and its different steps that helps us to have a better understanding of the system.

We have also presented three proof-based methods for modeling and verifying hybrid systems. The approaches presented in Section 3.2.4 are commonly used for modeling and verifying distributed hybrid systems. The main limitation of these approaches is that they represent higher-order logic without providing a means of checking this logic, making them difficult to use. The approach introduced in [47] proposes to model and verify hybrid systems using Isabelle/HOL. Compared to the level of automation in Isabelle/HOL, more automatic proofs can be discharged in KeYmaera X. Moreover, KeYmaera X proofs can be reused in Isabelle/HOL proofs [55]. Last, the approach described in [51] proposes to use Coq and its library Coquelicot to model and verify the continuous behaviors of hybrid systems. The approaches [47, 51] use two well known theorem provers, Isabelle/HOL and Coq. In our work, we are interested by proof-based approaches that use the formal method Event-B and its refinement strategy in order to bridge the gap between modeling and implementing cyber-physical systems.

3.3 Event-B Based Approaches

In this section, we focus on existing proof-based approaches, based on Event-B, that enable modeling and verification of hybrid systems.

3.3.1 A Formal Approach for Correct-by-Construction System Substitution

The approach presented by G. Babin et al in [56] allows formal modeling and verification of hybrid systems using discrete Event-B, Rodin tools and theory plugins. This approach relies on proof, refinement, and discretization of continuous functions to manage the evolution of discrete controllers. This development includes three levels. The first level defines an abstract model of the controller, the second level introduces a continuous controller, and the third level builds a discrete controller. This approach is illustrated by developing a stability controller. It is a simple stability controller for a generic plant model characterized by a single continuous function f that models its behavior. Control actions with this system are simple. This consists in shutting down the system when it exceeds the limits m and M representing the minimum/maximum values of the continuous variable. The goal of this development is to show how a controller featuring a simple state transition system and a physics plant featuring a continuous function can be formally integrated into a single formal Event-B development that incrementally encodes a hybrid automaton.

Replacing the control part (generic system) with the control part of the selected case study, the *Water Tank* case study, yields the following system specifications: if the water

level is kept rising until it reaches M, the system enters an operating state in which the water level can rise or fall, but the limits m and M must not be exceeded. Exceeding these limits will cause the system to enter a shutdown state and reduce the water level to zero. Developing a stability controller with this approach consists of three steps. The first step is to define the behavior of the system controller at an abstract level. After modeling the system at an abstract level with three discrete states, the second step is to introduce a continuous controller by defining a continuous function $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ that specifies the behavior of the plant. The final step is to implement the discrete controller. It is therefore essential to define the correct discretization of time that preserves the continuous behavior introduced in the second step. This step also introduces a margin z that allows the controller to anticipate the next observable behavior before the spurious behavior occurs. The control strategy for the last two steps is to check the values obtained from the control system after a control step called dt. That is, the controller must examine future values of f before deciding whether to keep the system in the same discrete state or change. For this purpose, the approach uses the notion of the variable *now* which models the current time as in the approaches [57, 58].

3.3.2 A Correct-by-Construction Design of Hybrid Systems in Event-B

The approach introduced by G. Dupont et al. in [10], uses the plug-in Theory of Event-B in order to handle continuous aspects of CPSs. It defines a theory named *DiffEq* that provides several abstract operators to model differential equations and their relevant properties. The approach presents a framework for modeling and verifying hybrid systems. This framework consists of two main patterns: the generic pattern, designed for modeling hybrid systems described with ODEs that admit exact solutions, can be applied to three types of CPSs architectures (single-controller-to-single-plant, single-controller-to-many-plants, many-controllers-to-many-plants), the approximated pattern can be applied to prove the refinement relation between an ordinary differential equation system and an approximately equivalent one.

3.3.2.1 The Generic Pattern

The behavior of CPSs is specified by the following three Event-B models: *System* model, *State System* model and *Controlled System* model.

- **System model** is used to describe the continuous evolution of the time t and the generic continuous measurement, represented by the variable $plantV$, using some operators of the *DiffEq* theory. $plantV$ is specified as a function ($plantV \in \mathbb{R}^+ \rightarrow S$) where $\mathbb{R}^+$ represents time and S is a constant defined in the associated context as follows: $\mathbb{R}^n$ with n representing the number of continuous variables of the system. For example, in case of the *Bouncing Ball* case study, we have two continuous variables x and v, so S is equal to $\mathbb{R} \times \mathbb{R}$. The progression of the time t is modeled by an event named *Progress* (see Figure 3.1), which states that the new value of t will become greater than its previous value. An Event-B event expresses the transformation applied to the state under the form of a predicate, called the before-after predicate, that links the value of the state before the event is triggered, and its value after it has been triggered[1]. In such a predicate, noted $: |$, x' denotes the after-value of the variable x [18].

  The behavior of $plantV$ is specified by the event *Behave* (see Figure 3.2). The clause $ANY\ e$ specifies that the parameter e must be chosen such that the guards $grd1$ and $grd2$ are verified. These guards respectively state that e belongs to $DE(S)$, set of differential equations built on S i.e e must have values in S, and must have a solution

[1] Substitution $(t : |t' \in \mathbb{R}^+ \wedge t < t')$ can be rewritten into (**ANY** t_1 **WHERE** $t_1 \in \mathbb{R}^+ \wedge t < t_1$ **THEN** $t := t_1$ **END**

EVENT Progress
THEN
act1: t :| $t' \in \mathbb{R}^+ \wedge t < t'$
END

Figure 3.1: Event Progress

in the interval $[t, \infty[$. Variable *plantV* evolves according to e in the intervals $[0, t[$ and $[t, \infty[$, which is specified using the *AppendSolutionBAP* operator defined in *DiffEq* (see *act*1). This operator permits to update the *plantV* by calculating its new value on both intervals $[0, t[$ and $[t, \infty[$. Consequently, the previous values on $[0, t[$ are overridden. In Event-B, events may be triggered only when all their guards are satisfied. Therefore, the event *Progress* is triggered continuously to update the time t and once the guards of the event *Behave* are satisfied, the event can be triggered to update the continuous variables until the new value of t. In the action *act*1, t denote the current value of variable t before the execution of the event *Behave*.

EVENT Behave
ANY e
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable([t, \infty[, e)$
THEN
act1: $plantV : |plantV' \in \mathbb{R}^+ \rightarrow S \wedge AppendSolutionBAP(e, \mathbb{R}^+,$
$[0, t[, [t, \infty[, plantV, plantV')$
END

Figure 3.2: Event Behave

- **State System model** refines the abstract *System* model by introducing the evolution of the discrete part represented by the controller. This evolution is specified by an event that updates the value of a variable modeling the possible states of the controller. Then, the interaction between the plant and the controller is modeled in the *Controlled System* model that refines the *State System* model.

To get the concrete model corresponding to a hybrid system using this approach, the Theory plug-in [20] is used to define all the concrete ODEs that describe the continuous behavior of the modeled system, and the properties on these ODEs are specified using the *DiffEq* operator *lipschitzContinuous* to ensure that they admit solutions.

3.3.2.2 The Approximation Pattern

In [59], G. Dupont et al. introduce a new Event-B pattern that encodes an approximate refinement operation for modeling and verifying hybrid systems described with ODEs that do not admit exact solutions. This pattern was developed to formalize commonly used approximation operations implemented by designers. It consists of an Event-B model that refines the generic model and encodes an approximate refinement operation. An example of such an approximation is linearization. Nonlinear ordinary differential equations are approximately refined by linear ordinary differential equations. The authors consider the well-known *Inverted Pendulum* case study to illustrate the usage of this pattern. This problem is particularly important because it does not admit an exact solution and requires the use of linearization for proper implementation. To implement this pattern in Event-B, the authors developed an approximation theory. This theory defines a set of operators used to model approximations as a refinement:

- Approximation Operator $\approx^{\delta}$: allows modeling the approximate equality in Event-B. $x \in E$ is approximately equals to $y \in E$ according to an approximation factor $\delta \in \mathbb{R}^{+}$ iff $d(x, y) \leq \delta$, where d is the distance of the space metric (E, d).

- δ-membership ($\in^{\delta}$). Let $S \subseteq E$ and $y \in E$. y belongs to S up to δ, denoted $y \in^{\delta} S$, iff $\exists x \in S, d(x, y) \leq \delta$.

To illustrate the application of this pattern, the authors use the *Inverted Pendulum*. To model this case study, three steps are identified. The first step consists in defining a theory that holds every important concepts needed to model this kind of system: differential equations (both non-linear and linearised) and adequate controls for the systems, as well as various physical and mathematical properties that will help in establishing the correctness of the system. The second step consists in modeling the abstract model of the case study, which features the nonlinear differential equation. The final step consists in using the approximation refinement to model the concrete model of the case study, which features the linear differential equation

3.3.3 Modeling and Refining Hybrid Systems in Event-B

The authors of [57] present a new approach to model an hybrid system with Event-B supported by the Rodin toolset. They start by modeling the discrete part of the hybrid system and then introduce the development of the continuous part using refinement strategy. The continuous part is represented by a variable named x and its evolution over time is described by a continuous function $x\_c(t)$. The discrete part is modeled using two discrete events UP and DN. For example, in the case of a *Water Tank*, DN represents the valve and UP the pump. The UP event is activated when $x\_c(t)$ is equal to the abstract value represented by 0 in the generic model, and the DN event is activated when $x\_c(t)$ is equal to the abstract value represented by 1. The time in Event-B must be explicitly modeled, so the author uses the event *CLICK* to introduce the progression of time. This updates a discrete variable named *now* to represent the current time. This is very similar to what we propose. The *CLICK* event is fired each time a normal DN or UP events is performed. The control strategy chosen by the approach should ensure that $x\_c(t)$ is always defined on the interval $[0, 1]$. To complement this approach, the author suggests adding the use of MATLAB to verify the system by examining analytical solutions of differential equations associated with continuous functions.

The authors of [58] propose an approach to model hybrid systems using Event-B. This approach, supported by the Rodin toolset and Theory plugin, introduces new concepts to Event-B: real, continuous and monotonic functions. The authors start by modeling the continuous part and then introduce the discrete part. Similar to [57], the evolution of the continuous part is described by a continuous function over real intervals and the progression of time is described by the variable *now* and the *Click* event. Preserving the properties of the continuous functions is the key for ensuring the correctness of refined machines. Additionally, the authors use a monotonic continuous function to ensure that "nothing bad" happens between two executions of the *Click* event. The main limitation of these approaches [57, 58] is that they use continuous functions instead of using differential equations that are more representative of the continuous parts of hybrid systems. Moreover, these approaches treat only *Event-Triggered* systems which restrict their use.

3.3.4 A Refinement Strategy for Hybrid System Design

The approach proposed in [60] is inspired by the work of [57] to integrate traditional refinement in hybrid system modeling. The approach is based on a series of refinement steps, each step aims to modularly introduce a specific kind of implementation detail. They start by a generic specification for hybrid systems then they refine this specification to introduce the safety property of a specific hybrid system. The third step of refinement consists in

introducing the period of control using the variable *now* such as in [58]. The period of control is used to introduce the notion of cycles. Each cycle is an interval between *now* and $now + \sigma$, where σ is a constant to model periodic cycle. The control part is introduced in the fifth step of refinement. In order to concretize the control part, the authors develop a discrete control refinement toolkit, which contains a list of domain-specific refinement steps. The final step of refinement aims to bridge the gap between the Event-B models and the implementation. Such as [57,58], the approach provides a generic pattern to only treat *Event-Triggered* systems. Moreover, the approach directly uses analytical solutions to model the evolution of the continuous part without expressing the differential equations that describe the continuous behaviors of hybrid systems.

3.3.5 Hybrid Event-B

The approach presented in [61] proposes a formal method, called Hybrid Event-B, to add continuous aspects to Event-B. It defines two kinds of events: *mode events* and *pliant events*. *mode events* represent the discrete Event-B events. *pliant events* allow the description of continuous behaviors with continuous functions and differential equations. Variables are also partitioned into two categories: *mode variables*, which represent the traditional discrete Event-B variables; *pliant variables* which can evolve both continuously and via discrete events. Time is modeled as a fixed left-closed right-open interval $T \subseteq \mathbb{R}$. To deal with discrete events, T is partitioned into a sequence of intervals, $T = ([t_0, t_1[\cup[t1, t2[\cup....)$. The structure of a Hybrid Event-B machine is described as follows: after the machine name is the $TIME$ declaration, which names a read-only variable used to denote real time (if needed). Next comes a $CLOCK$ variable *clk*, which increases at the same rate as time during every *pliant* event and which can be updated in *mode* events. Then come the $PLIANT$ and $VARIABLES$ declarations. Next come the $INVARIANTS$. Then come the $EVENTS$, starting with the $INITIALISATION$ which is a discrete event. Then come the remaining *mode* events and *pliant* events. Pliant events need new syntax, pliant variables can be assigned values either via the solution of a DE, or directly by being assigned the value of a time dependent expression, or indeed by being assigned a value consistent with some time dependent predicate.

Hybrid Event-B treats the Zeno behavior, where the time interval continually gets smaller and smaller, by adding a constant δ_{Zeno}, such that for all i, $t_{i+1} - t_i \geq \delta_{Zeno}$. The correctness of Hybrid Event-B models is ensured using a set of customized proof obligations patterns, defined in a way similar to classical Event-B. These Hybrid Event-B POs patterns allow deriving a number of proof obligations from any given Hybrid Event-B model. The major limitation of this approach is that it is not supported by any tool. However, the approach was successfully applied on many concrete examples and it was even used in the development of the approach introduced in [62] which proposes a conceptual model for hybrid systems engineering composed of a structural and a behavioral part. Moreover, it represents a source of inspiration for several approaches based on discrete Event-B including ours.

3.3.6 Discussion

The major limitation of the approaches described in [57,58] is the use of continuous functions to model the evolution of continuous parts, whereas, in practice, continuous behaviors are defined using differential equations. In our work, we propose to deal with this limitation by the use of a theory developed to treat differential equations in Event-B. The approach presented in [60] does not express differential equations in Event-B, it directly uses analytical solutions to model the evolution of the continuous part, which decreases the readability and the maintainability of the approach. Moreover, models specified by these approaches are at a level of abstraction comparable to that of *Time-Triggered* models. We think that it is easier to specify systems at a higher level of abstraction and then introduce step by step different kinds of functionalities and properties. On the other hand, Hybrid

EVENT-B supports differential equations and provides multiple concepts to treat continuous aspects of hybrid systems. However, it is not possible to use the RODIN platform to specify and prove Hybrid EVENT-B models. Indeed, in Hybrid EVENT-B, proof obligations must be generated and discharged manually, which makes it difficult to apply on critical systems.

3.4 Interfacing Theorem Provers With Computer Algebra Systems

The integration of theorem provers and computer algebra systems is of interest to many researchers. There are many ways to ensure this integration. The following methods are the most common:

- Theorem provers built on the top of computer algebra systems: approaches in this category develop new theorem provers that encapsulate existing computer algebra systems. For example, the approach described in [63] proposes a new theorem prover named Theorema 2.0, based on the computer algebra system *Mathematica*.
- Embedding a computer algebra system in proof assistants : this category is represented by the work presented in [64]. This approach represents an architecture that guarantees the results provided by the computer algebra system.
- Bridge or ad-hoc information exchange solutions: approaches in this category are based on building an interface between theorem provers and computer algebra systems and verifying the calculus generated by the computer algebra system.

Our focus is on what we call bridges or ad-hoc information exchange solutions. Approaches in this category are based on building an interface between theorem provers and computer algebra systems. Some approaches use the output of the theorem prover without checking its correctness, and others use it independently of how computer algebra systems obtained it. Among the approaches that have been developed for calling computer algebra systems from a theorem prover is the one that proposes an extensible ad-hoc interface for linking the Lean theorem prover with the computer algebra system Mathematica [65]. The results returned by Mathematica are examined separately in Lean by defining a set of tactics for each type of Mathematica expressions. We can also cite the approach that links the theorem prover Isabelle [66] and the computer algebra system Maple [23] by specifying the syntax of Maple in Isabelle and providing a prototype implementation of an interface designed by making modifications on Isabelle without modifying Maple [67].

3.4.1 An Extensible Ad-Hoc Interface between Lean and Mathematica

In this section, we describe the approach presented in [65] that proposes an extensible ad-hoc interface to link the theorem prover Lean with the computer algebra system Mathematica. The authors of this approach separate the steps of communication between the theorem prover/the computer algebra system, and the verification of simplifications made by the computer algebra system. Therefore, the results returned by Mathematica are verified separately in Lean by defining a set of tactics for each type of simplification.

Lean theorem prover is a new open source theorem prover and programming language developed by Microsoft Research in 2013. Lean can be accessed through a web browser, a JavaScript version of Lean, or installed on a user computer. Lean users can use a custom metaprogramming language to create functions that automatically prove some theorems. Lean is also based on the calculus of inductive construction (CIC). This is an extension of the lambda calculus with dependent types and an inductive definition. For example, the natural numbers are defined in Lean by:

$$inductive\ \ nat : Type$$
$$|zero : nat$$
$$|succ : nat \rightarrow nat$$

We can then define the function *add* that represents the addition operation on natural numbers in Lean. For example, the expression $x+x$ is written in Lean language: $add\ \ xx$.

$$definition\ \ add : nat \rightarrow nat \rightarrow nat$$
$$|nzero := n$$
$$|n(succm) := succ(addnm)$$

Calling Mathematica from Lean this approach expresses Mathematica expressions in Lean syntax and vice versa. It consists of the following steps (see Figure 3.3):

- **Step 1**: converts a particular Lean expression e, created using a grammar called *expr* that was designed especially in Lean to ensure the communication between Lean and Mathematica, into Mathematica syntax by using Lean functions.
- **Step 2**: converts the expression obtained in *Step 1* to a Lean expression in Mathematica syntax using Mathematica functions.
- **Step 3**: interprets the result of *Step 2* into the Mathematica representation.
- **Step 4**: uses Mathematica functions to solve mathematical problems needed to prove the safety properties of the modeled system.
- **Step 5**: converts the Mathematica expression e to a Mathematica expression in *mmexpr* syntax f. *mmexpr* is also a grammar developed by Lean for communication between Lean and Mathematica.
- **Step 6**: converts the result of the previous step into a Lean expression using the grammar *expr*.

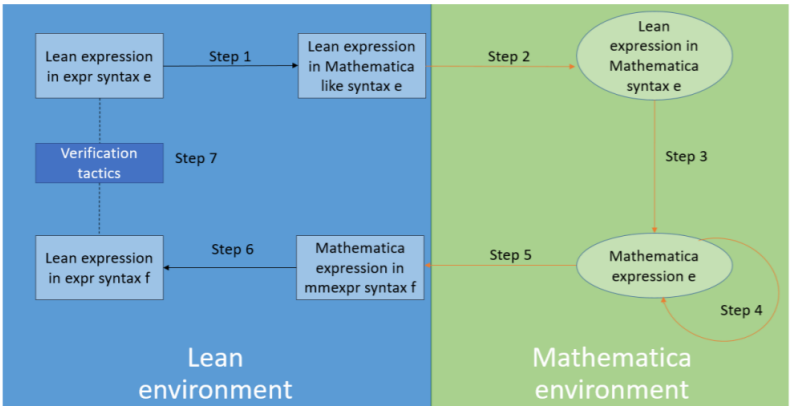

Figure 3.3: Calling Mathematica from Lean

3.4.2 An Interface between Isabelle and Maple

This section presents the approach introduced in [67] that provides a bridge between Isabelle theorem prover and the computer algebra system Maple by specifying Maple syntax in Isabelle and providing a prototype implementation of the interface designed with modifications to Isabelle without modifying Maple.

Isabelle theorem prover is a generic theorem prover for interactive theorem proving that supports a variety of logics such as the higher-order logic (HOL) implemented in Isabelle as Isabelle/HOL. Isabelle is based on the notion of theory, a named collection of types, functions, and theorems, much like a module in a programming language. A theory is built in Isabelle as follows (where $T_1 ... T_n$ are existing theories):

theory T
imports $T_1 ... T_n$
begin declarations, definitions, and proofs
end

Isabelle usually applies a proof method called simplification in order to prove theorems. Simplifications mean using equations $left = right$ from left to right (only) as long as possible. The tool that allows performing simplifications in Isabelle is called the $simplifier$. This latter uses a set of simplification rules, known as a *simpset* and declared as theorems using an attribute named *simp*.

Calling Maple from Isabelle is done thanks to a prototype implementation of an interface that enables Isabelle to communicate with a chosen computer algebra system, Maple for example. This prototype consists of a computer algebra system, a theorem prover, and a common evaluator (bridge) that evaluates expressions sent by Isabelle to be proceed by Maple as well as the results sent by Maple to Isabelle. The interface is developed by specifying the concrete syntax of Maple and extending the simplifier of Isabelle by adding new simplification rules called evaluation rules that make selected operations of Maple available and control the access to Maple. These new simplification rules are presented in Isabelle as data structures which contain a list of premises, a term pattern, and the name of a function which enables to call Maple.

3.4.3 Discussion

The approaches introduced in [65,67] cannot be reused since they do not deal with differential equations. In [68] the authors develop an algorithm that implements the *Runge-Kutta* methods for solving numerically ODEs with respect to the existing formalization of ODEs in Isabelle/HOL. However, the approach does not provide any mechanism to find exact solutions of ODEs. Moreover, all these approaches do not use any refinement technique, whereas one of the key elements of our approach is to take advantage of the Event-B method for designing and verifying CPS models in a stepwise manner.

3.5 Conclusion

In this chapter, we presented a state of art of the most relevant formal approaches for modelling and verifying hybrid systems. As stated before, these approaches can be grouped into two categories: model checking-based approaches and proof-based approaches. Model checking-based approaches are based on the calculation of the set of reachable states and use hybrid automata and algorithmic analysis methods to model and prove hybrid systems. As already mentioned, these approaches suffer from classical problems related to state-space explosion of the variables considered that is hard to solve. In the other hand, proof-based approaches use formal methods such as Event-B to model hybrid systems and use deductive verification to prove their safety properties.

The Event-B based approaches use the *Event-Triggered* model which is more difficult to implement. This motivates us to introduce a generic *Time-Triggered* model that refines the generic *Event-Triggered* model. The major limitation in using Event-B is it does not treat the resolution of ODEs. As stated before, the approach introduced in [10] defines the theory *DiffEq* that provides several abstract operators to model ODEs and their relevant

properties. To solve concrete ODEs, the approach consists in using the approximation concept during the refinement process. We have chosen a different approach by coupling Event-B with a differential equation solver inspired by the approaches presented in Section 3.4.

Chapter 4

Case Studies

Contents

This chapter describes three cyber-physical case studies, the *Stop Sign*, the *Water Tank* and the *Smart Heating* systems, used to illustrate our approach. While remaining simple, these case studies are didactic and quite representative of linear hybrid systems that admit exact solutions. The continuous behavior of the *Stop Sign* case study is represented by two state variables while the continuous behavior of both case studies, *Water Tank* and *Smart Heating*, is represented by a single state variable. Moreover, the Stop Sign case study is represented by two different modes of control that require a single safety envelope. In the other hand, the *Water Tank* and *Smart Heating* case studies are both composed of two modes and when their controllers 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 approaches. This chapter also describes a nonlinear case study, the *Inverted Pendulum*, an example of cyber-physical systems described with nonlinear differential equations.

4.1 The Stop Sign Case Study

The *Stop Sign* control system, is inspired from that described in [16] with some simplifications. It has the objective to stop a car before a stop signal *SP* as depicted by Figure 4.1. 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 and the velocity of the car specified respectively by the state variables p and v, 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 system can be in one of the three discrete states: *Accelerating*, *Braking* and *Stopped*. The system can enter state *Accelerating* when the car is very far from the stop signal *SP*. In that case, the car is allowed to accelerate by assigning the maximum limit of acceleration *A* to *a*. State *Braking* is entered when the car is very close to the stop signal *SP*. In that case, the controller must decrease the car velocity by assigning the maximum limit of braking *-B* to *a*. The state *Stopped* is entered when the car is stopped i.e $v=0$ (consequently $a=0$) presumably right before signal *SP*. In all states, the system safety property, $p(t) \leq SP$, must be fulfilled. So contrary to [16], we did not consider states where such property is not verified since they are not reachable.

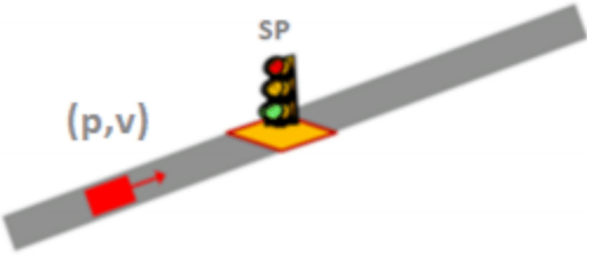

Figure 4.1: The Stop Sign System.

4.1.1 Modeling the Stop Sign Using Hybrid Automata

Figure 4.2 depicts the hybrid automaton associated with the *Stop Sign* case study. The continuous behavior is represented in the states by the ODEs that describe the evolution of the continuous variables. Each state is characterized by a local invariant. When this invariant is no longer satisfied, the system must leave the current state. The discrete behavior is represented by the transitions between the states which can be triggered automatically as soon as the state local invariant is not satisfied. A transition can be labeled by a set of constraints and a set of assignments to update the values of the variables of the system. The states of the automaton of Figure 4.2 are associated with the ODEs that describe the evolution of the car state variables *p* and *v*. The formula, $p(t) \leq p\_max$, denotes the local invariant, which is the condition to guarantee that the controller will react exactly at the right moment, by braking to allow the car to stop before the signal *SP*. Therefore the constant p_max must satisfy the following constraints:

- $p\_max \leq SP$.
- According to [16], when the car brakes with (a=-B), the next car position is calculated as follows: the position evolution $p(\Delta\ t)$ is expressed by:

$$p(\Delta t) = -\frac{1}{2} \times B \times (\Delta t)^2 + v \times \Delta t$$

The speed evolution $v(\Delta\ t)$ is expressed by:

$$v(\Delta t) = -B \times \Delta t + v$$

The car stops when its speed is zero, that is: $v(\Delta\ t)=0 \Rightarrow \Delta\ t= \frac{v}{B}$. Thus:

$$p(\Delta t) = -\frac{1}{2} \times (\frac{v}{B})^2 + v \times \frac{v}{B} = \frac{1}{2} \times \frac{v}{B}^2$$

To guarantee that the car will not exceed SP, p_max must satisfy the following constraint $p\_max \leq SP$ - $\frac{v^2}{2B}$.

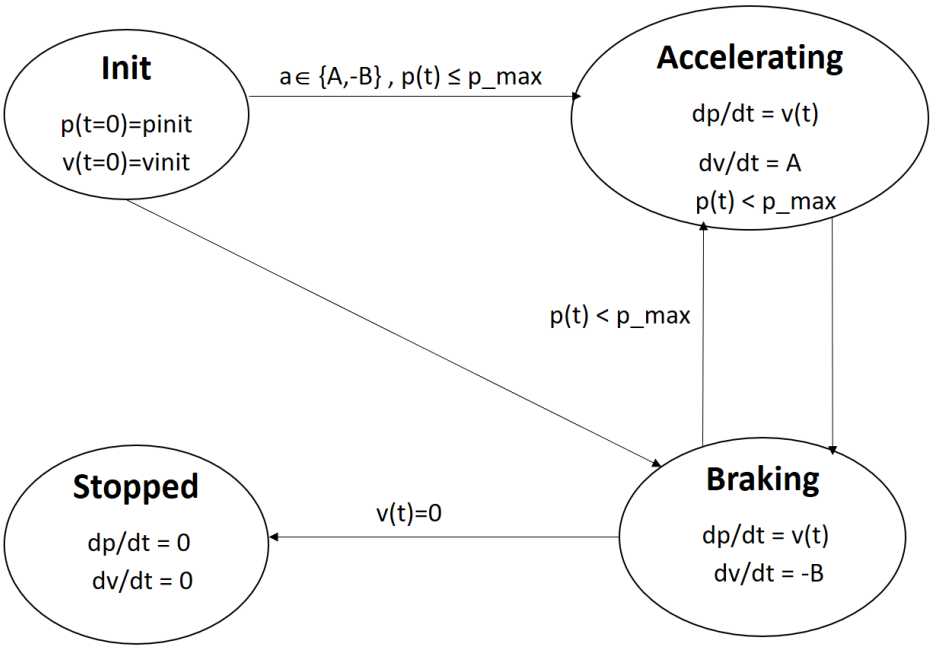

Figure 4.2: Hybrid Automaton of the Stop Sign System.

The discrete dynamic is represented as transitions between the states of the hybrid automaton. For example, when the formula $p(t) \leq p\_max$ is true, the car is non-deterministically allowed accelerating with $\frac{dv}{dt} = A$ (*Accelerating* state), or braking with $\frac{dv}{dt} = -B$ (*Braking* state). While when the formula $p(t) \leq p\_max$ is no longer true, the car must brake, and move to the state *Braking* if it is not already in it. As depicted by Figure 4.2, the system can be in one of the following discrete states:

- *Init state*: the initial values of the position and velocity are respectively represented by the constants *pinit* and *vinit*. These constants must be chosen such that $pinit < p\_max$.
- *Accelerating state*: in this state, the car is allowed to accelerate with a velocity A. As long as the formula $p(t) \leq p\_max$ is true, the car can either accelerate or brake. Once this formula is no longer satisfied, the car must switch to the *Braking* state.
- *Braking state*: in this state, the car must decelerate with a deceleration equal to $-B$ until it stops before the signal SP. If the current values of the position and velocity satisfy $p(t) \leq p\_max$, the car is then allowed to switch to the *Accelerating* state, transition from *Braking* to *Accelerating* states.
- *Stopped state*: the car stopped ($v = 0$). If the car is far enough from SP, the system can enter again in the state *Accelerating* which is represented by the transition from the *Stopped* to *Accelerating* states.

4.1.2 Modeling the Stop Sign Using Hybrid Programs

The authors of [16] have chosen the *Stop Sign* case study to illustrate the use of the differential dynamic logic d$\mathcal{L}$ and the use of the platform KeYmaera to model and prove linear hybrid systems. They started with a simple system, represented by the hybrid

program *Model 4.1*, in which the car starts at some non negative velocity $v \geq 0$ and accelerates at a constant rate denoted $A > 0$ (the maximum acceleration) along a straight lane. Then, they introduce some complexity to this model in order to model the interaction between the physical and discrete parts. This approach inspired us to instantiate our generic models (see Chapter 6).

The continuous behavior of *Model 4.1* is very simple. It is represented by the derivative of the position p which is the velocity v, $\frac{dp}{dt} = v(t)$, and the derivative of the velocity which is the maximum acceleration, $\frac{dv}{dt} = A$. The safety property is specified by the formula $v \geq 0$ which states that the car should never travels backward in space.

Model 4.1: A Sample Stop Sign Case Study

$$init \rightarrow [plant](req)$$
$$init \equiv v \geq 0 \wedge A > 0$$
$$plant \equiv p' = v, v' = A$$
$$req \equiv v \geq 0$$

The previous model specifies the physical part of the Stop Sign case study. *Model 4.2* introduces the control part represented by the discrete controller, *ctrl*, whose objective is to adjust the velocity by accelerating by A or braking by $-B$, and still never drive backward. For this purpose, *Model 4.2* introduces the variable a that models the controlled variable, the acceleration. The controller assigns a value to a according to the discrete state of the system: *Accelerating state* $a = A$, *Braking state* $a = -B$ and Stopped state $a = 0$. To model the interaction between the physical part and the continuous part, the derivative of the velocity is now the variable a. This model also introduces the evolution domain $v \geq 0$ that restricts the continuous evolution of the system to stay within that domain.

Model 4.2: Safety Property of a Hybrid Car Model

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv v \geq 0 \wedge A > 0 \wedge B > 0$$
$$ctrl \equiv a := A \cup a := 0 \cup a := -B$$
$$plant \equiv p' = v, v' = a \;\&\; v \geq 0$$
$$req \equiv v \geq 0$$

To add some complexity to the system, they model a stop sign assistant by introducing the controller part. For this purpose, *Model 4.3* introduces the safety envelope $safe = p + \frac{v^2}{2B} < SP$ ($p + \frac{v^2}{2B} < SP \equiv p < p\_max$). The controller of such system interrupts the physical part when certain events occur, which is specified by an *Event-Triggered* model. The choice of the acceleration in the *Accelerating state* is done in a deterministic way by assigning A to the acceleration a. The continuous part is composed of a non-deterministic choice between two differential equations specified by the formula *plant*. The difference between this two formulas is in their evolution domains. In fact, hybrid programs model the notion of transitions between discrete states of a hybrid system by adding a constraint to the system evolution domain. In the case of the *Stop Sign* case study, this constraint is represented by the domain $p + \frac{v^2}{2B} \leq SP$ which includes the domain of the formula $safe$. Therefore, when the car is rolling inside this domain, the controller can non-deterministically choice between the *Accelerating state* or the *Braking state*. Once the car is in the limit of the domain of the formula $safe$, i.e $p + \frac{v^2}{2B} = SP$, the controller must execute to force it to move to the *Braking state* that is the reason why the formula $p + \frac{v^2}{2B} \geq SP$ is added to the system evolution domain in the second formula of *plant*.

Model 4.3: Event-Triggered Car Model

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv v \geq 0 \wedge A > 0 \wedge B > 0 \wedge safe$$
$$safe \equiv p + \frac{v^2}{2B} < SP$$
$$ctrl \equiv (?safe; a := A) \cup (?v = 0; a := 0) \cup (a := -B)$$
$$plant \equiv p' = v, v' = a \;\&\; p + \frac{v^2}{2B} \leq SP \wedge v \geq 0$$
$$\cup\, p' = v, v' = a \;\&\; p + \frac{v^2}{2B} \geq SP \wedge v \geq 0$$
$$req \equiv p \leq SP$$

Model 4.3 is refined by *Model 4.4* that follows the generic model of a *Time-Triggered* system. The physical part of this model began with the formula $t := 0; p_0 := p$ which allows to reset the variable t to 0 after each execution of the controller in order to start a new control phase.

Model 4.4: Time-Triggered Car Model

$$init \rightarrow [(ctrlV; plantV)^*](req)$$
$$init \equiv v \geq 0 \wedge A > 0 \wedge B > 0 \wedge p + \frac{v^2}{2B} \leq SP \wedge \epsilon > 0$$
$$safe_\epsilon \equiv p + \frac{v^2}{2B} + (\frac{A}{B} + 1)(\frac{A}{2}\epsilon^2 + \epsilon v) \leq SP$$
$$ctrl \equiv (a := -B) \cup (a := A; ?safe_\epsilon)$$
$$plant \equiv t := 0; p_0 = p; (p' = v, v' = a, t' = 1 \;\&\; t \leq \epsilon \wedge v \geq 0)$$
$$req \equiv p \leq SP$$

The *Event-Triggered* model is easier to prove using KeYmaera but it is more difficult to implement. When proving the system safety properties in the *Event-Triggered* model and the refinement relation between the corresponding *Time-Triggered* and *Event-Triggered* models, we can admit that the *Time-Triggered* model satisfies the safety property of the system. The program *Model 4.5* represents the content of the KeYmaera file associated with *Stop Sign* case study, where the clause *ProgramVariables* specifies the system parameters that are defined as real numbers which are represented by the symbol R and the clause *Problem* models the initial, the physical and the discrete parts of the system as well as the safety property.

Model 4.5: KeYmaera Program for the Stop Sign Case Study

```
\programVariables {
   R p, v, a, A, B, SP;
}
\problem {
   v ≥ 0 ∧ A > 0 ∧ B > 0 ∧ p + (v × v)/(2 × B) < SP− >
   \ [
   (((?p + (v × v)/(2 × B) < SP; a := A) + +(?v = 0; a := 0) + +(a := −B));
   {p′ = v, v′ = a, p + (v × v)/(2 × B) ≤ SP, v ≥ 0} + +
   {p′ = v, v′ = a, p + (v × v)/(2 × B) ≥ SP, v ≥ 0}
  ) *
  \](p ≤ SP)
 }
```

4.2 The Water Tank Case Study

The second case study deals with a water tank, also known as a heat pump water heater, 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 water tank, a water pump to fill the tank and a water level sensor to get the level of the water in the tank as depicted by Figure 4.3. The control strategy is to activate the pump when the water level is too close to V_low and deactivate it when the water level is too close to V_high. The continuous behavior of the water tank over time is represented by the level of the water specified by the variable Vol that evolves according to the flowing linear ordinary differential equations, $\frac{dVol}{dt} = -f\_out$ or $\frac{dVol}{dt} = f\_in$. The flow of the water can be either f_in when the pump is activated or $-f\_out$ otherwise.

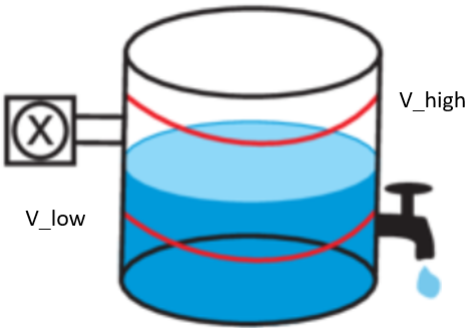

Figure 4.3: The Water Tank System

4.2.1 Modeling the Water Tank Using Hybrid Automata

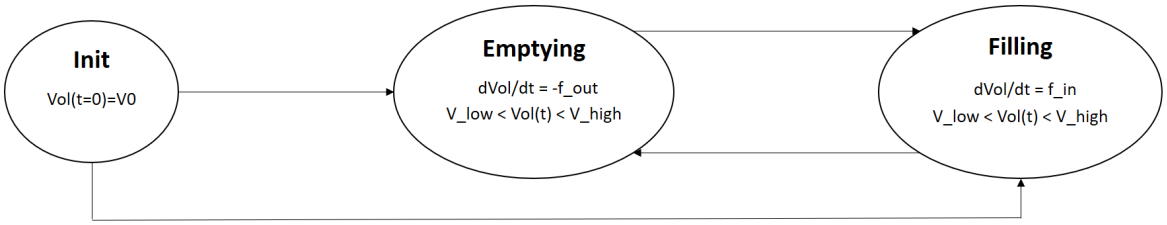

Figure 4.4: Hybrid automaton of the Water Tank System

Figure 4.4 depicts the hybrid automaton associated with the *Water Tank* case study. Two safety envelopes are defined, $safeFill = Vol(t) < V\_high$ and $safeEmp = Vol(t) > V\_low$, which are defined to guarantee that the controller will react exactly when the water level is too close to the two limits V_high and V_low. The system can be in one of the following three discrete states:

- *Init state:* represents the initial state. It is characterized by a constant level of the water denoted $V0$, which must be chosen between V_low and V_high to guarantee that the system is safe initially: $V\_low < V0 < V\_high$.
- *Filling state:* in this state, the pump is activated to fill the water tank by f_in as long as the formula *safeFill* is satisfied. Once this formula is no longer satisfied, the system must move to state *Emptying*.
- *Emptying state:* in this state, the pump is disabled to empty the water tank by $-f\_out$. Once the formula *safeEmp* is no longer satisfied, the system must move to state *Filling*.

4.2.2 Modeling the Water Tank Using Hybrid Programs

To describe an hybrid system using hybrid programs notation we can either translate the associated hybrid automaton or start by an abstract hybrid program and then enrich this model step by step, using an incremental approach as described in the previous section 4.1.2. To model the *Water Tank* case study using hybrid programs notation, we start with an abstract water tank where we describe the continuous part of the *Water Tank* case study (see *Model 4.6*):

Model 4.6: Abstract Water Tank

$$init \rightarrow [plant](req)$$
$$init \equiv Vol \geq 0 \land f\_in > 0$$
$$plant \equiv Vol' = f\_in$$
$$req \equiv Vol \geq 0$$

We add some complexity to *Model 4.6* by modeling the interaction between the control and the physical parts and by introducing the system safety envelops. We also introduce the *emptying* state by assigning the value $-f\_out$ to the controlled variable $ctrlV$. The physical part evolves according to the discrete state represented by the controlled variable $ctrlV$ as depicted by the *Model 4.7* which represents the *Event-Triggered* model associated with the *Water Tank* case study.

Model 4.7: Event-Triggered Water Tank Model

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv Vol \geq 0 \land f\_in > 0 \land f\_out > 0 \land V\_low < Vol < V\_high$$
$$safeFill \equiv Vol < V\_high$$
$$safeEmp \equiv Vol > V\_low$$
$$ctrl \equiv (?safeFill; ctrlV := f\_in) \cup (?safeEmp; ctrlV := -f\_out)$$
$$plant \equiv Vol' = ctrlV \;\&\; Vol \leq V\_high \land Vol \geq 0$$
$$\cup\, Vol' = ctrlV \;\&\; Vol \geq V\_low \land Vol \geq 0$$
$$req \equiv V\_low \leq Vol \leq V\_high$$

In the associated *Time-Triggered* system represented by *Model 4.8*, we replaced respectively the safety envelopes $SafeFill$ and $SafeEmp$ by $SafeEpsilonFill$ and $SafeEpsilonEmp$ to take into account the control period ϵ.

Model 4.8: Time-Triggered Water Tank Model

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv Vol \geq 0 \land f\_in > 0 \land f\_out > 0 \land V\_low \leq Vol \leq V\_high \land \epsilon > 0$$
$$safeEpsilonFill \equiv Vol + (ctrlV \times \epsilon) \leq V\_high$$
$$safeEpsilonEmp \equiv Vol + (ctrlV \times \epsilon) \geq V\_low$$
$$ctrl \equiv (?safeEpsilonFill; ctrlV := f\_in) \cup (?safeEpsilonEmp; ctrlV := -f\_out)$$
$$plant \equiv t = 0; V0 = Vol; Vol' = ctrlV, t' = 1 \;\&\; t \leq \epsilon \land Vol \geq 0$$
$$req \equiv V\_low \leq Vol \leq V\_high$$

4.3 The Smart Heating Case Study

The *Smart Heating* case study deals with a heater equipped with a thermostat controller whose objective is to maintain the temperature between a high level T_max and a low level T_min, with $0 < T\_min < T\_max$. The heater is switched "off" if the temperature is too close to T_max and it is switched "on" if the temperature is too close to T_min. The continuous behavior of this system is represented by the temperature T that evolves according to the flowing linear ODEs, where $temp$ denoting the flow of the temperature:

- when the mode of the heating system is "on", the value of temperature follows: $\frac{dT}{dt} = temp$;
- when the mode of the heating system is “off”, the value of temperature follows: $\frac{dT}{dt} = -temp$.

4.3.1 Modeling the Smart Heating Using Hybrid Automata

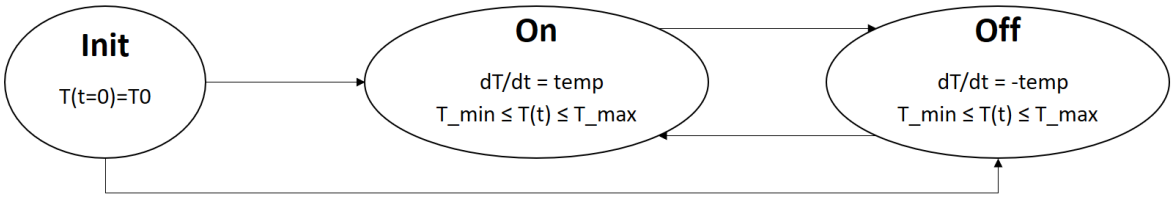

Figure 4.5: Hybrid automaton of the Hybrid Smart Heating System

Figure 4.5 depicts the hybrid automaton associated with the *Smart Heating* case study. The formula, $T\_min \leq T(t) \leq T\_max$, denotes the local invariant. The discrete behavior of this system is described by the following three discrete states:

- *Init* state: represents the initial state. It is characterized by the initial temperature degree denoted $T0$, which must be chosen between T_min and T_max ($T\_min < T0 < T\_max$) to guarantee that the system is safe initially.
- State *On*: in this state, the heater is switched "on" as long as the formula, $T < T\_max$, is satisfied. Once this formula is no longer satisfied, the system must move to the state *Off*.
- State *Off*: the heater is switched "off" as long as the formula, $T > T\_min$, is satisfied. Once this formula is no longer satisfied, the system must move to the state *On*.

The *Smart Heating* case study is a bit similar to the *Water Tank* case study. Contrary to the *Stop Sign* case study which describes a system with two modes of control, *normal* mode and *evade* mode, the *Smart Heating* and the *Water tank* case studies both describe a system with two normal modes and when the system is in one of the normal modes the second one represents the evade mode.

4.3.2 Modeling the Smart Heating Using Hybrid Programs

Model 4.9 describes the hybrid program associated with the Smart Heating system. The control part is modeled as a non-deterministic choice ($\cup$): if the safety envelope $safeOn$ is satisfied (resp. $safeOff$), we assign $temp$ or (resp. $-temp$) to the controlled variable $ctrlV$. The physical part is modeled using the ordinary differential equation, $T' = ctrlV$, and the formula $T \leq T\_max \wedge T \geq T\_min$, which specifies the event trigger of this model. For

example, if the controller assigns $temp$ to $ctrlV$, the generic ordinary differential equation is replaced by $T' = temp$ as longer as the formula event trigger is satisfied. Once the event trigger is no longer satisfied, the controller must update the value of $ctrlV$.

Model 4.9: The Event-Triggered Model of the Smart Heating System

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv T \geq 0 \wedge temp > 0 \wedge T\_min < T < T\_max$$
$$safeOn \equiv T < T\_max$$
$$safeOff \equiv T > T\_min$$
$$ctrl \equiv (?safeOn; ctrlV := temp) \cup (?safeOff; ctrlV := -temp)$$
$$plant \equiv T' = ctrlV \ \& \ T \leq T\_max \wedge T \geq 0$$
$$\cup \ T' = ctrlV \ \& T \geq T\_min \wedge T \geq 0$$
$$req \equiv T\_min \leq T \leq T\_max$$

Model 4.10 represents the *Time-Triggered* model associated with the *Smart Heating* system where we replaced the safety envelops $safeOn$ and $safeOff$ respectively by $safeEpsilonOn$ and $safeEpsilonOff$ to take into account the period of control ϵ, so $safeEpsilonOn = T + (ctrlV \times epsilon) < T\_max$ land $safeEpsilonOff = T + (ctrlV \times epsilon) > T\_min$. Moreover, we replaced the formula event trigger by the formula $t \leq epsilon$ to specify that the controller must react at least every epsilon time.

Model 4.10: The Time-Triggered Model of the Smart Heating System

$$init \rightarrow [(ctrl; plant)^*](req)$$
$$init \equiv T \geq 0 \wedge temp > 0 \wedge T \leq T\_max \wedge T \geq T\_min \wedge \epsilon > 0$$
$$safeEpsilonOn \equiv T + (ctrlV \times epsilon) \leq T\_max$$
$$safeEpsilonOff \equiv T + (ctrlV \times epsilon) \geq T\_min$$
$$ctrl \equiv (?safeEpsilonOn; ctrlV := temp) \cup (?safeEpsilonOff; ctrlV := -temp)$$
$$plant \equiv t = 0; T0 = T; T' = ctrlV, t' = 1 \ \& \ t \leq epsilon \wedge T \geq 0$$
$$req \equiv T\_min \leq T \leq T\_max$$

4.4 The Inverted Pendulum Case Study

In this section, we describe a nonlinear case study, the *Inverted Pendulum*, which will be used in Section 8 to demonstrate how we can use linearization methods to obtain a linear model of this case study. In contrast to the previous section, which suggested modeling linear case studies, the purpose of this section is to present the properties of the *Inverted Pendulum* and show how to linearize it in Section 8. Note that in this thesis we are only interested in modeling linear systems. For nonlinear systems, it is recommended to use linearization techniques whenever possible to apply approaches for modeling and verifying linear systems. The *Inverted Pendulum* is a well-known case study in dynamics and control theory that consists of a pendulum of mass M attached to the top of a rigid rod of length l, that is itself attached to a moving cart and which can move in two directions. A simple application of the inverted pendulum in real life is the balancing of a broom on the palm of the hand, as long as possible. The Inverted Pendulum controller has for objective to stabilise the rod in its vertical position. The control strategy is to balance the inverted pendulum by applying a force F to the cart in a way to make it vertical again. The system is subject to standard G-force, of intensity g. In [59], instead of attaching the rod to a cart, G. Dupont et al. chose to model an inverted pendulum attached to a step motor (see Figure

4.6). In that case, the force F is replaced by a torque u, the rotational equivalent of linear force, provided by the motor. The main difference between a torque and a force is that a torque results from a circular or rotational movement and the force results from a rectilinear movement. Therefore, the controller of this system has for objective to stabilise the rod in its vertical position by controlling the motor and its torque u. The continuous behavior of this system is modeled by the angle between the rod and the vertical axis, denoted θ, that permits to identify the position of the pendulum, and the angular velocity $\dot{\theta} = \frac{d\theta}{dt}$ that allows to identify the velocity of the pendulum. These state variables are represented in [59] by the state vector $\eta = [\theta\ \dot{\theta}]^T$.

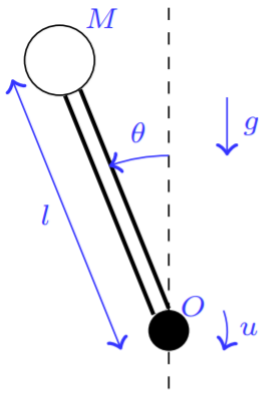

Figure 4.6: The Inverted Pendulum

The associated ordinary differential equation is derived from the equation: $\ddot{\theta} = \frac{g}{l}\sin\theta$ with $\ddot{\theta} = \dfrac{d\dot{\theta}}{dt}$, taking into account the controlled variable u. This equation is calculated using the kinetic energy and the potential energy of gravity as follows: the kinetic energy of this system is: $E_c = \dfrac{ml^2\dot{\theta}^2}{2}$, and the potential energy of gravity is: $E_p = mgl(1 + \cos\theta)$. In the absence of the force F applied to the system, it is possible to write the conservation of mechanical energy, $E = E_c + E_p$. This mechanical energy is constant during the movement. By deriving this expression over time, we obtain, where $\ddot{\theta} - \dfrac{g}{l}\sin(\theta) = u\cos(\theta)$ [59]:

$$f_{NonLin}((\theta, \dot{\theta}), u) = (\dot{\theta}, u * cos(\theta) + \frac{g}{l})$$

4.5 Conclusion

In this chapter, we presented the case studies that we have chosen to illustrate the feasibility of the approaches developed during my thesis. The *Stop Sign*, the *Water Tank* and the *Smart Heating* case studies are linear hybrid systems that admit exact solutions, polynomial ordinary differential solutions. As stated before, there are two main types of hybrid systems, linear hybrid systems and nonlinear hybrid systems. Most linear hybrid systems admit exact solutions which makes their verification easier than nonlinear systems that must use approximation methods. The *Inverted Pendulum* system is a representative case study whose continuous behavior is described by a nonlinear differential equation. For this case study, we do not provide a hybrid automaton nor a hybrid program and so far we have not yet modeled a nonlinear system following our generic approach. However, in Section 8, we will use this case study to demonstrate how we can use linearization methods to obtain a linear model and then apply our generic approach.

Chapter 5

Modeling and Proving Hybrid Systems in Event-B

Contents

The main goal of this thesis is to propose a correct-by-construction approach for modeling and verifying hybrid systems using the Event-B formal method and the platform Rodin. The proposed approach, described in this chapter, aims at providing generic templates for modeling *Event* and *Time-Triggered* systems in Event-B and verifying the relation of refinement between these two models using the platform Rodin.

Section 5.1 describes the structure of the generic models. Section 5.2 presents the Event-B theories reused to model the generic models. Section 5.3 introduces the abstract model that specifies the continuous behavior of hybrid systems in Event-B. Section 5.4 describes the modeling of *Event* and *TimeTriggered* systems in Event-B. Section 5.5 describes the main proof obligations generated by the platform Rodin to prove the correctness of our models. Finally, Section 5.6 concludes the chapter with a discussion on the different elements of the approach.

5.1 Structure of the Generic Models

One of the objectives of the DISCONT project [9] is to elaborate correct-by-construction approaches, based on EVENT-B, to specify and verify hybrid systems. In the context of this project, we propose to represent *Event* and *Time-Triggered* templates, described in [3] and presented in Section 2.2, using the event-based paradigm of EVENT-B, in order to develop a generic template for modeling *Time-Triggered* systems in EVENT-B by providing a link with a possible abstract *Event-Triggered* system.

To model an hybrid system, our approach consists of three models as depicted by Figure 5.1, *ContSystem* model that specifies the continuous behavior of the system, *EventTriggered* model that specifies the interactions between the discrete and the continuous parts of the system, and *TimeTriggered* model that specifies the discrete behavior of the discrete part of the system. The whole models are available in Appendices A.

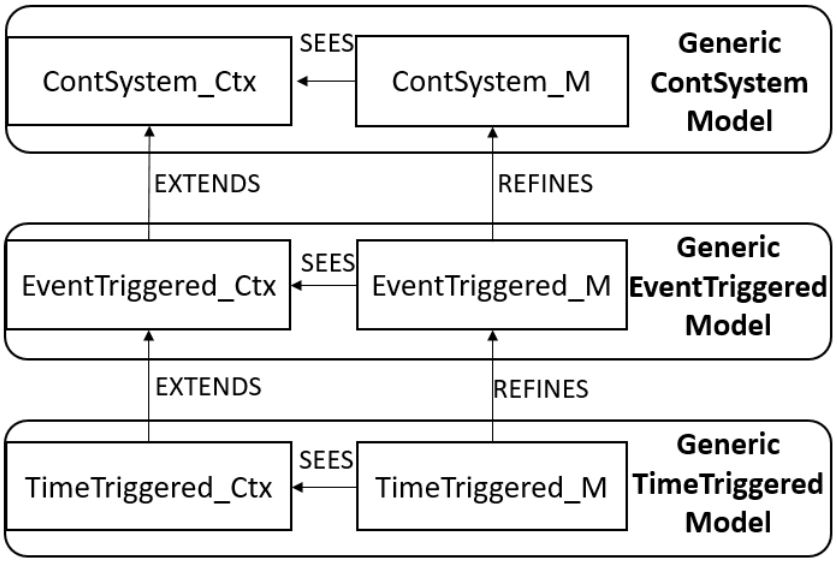

Figure 5.1: Structure of the Generic EVENT-B Specification.

5.2 Preliminary for Modeling the Generic Models

This section presents the elements that we reuse for defining our generic approach.

5.2.1 Theories for Modeling Real Numbers in EVENT-B

The theory *Real* is defined in the standard library of theories available to download at `https://sourceforge.net/projects/rodin-b-sharp/files/Theory_StdLib/StandardTheory0.1.zip/download`. It is written by Abrial and Butler, and provides: 1 datatype REAL, 13 operators: plus (+), minus (−), mult (×), leq (≤), gtr (>) etc, 24 axioms that define the semantics of the operators, 18 interactive rewrite rules for use in proofs. To treat continuous functions and ordinary differential equations in EVENT-B. The theory *Reals* introduced in [10] extends the theory *Real* by adding the operators needed to treat continuous behaviors of hybrid systems in EVENT-B.

5.2.2 Theories for Modeling Differential Equations in EVENT-B

To treat continuous aspects of cyber-physical systems, G. Dupont et al introduce in [10] a theory named *DiffEq* that provides several abstract operators to model differential equations, continuous functions and their relevant properties. In the context of this thesis, we use the following operators of the theory *DiffEq*:

- Operator *bind(specV1, specV2)*: links the generic variables with the specific ones. When replacing an abstract variable $abstV$ with specific variables, $specV1$ and $specV2$, we use this operator as follows: $abstV = bind(specV1, specV2)$.
- Operator *ode(func, Var0, t0)*: represents the constructor for differential equations in EVENT-B. $func$ specifies the right part of a given differential equation eq ($func(ctrlV) = eq$) which depends on the controlled measurement $ctrlV$, the continuous time t and the continuous variables Var with initial condition $Var(t0) = Var0$.
- Operator *Solvable(D,eq)*: states that a given ordinary differential equation eq admits solutions on domain D.
- Operator *SolutionOf(f,eq)*: models the fact that the function f is a solution of equation eq.
- Operator *AppendSolutionBAP(eq,DR,B,etap)*: updates the values of the continuous variables on the domain DR by calculating their new values using the differential equation eq on a specific time interval B. Consequently, the previous values are overridden by those of the function *etap*.

5.3 Model ContSystem

Model *ContSystem* represents the abstract model of our approach. It is inspired by the abstract model of [10] that aims at modeling the continuous part of hybrid systems in EVENT-B. Model *ContSystem* is composed of Context *ContSystem_Ctx* and Machine *ContSystem_M*.

5.3.1 Context ContSystem_Ctx

Context *ContSystem_Ctx* (see Figure 5.2) defines four constants [1].

- Constant S is defined as $S = \mathbb{R}^n$, with n representing the number of continuous variables of the system.
- Constant *TIME* to specify that the values of the continuous time are chosen in $\mathbb{R}^+$.
- Constant *sigma* is defined in $\mathbb{R}^+$ to avoid the Zeno problem in EVENT-B [61] as explained in the next section.
- Constant *plantVInit* is defined in S to represent the initial values of the continuous variables.

[1] to simplify the reading of our models, we will use in the rest of the manuscript the usual mathematical symbols instead of using that of the theory of *Reals*. Therefore, instead of using *RRealPlus* we use $\mathbb{R}^+$

CONTEXT ContSystem_Ctx
CONSTANTS S, $TIME$, $sigma$, $plantVInit$
AXIOMS
axm1: $S = \mathbb{R}^n$
axm2: $TIME = \mathbb{R}^+$
axm3: $sigma \in \mathbb{R}^+ \wedge sigma > 0$
axm4: $plantVInit \in S$
END

Figure 5.2: Context ContSystem_Ctx.

5.3.2 Machine ContSystem_M

Machine *ContSystem_M* contains two variables and two events (see Appendices A.2). Variable t represents the continuous evolution of time and Variable *plantV* represents the continuous evolution of the state variables. Event *Progress* models the progression of time (see Figure 5.3). We have adapted that defined in Section 3.3.2 to deal with the Zeno problem as the time progression can approach zero. To avoid this, we use the constant *sigma* by adding in the event *Progress* the following constraint: t' - $t \geq sigma$ to guarantee that time progression is always greater than *sigma* where t (resp. t') represents the time before (resp. after) the execution of the event.

EVENT Progress
THEN
act1: $t :| t' \in TIME \wedge t < t' \wedge t' - t \geq sigma$
END

Figure 5.3: Event Progress.

The evolution of the continuous part is modeled using Event *Plant* (see Figure 5.4) that modifies Event *Behave* (Section 3.3.2) to allow the verification of safety properties in *EventTriggered* level. Event *Plant* uses the operator *AppendSolutionBAP* adapted to update the value of *plantV* ($plantV \lhd\!\!+ plant1$) between the last time and the new value of t where the generic differential equation e belongs to $DE(S)$, the set of differential equations built on S. In other words, *AppendSolutionBAP* calculates the evolution of the physical part from the last instant until the current one t. Formally, *AppendSolutionBAP* is redefined by:

$$AppendSolutionBAP(e, ([0,t] - dom(plantV)), ([0,t] - dom(plantV)), plant1)$$

Let us remark that we adopt the same approach as in [10] to model time progression and physical behavior by two distinct events. This makes the solution more generic and adapted for systems with several physical parts, each with a specific behavior. In that case, each part is associated with its own *Plant* event and time progression is modeled in a separate event.

5.4 Event and TimeTriggered Models

As stated before, one of our objectives is to define a generic pattern for modeling *Time-Triggered* systems in Event-B by providing a link with an *Event-Triggered* generic model and use the Event-B refinement and its associated provers to demonstrate the compliance between both models.

EVENT Plant
ANY e, $plant1$
WHERE
grd1 : $e \in DE(S)$
grd2 : $Solvable([0,t] - dom(plantV), e)$
grd3 : $plant1 \in [0,t] - dom(plantV) \rightarrow S$
$\wedge AppendSolutionBAP(e, [0,t] - dom(plantV), [0,t] - dom(plantV), plant1)$
THEN
act1 : $plantV := plantV \lhd plant1$
END

Figure 5.4: Abstract Event Plant.

5.4.1 Generic EventTriggered Model

The generic *EventTriggered* model is composed of a Context named *EventTriggered_Ctx* which introduces the system safety envelope *Safe* calculated from the safety requirement that the system must satisfy, and a Machine named *EventTriggered_M* which added the discrete behavior of the system represented by the controlled variable *ctrlV*. It corresponds to the *EventTriggered* model introduced by Kopetz, also specified with dR$\mathcal{L}$, *Model 3.1* defined in Section 3.2.2.1. The semantics of this model is that the physical part evolves in parallel with the time and both are interrupted as soon as the safety envelope, represented by the formula *safe*, becomes false (represented by the expression *evt_trig* that defines the boundary of the safety envelope). As in Event-B it is not possible to state that two events are executed in parallel or to interrupt the execution of an event, we proceed as follows. The controller is executed at first to choose an adequate value of *ctrlV* that satisfies formula *safe*, then time progresses until a given value denoted by t_1 for which *evt_trig* is true with respect to the value *ctrlV*. Finally, *plant* is executed to make the physical part evolve until the current value of time t_1.

5.4.1.1 Context EventTriggered_Ctx

At this level, we express the properties desired for the system. To do this, *EventTriggered_Ctx*, depicted by Figure 5.5, extends *ContSystem_Ctx* to represent these properties. It defines a set named *EXEC* to represent the different states of a hybrid system, time progression, discrete and continuous parts ($axm3$). Moreover, *EventTriggered_Ctx* defines a set of constants and axioms:

- *safe* represents the safety envelope for the modeled system (see $axm1$). As in dR$\mathcal{L}$, the formula $safe$ depends on the current physical state variable as well as the controlled variable since it may contain some limits on how this latter may be set. The domain of this formula must be included in that of *evt_trig* formula. Moreover, $safe$ must be initially satisfied.
- *evt_trig* specifies the boundaries of the safety envelope $safe$ (see $axm2$).
- *f_evol* is used to describe how the physical state variable *plantV* evolves according to the discrete state of the system (see $axm4$).
- *f_evol_plant* is used to model the evolution of the state variable *plantV*. We must define this type of function for each discrete state of the system. Thanks to the notion of the controlled variable, we have defined a single evolution function indexed on this variable (see $axm5$ and $axm6$).
- *evade_value* is defined as a subset of $\mathbb{R}$ to represent the evade values of the modeled system (see $axm7$).

CONTEXT EventTriggered_Ctx
EXTENDS ContSystem_Ctx
SETS $EXEC$
CONSTANTS $safe$, evt_trig, $ctrl$, $plant$, prg, f_evol, f_evol_plantV, $evade\_value$
AXIOMS
axm1: $safe \in (S \times \mathbb{R}) \rightarrow BOOL$
axm2: $evt\_trig \in (S \times TIME \times \mathbb{R}) \rightarrow BOOL$
axm3: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm4: $f\_evol \in \mathbb{R} \rightarrow S$
axm5: $f\_evol\_plantV \in (\mathbb{R} \rightarrow (TIME \times S \rightarrow S))$
axm6: $\forall\, ctrlV \cdot ctrlV \in \mathbb{R} \Rightarrow (f\_evol\_plantV(ctrlV) = (\lambda\, t \mapsto plantV \cdot t \in TIME \wedge plantV \in S \mid f\_evol(ctrlV)))$
axm7: $evade\_value \subseteq \mathbb{R} \wedge evade\_value \neq \emptyset$
END

Figure 5.5: Context EventTriggered_Ctx.

5.4.1.2 Machine EventTriggered_M

Machine *EventTriggered_M* refines Machine *ContSystem_M* by adding two new variables:

- $ctrlV$ represents the controlled variable and belongs to $\mathbb{R}$. The current value of this variable corresponds to the current controller state.
- *exec* is a flag used to model the alternation between the control and the physical parts as represented in the high-level structure of hybrid programs, $(ctrl;plant)^*$. Therefore, *exec* can take two values *ctrl* and *plant*. In Event-B, time must be explicitly handled. To be sure that this explicit time will progress between the control and the plant parts, we add a third value to *exec*, *prg*, in order to give the turn to the event *Progress*. Therefore, our model follows the following structure: $init; (ctrl; prg; plant)^*$, where *init* represents the *INITIALISATION* event.

The *INVARIANTS* part defines four predicates (see Figure 5.6). Invariants $inv1$ and $inv2$ specify respectively the types of the variables $ctrlV$ and *exec*. Since the time progresses between the control and the plant parts, Invariants $inv3$ and $inv4$ are defined to guarantee that the physical part updates the continuous variables between 0 and the last progression of time. The *INITIALISATION* event (see Figure 5.7) specifies the initial values of each continuous and discrete variables. At $t = 0$, the system gives the turn to the controller to update the value of the controlled variable $ctrlV$ depending on the value of $plantV$ which is initialized to $plantV\,Init$.

INVARIANTS
inv1: $ctrlV \in \mathbb{R}$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \Rightarrow dom(plantV) = [0, t]$
inv4: $exec = plant \Rightarrow t \notin dom(plantV)$

Figure 5.6: EventTriggered INVARIANTS.

To make time evolve according to the formula evt_trig such that the physical part does not go beyond the boundaries of the safety envelope, we refine the event *Progress* (see Figure 5.8) by adding guards to specify that: (1) it is the turn of the event *Progress* to execute (*grd1*); (2)when the system is in a normal mode($ctrlV \notin evade\_values$), the value of t_1 must satisfy the formula evt_trig (*grd3*). The parts added by refinement are written in blue.

INITIALISATION:
THEN
act1 : $t := 0$
act2 : $plantV := \{0 \mapsto plantVInit\}$
act3 : $ctrlV :\in \mathbb{R}$
act4 : $exec := ctrl$
END

Figure 5.7: EventTriggered INITIALISATION.

EVENT Progress
REFINES Progress
ANY t_1
WHERE
$grd1 : exec = prg$
grd2 : $t_1 \in TIME \wedge t < t_1 \wedge t_1 - t \geq sigma$
$grd3 : ctrlV \notin evade\_value \Rightarrow evt\_trig(plantV(t), t_1 - t, ctrlV) = TRUE$
THEN
act1 : $t := t_1$
$act2 : exec := plant$
END

Figure 5.8: EventTriggered Progress.

To model the evolution of the physical part, we refine the *Plant* of the *ContSystem_M* machine by replacing the abstract differential equation e with that defined for a function denoted f_evol_plantV (see Figure 5.9). Function f_evol_plantV describes the evolution of the state variable $plantV$ according to the system discrete state. Regarding the evolution of the control part, two new events are added: *Ctrl_normal* and *Ctrl_evade* (see Figure 5.10).

EVENT Plant **REFINES** Plant
ANY plant1
WHERE
$grd1 : exec = plant$
grd2 : $plant1 \in [0, t] - dom(plantV) \rightarrow S$
grd3 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
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 : $plantV := plantV \lhd plant1$
$act2 : exec := ctrl$
END

Figure 5.9: EventTriggered Plant.

- *Ctrl_normal* event represents the *normal* mode. It is triggered when it is the turn of the controller ($exec = ctrl$) and when it exists a value *nrml_value* of $ctrlV$, different from the system evade values, for which the formula *safe* is true; it then gives the turn to the event *Progress*.

- *Ctrl_evade* event represents the *evade* mode. It assigns the parameter *evade_val* to

EVENT Ctrl_normal
ANY *nrml_value*
WHERE
grd1 : $exec = ctrl$
grd2 : $nrml\_value \in \mathbb{R}$
grd3 : $nrml\_value \notin evade\_value \Rightarrow safe(plantV(t), nrml\_value) = TRUE$
THEN
act1 : $ctrlV := nrml\_value$
act2 : $exec := prg$
END

EVENT Ctrl_evade
ANY *evade_val*
WHERE
grd1 : $exec = ctrl$
grd2 : $evade\_val \in evade\_value$
THEN
act1 : $ctrlV := evade\_val$
act2 : $exec := prg$
END

Figure 5.10: EventTriggered Ctrl_normal and Ctrl_evade.

the control variable *ctrlV* and gives the turn to the event *Progress*. The evade value must be chosen in the set of the system evade values.

5.4.2 Generic TimeTriggered Model

This model refines the previous model to get a system corresponding to the *TimeTriggered* model of Kopetz, also specified with dR$\mathcal{L}$ (*Model 3.2* defined in Section 3.2.2.1). As mentioned earlier, the sensors of a *TimeTriggered* model take periodic measurements of physical state variables and its controller executes each time those sensor updates are taken. *TimeTriggered* model is composed of the context *TimeTriggered_Ctx* and the machine *TimeTriggered_M*.

5.4.2.1 Context TimeTriggered_Ctx

Context *TimeTriggered_Ctx* depicted by Figure 5.11 extends context *EventTriggered_Ctx* by adding two new constants:

- *epsilon*: specifies a symbolic duration. It models the longest time between *TimeTriggered* sensor updates.
- $safeEpsilon$: guarantees that the controller will make a choice that will be safe for up to epsilon time.

5.4.2.2 Machine *TimeTriggered_Ctx*

The main difference between the *Event* and *TimeTriggered* models is in the modeling of the progression of time. The longest time between *TimeTriggered* sensor updates is bounded by the symbolic duration *epsilon*. Therefore, the controller can execute at least every *epsilon* time. For this purpose, we refine the event *Progress_event* by adding the formula $t' - t \leq$ *epsilon* (see Figure 5.12). This formula states that the time cannot progress by more than *epsilon* time units.

CONTEXT TimeTriggered_Ctx
EXTENDS EventTriggered_Ctx
CONSTANTS $epsilon$, $safeEpsilon$
AXIOMS
axm1: $epsilon \in TIME \land sigma \leq epsilon$
axm2: $safeEpsilon \in (S \times \mathbb{R} \rightarrow BOOL)$
axm3: $epsilon > 0$
END

Figure 5.11: Context TimeTriggered_Ctx.

EVENT Progress_time
REFINES Progress
ANY t_1
WHERE
grd1: $exec = prg$
grd2: $t_1 \in TIME \land t < t_1 \land t_1 - t \geq sigma \land t_1 - t \leq epsilon$
grd3: $ctrlV \notin evade\_value \Rightarrow evt\_trig(plantV(t), t_1 - t, ctrlV) = TRUE$
THEN
act1: $t := t_1$
act2: $exec := plant$
END

Figure 5.12: Event Progress_time.

Since the controller of a *TimeTriggered* model must make a choice that will be safe for up to *epsilon* time, we define a new safety envelope named *safeEpsilon* ($safe_\epsilon(plantV, ctrlV)$ in dR$\mathcal{L}$) in the context *TimeTriggered_Ctx*. Then, in the event *Ctrl_normal_time* that refines *Ctrl_normal*, we add a guard to ensure that *safeEpsilon* is true when a non evade value is chosen for *ctrlV* (see *grd*4 of Figure 5.13).

EVENT Ctrl_normal_time
REFINES Ctrl_normal
ANY $nrml\_value$
WHERE
grd1: $exec = ctrl$
grd2: $nrml\_value \in \mathbb{R}$
grd3: $nrml\_value \notin evades\_value \Rightarrow safe(plantV(t), nrml\_value) = TRUE$
grd4: $nrml\_value \notin evades\_value \Rightarrow safeEpsilon(plantV(t), nrml\_value) = TRUE$
THEN
act1: $ctrlV := nrml\_value$
act2: $exec := prg$
END

Figure 5.13: Event Ctrl_normal_time.

Let us remark that contrary to the models described in [5], we kept the guards related to the formula *evt_trig* (guard *grd*3 of the event *Progress_time*) and *safe* (guard *grd*3 of the event *Ctrl_normal_time*). Removing them does not enable us to discharge the associated POs since it is not possible to establish, on the generic models, that they are induced by the invariant and the other guards. However it is generally not a good practice to leave proof obligation undischarged because this makes the correctness of the Event-B development questionable. In Chapter 8, on specific case studies, we show how such guards are removed, which give rise to refinement proof obligations to discharge.

5.4.3 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 $prop \in \mathbb{R}^n \rightarrow BOOL$ is defined in the context *EventTriggered_Ctx*, where n denotes the number of variables occurring in the property. Then an invariant is added in the machine *EventTriggered_M*, where *plantV* will be replaced by the specific continuous variable x that permits to cover all the moments from the beginning until the current time and *prop(plantV(x))* is replaced by the specific safety property. This formula expresses that the safety property of the system shall be satisfied in the time interval $[0, t]$ which denotes the domain of *plantV*.

$$inv5 : \forall x \cdot x \in dom(plantV) \Rightarrow prop(plantV(x)) = TRUE$$

To discharge the PO generated for this invariant, we added the following guard to the event *Plant* that states that the new values of *plantV*, that is *plant*1, verify the property:

$$grd6 : \forall xx \cdot xx \in dom(plant1) \Rightarrow prop(plant1(xx)) = TRUE.$$

This guard will be removed on a specific case to generate a proof obligation that aims at proving that this guard is actually satisfied. We give more details in Chapter 8.

5.5 Correctness of the Generic Models

Table 5.1 gives the statistics of the proof obligations generated to ensure the correctness of the generic models of our approach. It is noticeable that 45% of them were automatically discharged. The remaining proof obligations are discharged using the automatic/interactive provers of the RODIN platform (Version 3.5.0) and the theories developed in [10].

Table 5.1: RODIN Proof Statistics for the Generic Models

Generic_Models	**Total**	**Automatic**	**Interactive**
ContSystem_M	8	1	7
EventTriggered_M	22	14	8
TimeTriggered_M	4	1	3

The platform RODIN automatically generates proof obligations for properties that need to be proven on a given EVENT-B machine or context. Each proof obligation has a name that identifies where it was generated and also its goal. Some of these proof obligations must be manually discharged using theories and hypotheses defined in the associated machines and contexts. The theory *Reals* does not handle all the properties of real numbers. For this purpose, we define a context named *Theorems* (see Figure 5.14) that contains all the properties and theorems needed to discharge the proof obligations generated by the RODIN platform to prove our generic and specific models.

- Axiom 1: specifies that the addition preserves the order of real numbers: $a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$.
- Axiom 2: specifies that the operator *mult* ($\times$) preserves the order of real numbers: $0 \leq a \wedge 0 \leq b \wedge 0 \leq c \wedge 0 \leq d \wedge a \leq b \wedge c \leq d \Rightarrow a \times c \leq b \times d$.
- Axiom 3: let a, b and c be real numbers, if $a \leq b \wedge b \leq c$ then $a \leq c$.

- Axiom 4: encodes the following property defined using the operators *minus* $(-)$ and *mult* $(\times)$: $(a^2) - (b^2) = (a + b) \times (a - b)$.
- Axiom 5: specifies the following property defined for the operator *minus*: $-a = 0 - a$.
- Axiom 6: specifies the following property defined for the operators *divide*, *times* and *plus*: $a = \frac{1}{2} \times a + \frac{1}{2} \times a$
- Axiom 7: specifies the following property defined for the operator *inverse* $\frac{1}{a \times b} = \frac{1}{a} \times \frac{1}{b}$.

CONTEXT Theorems
AXIOMS
axm1 : $\forall a, b, c, d \cdot a \leq b \land c \leq d \Rightarrow a + c \leq b + d$
axm2 : $\forall a, b, c, d \cdot 0 \leq a \land 0 \leq b \land\ 0 \leq c \land 0 \leq d \land a \leq b \land c \leq d$
$\Rightarrow a \times c \leq b \times d$
axm3 : $\forall a, b, c \cdot a \leq b \land b \leq c \Rightarrow a \leq c$
axm4 : $\forall a, b \cdot a \in \mathbb{R} \land b \in \mathbb{R} \Rightarrow (a^2 - b^2) = (a + b) \times (a - b)$
axm5 : $\forall a \cdot a \in \mathbb{R} \Rightarrow -a = 0 - a$
axm6 : $\forall a \cdot a \in \mathbb{R} \Rightarrow a = \frac{1}{2} \times a + \frac{1}{2} \times a$
axm7 : $\forall a, b \cdot a \in \mathbb{R} \land b \in \mathbb{R} \land a \times b \in \mathbb{R}^* \Rightarrow \frac{1}{a \times b} = \frac{1}{a} \times \frac{1}{b}$
END

Figure 5.14: Theorems.

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 *evt_trig* in the events *Progress*, *Ctrl_normal* and *Ctrl_normal_time*, the refinement proofs are rather simple and related mainly to the type checking of the different variables and the feasibility of the events *Progress* and *Progress_time* since we have to exhibit a value of t_1 that verifies the stated conditions (see *grd2* of Figure 5.8). In the following we describe the most relevant proof obligations generated for each generic model:

- *ContSystem Model:* $PO1$ is also generated for the event *Progress* to prove that the action which updates the value of t by t' is feasible. $PO2$ is generated for the event *Plant* to verify that the action which updates the value of $plantV$ using the parameter $plant1$ verifies the type of $plantV$.

  PO1: $\exists\, t' \cdot t' \in TIME \land t < t' \land t' - t \geq sigma$
  PO2: $plantV \lhd plant1 \in [0, t] \pfun S$

- *EventTriggered Model:* PO3 is a well-definedness proof obligation generated for the event *Progress* due to adding the guard $grd3$ which guarantees that the new value of t satisfies the formula *evt_trig*. This PO is discharged using the properties of the theory *Reals* and the invariants defined in the *EventTriggered_M* machine. PO4 is generated for the event *Plant* to prove that it satisfies the following invariant: $exec \neq plant \Rightarrow dom(plantV) = [0, t]$. This PO is discharged by adding as hypothesis the definition of $plantV$, $plantV \in [0, t] \pfun RReal$. PO5 is also a well-definedness proof obligation generated for the event *Ctrl_normal* to prove that it satisfies the safety envelope represented by the formula $safe$.

  PO3: $ctrlV \notin evade\_value \implies t \in dom(plantV) \land plantV \in \mathbb{R} \pfun S$
  $\land plantV(t) \mapsto (t1 - t) \mapsto ctrlV \in dom(evt\_trig) \land$
  $evt\_trig \in S \times TIME \times \mathbb{R} \pfun BOOL$
  PO4: $ctrl \neq plant \Rightarrow dom(plantV \lhd plant1) = [0, t]$

PO5: $nrml\_value \notin evade\_value \implies t \in dom(plantV) \land$
$plantV \in \mathbb{R} \pfun \mathbb{R} \times \mathbb{R} \land plantV(t) \mapsto$
$nrml\_value \in dom(safe) \land safe \in S \times \mathbb{R} \pfun BOOL$

- *TimeTriggered Model:* PO6 is a Well-definedness proof obligation generated for the event *Ctrl_normal_time* for replacing formula $safe$ by $safeEpsilon$.

PO6: $nrml\_value \notin evade\_value \implies t \in dom(plantV) \land$
$plantV \in \mathbb{R} \pfun S \land plantV(t) \mapsto nrml\_value \in$
$dom(safeEpsilon) \land safeEpsilon \in \mathbb{R} \times \mathbb{R} \pfun BOOL$

5.6 Conclusion

In this chapter, we have presented a proof-based approach that uses the formal method Event-B and its refinement technique to specify and prove the refinement between *Event* and *TimeTriggered* systems. We have defined two generic templates for these systems that represent hybrid systems as hybrid programs, *Event* and *TimeTriggered* models described in Section 5.4. We have also introduced a more abstract level, the *ContSystem* model, that specifies the continuous aspects of hybrid systems, adapted from the approach of [10]. This permits to cope with the proof complexity by decomposing the proof obligations, such that in the abstract model we only deal with the proof obligations related to the continuous aspects of the system and in the refined model we will have the proof obligations related to the safety properties of the controlled system.

In [1], we presented our first attempt to formalize the generic structure. However, the formal models have proved to be not suitable to deal with systems with complex properties. Moreover, the approach is abstract and did not consider the resolution of ODEs. Consequently, it cannot be instantiated for the verification of a specific application. This approach also consists on three generic models: *System*, *Event* and *TimeTriggered* models. It reuses the abstract model *System* presented in Section 3.3.2. The progression of time, the physical part are described respectively in [1] by the following events: *Progress* (see Section 3.3.2), *Plant* (see Figure 5.15).

- Event *Progress* is defined as in the Section 3.3.2. It models the progression of time in Event-B. The major limit of this event is that it does not specify that the time must not evolve beyond a value that makes the physical part cross the boundary of the safety property. For this purpose, we added the guard that uses the *evt_trig* formula to check if in the new period of control the system remains safe (Figure 5.8)

- Event *Plant* represents the evolution of the continuous part represented by the state variable *plantV*. It refines the event *Behave* (see Section 3.3.2) by replacing the abstract differential equation e by that defined for a function denoted *f_evol_plantV* in order to model the specific ODE. The function *f_evol_plantV* describes the evolution of the state variable *plantV* according to the system discrete state. The major limit of this event is that it assigns the new values of the continuous variables without checking if they satisfy the safety properties of the modeled system.

The generic approach proposed in this thesis extends and improves [1] with the following contributions:

- The new model introduced is more faithful to the *Event* and *TimeTriggered* patterns on which it is based; this facilitates the proof of any safety property.

- To model the alternation between the progression of time, the control and the physical parts, the approach of [1] followed the following structure: $init; (ctrl; plant; prg)^*$.

EVENT Plant
REFINES Behave
WHERE
grd1 : $ode(f\_evol\_plantV(ctrlV), plantV(t), t) \in DE(S)$
grd2 : $Solvable([t, \infty[, ode(f\_evol\_plantV(ctrlV), plantV(t), t))$
grd3 : $exec = plant$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), plantV(t), t)$
THEN
act1 : $plantV :| plantV' \in (\mathbb{R}^+ \rightarrow S) \land AppendSolutionBAP($
$ode(f\_evol\_plantV(ctrlV), plantV(t), t), \mathbb{R}^+, [0, t[, [t, \infty[, plantV, plantV')$
act2 : $exec := prg$
END

Figure 5.15: Event Plant [1].

The controlled part followed by the physical part followed by the progression of time. In the new model, we let the time progresses before executing the physical part, as if they are executed in parallel, which facilities the prove of the safety properties.

- The approach of [1] remains abstract regarding the prove of safety properties. We fixed that by calculating the new values of the continuous variables between 0 and the current time t. Additionally, we define the generic parameter $plant1$ which is used to check that these new values satisfy the safety properties.

In comparison to the approach of dR$\mathcal{L}$, Event-B refinement is based on the execution traces starting from the initial state, that is, to prove that a concrete Event-B machine refines an abstract one, we have to establish that the set of execution traces of the concrete one is included in that corresponding to the abstract one. However, the refinement of dR$\mathcal{L}$ is based on reachable states, that is, a hybrid program α refines another hybrid program β ($\alpha \leq \beta$), *iff* the set of reachable states from a state s following the transitions of α is included in the set of reachable states from the same state s following some transitions of β. Contrary to dR$\mathcal{L}$, Event-B refinement can be used to introduce and prove new properties, which is different from typical usage in dR$\mathcal{L}$.

Both Event-B and dR$\mathcal{L}$ refinements allow preserving the safety properties of the abstract model. This is ensured in dR$\mathcal{L}$ through combining refinement relations and modalities. Despite the several features of dR$\mathcal{L}$'s refinement, computing reachable states for non linear systems requires solving non-linear real arithmetic problems which is difficult in general. Since, dR$\mathcal{L}$ refinement is based on reachable states it does not preserve the safety properties on the traces and it is weaker than the Event-B refinement. Moreover, dR$\mathcal{L}$ is not supported by any prover, therefore proving manually the correctness of systems is error-prone in the case of complex systems especially for systems with more than two modes. Through using Event-B, we have succeed to overcome this limitation since its support tools aid in discharging proof obligations either automatically or interactively by guiding the prover in applying the adequate deductive/rewriting rules.

The major limitation in using Event-B to model and verify hybrid systems is the absence of support for the continuous aspects of CPSs, such as continuous time and differential equations. To overcame this limitation the approach proposed in [10] defines an Event-B theory that includes different kinds of differential equations. This is why we have adapted the abstract model of this approach, so that it becomes possible to reason on hybrid programs in Event-B. To solve concrete ODEs, the approach of [10] consists in using the approximation concept during the refinement process. We have chosen a different approach by coupling Event-B with a differential equation solver in order to solve ODEs using Rodin (see Chapter 7).

Chapter 6

Instantiating the Generic Approach

Contents

In the previous chapter, we presented a generic correct-by-construction approach for modeling hybrid systems using the formal method Event-B and its supported tool Rodin. The generic approach consists of three generic models. To design specific systems following the generic approach, two strategies can be applied to instantiate the generic models. For each strategy, we present a set of instantiation rules defined to systematically build the models of any specific application.

Section 6.1 describes the first strategy that consists in starting by an abstract model of the specific system obtained by refining the abstract generic *ContSystem* model. Section 6.2 describes the second one that consists in directly instantiating by refinement the *EventTriggered* model. Last, Section 6.3 discusses the main advantages and limitations of each strategy as well as the main differences between both strategies.

6.1 Instantiation from the ContSystem Level

The instantiation starts by refining the abstract generic *ContSystem* model. The obtained model is then refined to design a specific *EventTriggered* model that instantiates the generic *EventTriggered* model. Last, the specific *EventTriggered* model is refined to obtain a specific *TimeTriggered* model that instantiates the generic *TimeTriggered* model as depicted by Figure 6.1.

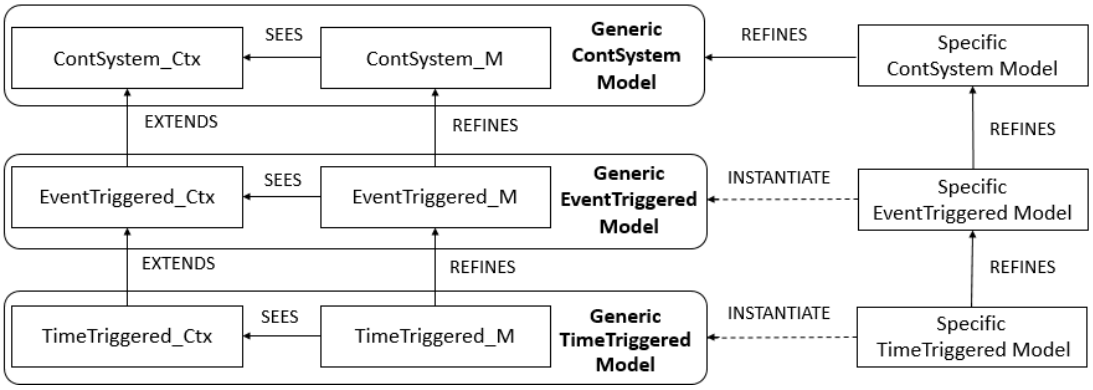

Figure 6.1: First Strategy: Instantiation from the *ContSystem* Level.

6.1.1 Instantiating the Generic ContSystem Model

The generic *ContSystem* model is refined by instantiating the generic parameters following a set of rules defined below. The specific *ContSystem* model is composed of two elements, a *SpecificContSystem_Ctx* context and a *SpecificContSystem_M* machine. *SpecificContSystem_Ctx* given in Figure 6.2 extends the *ContSystem_Ctx* context given in Figure 5.2. It is instantiated following the set of rules *Rule_CS_Ctx_i* below:

- *Rule_CS_Ctx_1*: instantiates the constant S defined in the context *ContSystem_Ctx*. This constant depends on the number of continuous variables of the system to be modeled. It must be instantiated in the specific *ContSystem* model by replacing n in $axm1$ of Figure 5.2 by the number of specific continuous variables.

- *Rule_CS_Ctx_2*: instantiates the constant *plantVInit* defined in the generic *ContSystem* by replacing its value by the specific initial value of the continuous variable denoted $specPlantVinit_i$ (see $axm2$ of Figure 6.2). If the modeled system is composed of two or more continuous state variables, it is necessary to add n constants defined in $\mathbb{R}$ or $\mathbb{R}^+$ depending on the characteristics of the corresponding continuous variable, where n represents the number of continuous variables to be treated (see $axm1$ of Figure 6.2).

- *Rule_CS_Ctx_3*: defines all the constants and properties needed to model and prove the modeled hybrid system. It defines l constants cst_1,...,cst_l in the *CONSTANTS* clause and their properties in the *AXIOMS* clause (see $axm3$ of Figure 6.2).

- *Rule_CS_Ctx_4*: defines the ordinary differential equations of the modeled system using the function *f_evol_plantV* (see $axm4$ and $axm5$ of Figure 6.2). It defines n differential equations eq_i, where n denotes the number of specific continuous variables $specPlantVinit_i$. *f_evol_plantV* binds each continuous measurement, y_i, with its associated differential equation, eq_i, using a lambda expression, see $axm5$. It depends on the controlled measurement of the modeled system which is specified by x. A differential equation eq_i must be expressed in terms of t, y_i and x.

The specific context *SpecificContSystem_Ctx* associated with the *Smart Heating* case study is represented by the context *Specific_Heater_Ctx* depicted by Figure 6.3. The continuous behavior of this case study is represented by a single continuous variable T denoting the measured temperature. Therefore, rule *Rule_CS_Ctx_1* instantiates S in the associated generic *ContSystem_Ctx* of Figure 5.2 by $\mathbb{R}$ and *Rule_CS_Ctx_2* defines a single initial continuous constant denoted $T0$ (see $axm1$ and $axm2$ of Figure 6.3). *Rule_CS_Ctx_3* defines two constants, the maximum value of the temperature *T_max* and the minimum value *T_min* as well as their properties specified by the axiom $axm3$ of Figure

CONTEXT SpecificContSystem_Ctx **EXTENDS** ContSystem_Ctx
CONSTANTS $specPlantVinit_1$, ... , $specPlantVinit_n$, cst_1 , ... , cst_l ,
f_evol_plantV
AXIOMS
axm1 : $\bigwedge_{i=1..n} specPlantVinit_i \in \mathbb{R}$
axm2 : $(specPlantVinit_1, ..., specPlantVinit_n) = plantVInit$
axm3 : $\bigwedge_{i=1..l} prt_i$
axm4 : $f\_evol\_plantV \in \mathbb{R} \rightarrow (TIME \times S \rightarrow S)$
axm5 : $\forall x \cdot x \in \mathbb{R} \implies (f\_evol\_plantV(x) = (\lambda\, t \mapsto (y_1 \mapsto .... \mapsto y_n) \cdot t \in TIME$
$\wedge\ (y_1 \mapsto .... \mapsto y_n) \in S \mid (eq_1, ..., eq_n)))$
END

Figure 6.2: SpecificContSystem_Ctx.

6.3. The differential equation of the *Smart Heating* case study is represented by a single formula $\frac{dT}{dt} = x$, where x represents the controlled measurement. Rule *Rule_CS_Ctx_4* is then applied by replacing S by its specific value in $axm4$ and by instantiating the specific continuous equation by x (see $axm5$ of Figure 6.3).

CONTEXT Specific_Heater_Ctx **EXTENDS** ContSystem_Ctx
CONSTANTS $T0$, T_max , T_min , f_evol_plantV
AXIOMS
axm1 : $T0 \in \mathbb{R}^+$
axm2 : $T0 = plantVInit$
axm3 : $T0 < T\_max \wedge T0 > T\_min \wedge T\_max \in \mathbb{R} \wedge$
$T\_max > T\_min \wedge T\_min \in \mathbb{R} \wedge T\_min > 0$
axm4 : $f\_evol\_plantV \in \mathbb{R} \rightarrow (TIME \times \mathbb{R} \rightarrow \mathbb{R})$
axm5 : $\forall x \cdot x \in \mathbb{R} \implies (f\_evol\_plantV(x) = (\lambda t \mapsto T \cdot t \in TIME \wedge T \in \mathbb{R} \mid x))$
END

Figure 6.3: Specific_Heater_Ctx.

Machine *SpecificContSystem_M* refines the generic machine *ContSystem_M*, presented in A.2, by applying the set of rules *Rule_CS_M_i* defined below:

- *Rule_CS_M_1*: instantiates the generic continuous variable *plantV* by defining n invariants used to define the specific continuous variables as follows: $specPlantV_i \in [0, t] \nrightarrow \mathbb{R}^+$ or $specPlantV_i \in [0, t] \nrightarrow \mathbb{R}$ depending on the nature of the continuous variable to be measured. This rule also adds an invariant used to link *plantV* to the specific continuous variables using the operator *bind* of *DiffEq* (see $inv1$ and $inv2$ of Figure 6.4).

- *Rule_CS_M_2*: assigns the initial value $specPlantVinit_i$ to the specific continuous variable $specPlantV_i$ (see $act2$ of Figure 6.5).

- *Rule_CS_M_3*: refines the generic event *Plant* of Figure 5.15 by replacing the generic continuous variable *plantV* by the specific ones. In addition, the generic parameter *plant*1 is replaced by the specific ones by adding a witness (see WITH clause of Figure 6.5).

Machine *Specific_Heater_M* refines the generic *ContSystem_M* machine by applying the three rules defined above. *Rule_CS_M_1* is applied by defining the invariants $inv1$ and $inv2$ described in Figure 6.6. $inv1$ specifies the definition of the temperature T, and $inv2$ replaces the generic variable *plantV* by T. *Rule_CS_M_2* is applied by initialising the value of T at the instant $t = 0$ by $T0$ in $act2$ of Figure 6.6.

MACHINE SpecificContSystem_M
REFINES ContSystem_M
SEES SpecificContSystem_Ctx
VARIABLES t, $specPlantV_1$, ..., $specPlantV_n$
INVARIANTS
inv1: $\bigwedge_{i=1..n} specPlantV_i \in [0,t] \nrightarrow \mathbb{R}$
inv2: $plantV = bind(specPlantV_1, bind(specPlantV_2, bind(...)))$
EVENTS
INITIALISATION
THEN
act1: $t := 0$
act2: $specPlantV_i := \{0 \mapsto specPlantVinit_i\}$
END

Figure 6.4: Header of SpecificContSystem_M.

EVENT Plant
REFINES Plant
ANY e, $specPlant1_1$, ..., $specPlant1_n$
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable([0,t] - dom(specPlantV), e)$
grd3: $\bigwedge_{i=1..n} specPlant1_i \in [0,t] - dom(specPlantV_i) \rightarrow \mathbb{R} \wedge$
$AppendSolutionBAP(e, [0,t] - dom(specPlantV_i),$
$[0,t] - dom(specPlantV_i), specPlant1_i)$
WITH
$plant1 : plant1 = bind(specPlant1_1, bind(specPlant1_2, bind(...)))$
THEN
act1: $specPlantV_i := specPlantV_i \lhd\!\!+ specPlant1_i$
END

Figure 6.5: Specific Plant.

MACHINE Specific_Heater_M
REFINES ContSystem_M
SEES Specific_Heater_Ctx
VARIABLES t, T
INVARIANTS
inv1: $T \in [0,t] \nrightarrow \mathbb{R}^+$
inv2: $T = plantV$
EVENTS
INITIALISATION
THEN
act1: $t := 0$
act2: $T := \{0 \mapsto T0\}$
END

Figure 6.6: Header of Specific_Heater_M.

Event *Plant* is refined by applying rule *Rule_CS_M_3* that replaces the generic continuous variable *plantV* by T and the generic parameter *plant*1 by $T1$ defined in the *ANY* clause of Figure 6.7.

EVENT Plant
REFINES Plant
ANY e, $T1$
WHERE
grd1 : $e \in DE(\mathbb{R})$
grd2 : $Solvable([0,t] - dom(T), e)$
grd3 : $T1 \in [0,t] - dom(T) \rightarrow \mathbb{R}^+ \wedge$
$AppendSolutionBAP(e, [0,t] - dom(T), [0,t] - dom(T), T1)$
WITH $plant1 : plant1 = T1$
THEN
act1 : $T := T \Leftarrow T1$
END

Figure 6.7: Specific Heater Plant.

6.1.2 Instantiating the Generic EventTriggered Model

The specific *EventTriggered* model refines the specific *ContSystem* model and instantiates the generic *EventTriggered* model. It consists of a specific context, *SpecificEventTriggered_Ctx1* depicted by Figure 6.8, and a specific machine *SpecificEventTriggered_M1*. Context *SpecificEventTriggered_Ctx1* extends the context *SpecificContSystem_Ctx* and instantiates the generic context *EventTriggered_Ctx* of Figure 5.5 by following the rules *Rule_ET1_Ctx_i* described below:

- *Rule_ET1_Ctx_1*: defines the constant set $EXEC$ and its elements, prg, *plant* and *ctrl*, that are used to describe the alternation between the control and the physical parts as well as the progression of time (see $axm1$ of Figure 6.8).
- *Rule_ET1_Ctx_2*: instantiates the system safety envelopes by defining k constant formulas, $safe_1$, ..., $safe_k$. For each safety envelop $safe_i$, two axioms are defined. The first axiom specifies the type of the formula (see $axm2$ of Figure 6.8). The second one, called $SafeForm_i$, specifies the property to be modeled. This latter depends on the continuous measurements $y_1, ..., y_n$ and the controlled measurement x (see $axm4$ of Figure 6.8).
- *Rule_ET1_Ctx_3*: instantiates formula evt_trig_i by defining k constant formulas, evt_trig_1, ..., evt_trig_k. Two axioms are defined for each event trigger property (see axioms $axm3$ and $axm5$ of Figure 6.8). These formulas depend on the continuous measurements $y_1, ..., y_n$ and the controlled measurement x as well as the duration between two periods of control.
- *Rule_ET1_Ctx_4*: instantiates the concrete values of the controlled measurement by defining k constants in $\mathbb{R}$ (see $axm6$ of Figure 6.8). These constants will be used as values of the controlled variable $ctrlV$.
- *Rule_ET1_Ctx_5*: instantiates evade values by those associated with the specific case study (see $axm7$ of Figure 6.8).
- *Rule_ET1_Ctx_6*: defines a constant, called *prop*, used to specify the system safety property (see $axm8$ of Figure 6.8). *prop* is instantiated by defining the specific *PropForm* (see $axm9$ of Figure 6.8).

The specific *EventTriggered* model associated with the *Smart Heating* system is represented by the *Event_Heater_Ctx* context depicted by Figure 6.9 and the *Event_Heater_M* machine. *Event_Heater_Ctx* extends the *Specific_Heater_Ctx* context and instantiates the generic *EventTriggered_Ctx*. The instantiation starts by defining the set $EXEC$ and its elements following *Rule_ET1_Ctx_1*. As described in Section 4.3, two safety envelops

CONTEXT SpecificEventTriggered_Ctx1
EXTENDS SpecificContSystem_Ctx
SETS $EXEC$
CONSTANTS $ctrl$, $plant$, prg, $safe_1$, ..., $safe_k$, evt_trig_1, ..., evt_trig_k, val_1, ..., val_k, $evade\_value_1$, ..., $evade\_value_k$, $prop$
AXIOMS
axm1: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm2: $\bigwedge_{i=1..k} safe_i \in (S \times \mathbb{R}) \rightarrow BOOL$
axm3: $\bigwedge_{i=1..k} evt\_trig_i \in (S \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm4: $\bigwedge_{i=1..k} safe_i = (\lambda(y_1 \mapsto .... \mapsto y_n) \mapsto x \cdot (y_1 \mapsto .... \mapsto y_n) \in S \land x \in \mathbb{R} \mid bool(SafeForm_i))$
axm5: $\bigwedge_{i=1..k} evt\_trig_i = (\lambda(y_1 \mapsto .... \mapsto y_n) \mapsto t1 \mapsto x \cdot (y_1 \mapsto .... \mapsto y_n) \in S \land t1 \in TIME \land x \in \mathbb{R} \mid bool(TrigForm_i))$
axm6: $\bigwedge_{i=1..z} val_i \in \mathbb{R}$
axm7: $\bigwedge_{i=1..k} evade\_value_i \subseteq \mathbb{R}$
axm8: $prop \in \mathbb{R}^n \rightarrow BOOL$
axm9: $prop = (\lambda(y_1 \mapsto .... \mapsto y_n) \cdot (y_1 \mapsto .... \mapsto y_n) \in \mathbb{R}^n \mid bool(PropForm))$
END

Figure 6.8: SpecificEventTriggered_Ctx1.

need to be defined (*Rule_ET1_Ctx_2*), see *axm*2, *axm*4, *axm*6 and *axm*8 of Figure 6.9. For each safety envelop, we define an event trigger formula (*Rule_ET1_Ctx_3*), see *axm*3, *axm*5, *axm*7 and *axm*9 of Figure 6.9. The continuous evolution of the *Smart Heating* case study is described by two constants, *temp* and $-temp$, that represent the values val_i of the controlled variable *ctrlV* (*Rule_ET1_Ctx_4*). *Rule_ET1_Ctx_5* instantiates the set of evade values (axioms *axm*11 and *axm*12 of Figure 6.9). Last *Rule_ET1_Ctx_6* instantiates the safety property, $T\_min \leq T \leq T\_max$, using the constant *prop* (see *axm*13 and *axm*14 of Figure 6.9).

CONTEXT Event_Heater_Ctx
EXTENDS Specific_Heater_Ctx
SETS EXEC
CONSTANTS $ctrl$, $plant$, prg, $safe_1$, evt_trig_1, $safe_2$, evt_trig_2, $evade\_value_1$, $evade\_value_2$, $temp$, $prop$
AXIOMS
axm1: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm2: $safe_1 \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm3: $evt\_trig_1 \in (\mathbb{R} \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm4: $safe_2 \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm5: $evt\_trig_2 \in (\mathbb{R} \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm6: $safe_1 = (\lambda T \mapsto x \cdot T \in \mathbb{R} \land x \in \mathbb{R} \mid bool(T < T\_max))$
axm7: $evt\_trig_1 = (\lambda\, T \mapsto t1 \mapsto x \cdot T \in \mathbb{R} \land\ t1 \in TIME \land x \in \mathbb{R} \mid bool(T + x \times t1 \leq T\_max))$
axm8: $safe_2 = (\lambda\, T \mapsto x \cdot T \in \mathbb{R} \land x \in \mathbb{R} \mid bool(T > T\_min))$
axm9: $evt\_trig_2 = (\lambda T \mapsto t1 \mapsto x \cdot T \in \mathbb{R} \land t1 \in TIME \land x \in \mathbb{R} \mid bool(T + x \times t1 \geq T\_min))$
axm10: $temp \in \mathbb{R} \land temp > 0$
axm11: $evade\_value_1 \subseteq \mathbb{R} \land evade\_value_1 = \{-temp\}$
axm12: $evade\_value_2 \subseteq \mathbb{R} \land evade\_value_2 = \{temp\}$
axm13: $prop \in \mathbb{R} \rightarrow BOOL$
axm14: $prop = (\lambda T \cdot T \in \mathbb{R} \mid bool(T \leq T\_max \land T \geq T\_min))$
END

Figure 6.9: Event_Heater_Ctx.

Machine *SpecificEventTriggered_M1* refines the *SpecificContSystem_M* machine by instantiating the generic *EventTriggered_M* machine. The instantiation is done following the rules *Rule_ET1_M_i* defined below:

- *Rule_ET1_M_1*: defines the discrete variables *ctrlV* and *exec* with their typing in the *INVARIANTS* clause, where *setVal* denoting the set of possible values of *ctrlV* (see *inv*1 and *inv*2 of Figure 6.10).
- *Rule_ET1_M_2*: instantiates the invariants *inv*3 and *inv*4 of Figure 5.6 by replacing the generic continuous variable *plantV* by the specific ones, $specPlantV_1$, ..., $specPlantV_n$. These invariants are required to prove the specific proof obligations. This rule also adds the system safety property as an invariant using the constant *prop* (see *inv*5 of Figure 6.10).

INVARIANTS

inv1 : $ctrlV \in setVal$
inv2 : $exec \in EXEC$
inv3 : $\bigwedge_{i=1..n} exec \neq plant \implies dom(SpecPlantV_i) = [0, t]$
inv4 : $\bigwedge_{i=1..n} exec = plant \implies t \notin dom(specPlantV_i)$
inv5 : $\forall w \cdot w \in dom(bind(specPlantV_1, bind(specPlantV_2,$
$bind(...))))) \implies prop(specPlantV_1(w), ..., specPlantV_n(w)) = TRUE$

Figure 6.10: Instantiating the INVARIANTS Clause.

- *Rule_ET1_M_3*: instantiates for each sub-formula $safe_i$, a set of evade values $evade\_value_i$ for the controlled variable *ctrlV* and a formula evt_trig_i for its boundary. Thus the generic event *Progress* is instantiated as depicted in Figure 6.11, where *grd3* states that time progresses without crossing the boundaries of the event trigger evt_trig_i of any safety envelope.

EVENT Progress
REFINES Progress
ANY t_1
WHERE
grd1 : $exec = prg$
grd2 : $t_1 \in TIME \wedge t < t_1 \wedge t_1 - t \geq sigma$
grd3 : $\bigwedge_{i=1..k}(ctrlV \notin evade\_value_i \Rightarrow$
$evt\_trig_i(specPlantV_1(t), ..., specPlantV_n(t), t_1 - t, ctrlV))$
THEN
act1 : $t := t_1$
act2 : $exec := plant$
END

Figure 6.11: Instantiating the Event Progress.

- *Rule_ET1_M_4*: instantiates the event *Plant* by replacing the generic ordinary differential equation *e* by the specific ones using the function *f_evol_plantV* (see *grd*3 of Figure 6.12). It replaces the generic ODE *e* by its specific value in the guards *grd*4 and *grd*5 of Figure 6.12. This rule also adds the safety property specified using the involved parameters $specPlant1_i$ (see *grd*6 of Figure 6.12).

- *Rule_ET1_M_5*: instantiates the generic event *Ctrl_normal* depicted by Figure 5.10 in Figure 6.13, where *grd3* checks that the chosen normal value makes all the sub-formulas $safe_i$ satisfied.

EVENT Plant
REFINES Plant
ANY $specPlant1_1, \ldots, specPlant1_n$
WHERE
grd1 : $exec = plant$
grd2 : $\bigwedge_{i=1..n} specPlant1_i \in [0,t] - dom(specPlantV_i) \rightarrow \mathbb{R}$
grd3 : $ode(f\_evol\_plantV(ctrlV), (specPlant1_1(t), ..., specPlant1_n(t)), t) \in DE(S)$
grd4 : $Solvable([0,t] - dom(bind(specPlantV_1, bind(specPlantV_2, bind(...)))),$
$ode(f\_evol\_plantV(ctrlV), (specPlant1_1(t), ..., specPlant1_n(t)), t)$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV),$
$(specPlant1_1(t), ..., specPlant1_n(t)), t),$
$[0,t] - dom(bind(specPlantV_1, bind(specPlantV_2, bind(...)))),$
$[0,t] - dom(bind(specPlantV_1, bind(specPlantV_2, bind(...)))),$
$bind(specPlant1_1, bind(specPlant1_2, bind(...))))$
grd6 : $\forall xx \cdot xx \in dom(bind(specPlant1_1, bind(specPlant1_2, bind(...))))$
$\implies prop(specPlant1_1(xx), ..., specPlant1_n(xx)) = TRUE$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), (specPlant1_1(t), ..., specPlant1_n(t)), t)$
THEN
act1 : $\bigwedge_{i=1..n} specPlantV_i := specPlantV_i \Leftarrow specPlant1_i$
act2 : $exec := ctrl$
END

Figure 6.12: Instantiating the Event Plant.

EVENT Ctrl_normal
ANY nrml_value
WHERE
grd1 : $exec = ctrl$
grd2 : $nrml\_value \in \mathbb{R}$
grd3 : $\bigwedge_{i=1..k} nrml\_value \notin evade \Rightarrow$
$safe_i(specPlantV_1(t), ..., specPlantV_n(t), nrml\_value) = TRUE$
THEN
act1 : $ctrlV := nrml\_value$
act2 : $exec := prg$
END

Figure 6.13: Instantiating the Event Ctrl_normal.

- *Rule_ET1_M_6*: instantiates the event *Ctrl_evade* of Figure 6.14. It produces n events: one for each sub-formula $safe_i$. In event $Ctrl\_evade_i$ *grd3* checks that the value chosen for sub-formula $safe_i$ makes the other safety properties satisfied.

EVENT Ctrl_evade$_i$
ANY evade_val
WHERE
grd1 : $exec = ctrl$
grd2 : $evade\_val \in evade\_value_i$
grd3 : $\bigwedge_{j=1..k \wedge j \neq i} safe_j(specPlantV_1(t), ..., specPlantV_n(t), evade\_val) = TRUE$
THEN
act1 : $ctrlV := evade\_val$
act2 : $exec := prg$
END

Figure 6.14: Instantiating the Event Ctrl_evade.

Machine *Event_Heater_M* represents the specific *EventTriggered* model associated with the *Smart Heating* system. Figure 6.15 shows the definition of the variables *ctrlV* and *exec* as stated by *Rule_ET1_M_1*. The invariants *inv*3 and *inv*4 are instantiated by replacing the generic variable, $SpecPlantV_i$, by the specific one T (*Rule_ET1_M_2*). The safety property is defined using the formula *prop* specified in the *Event_Heater_Ctx* by instantiating *inv*5 of Figure 6.10. Figure 6.16 depicts the specific event *Progress* associated with the *Smart Heating* case study. For each event trigger formula, evt_trig_1 and evt_trig_2, *grd*3 of Figure 6.11 is instantiated by replacing $SpecPlantV_i$ by T (*Rule_ET1_M_3*).

MACHINE Event_Heater_M
REFINES Specific_Heater_M
SEES Event_Heater_Ctx
VARIABLES t, T, $ctrlV$, $exec$
INVARIANTS
inv1: $ctrlV \in \{temp, -temp\}$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \implies dom(T) = [0, t]$
inv4: $exec = plant \implies t \notin dom(T)$
inv5: $\forall w \cdot w \in dom(T) \implies prop(T(w)) = TRUE$

Figure 6.15: Header of Event_Heater_M.

EVENT Progress
REFINES Progress
ANY t_1
WHERE
grd1: $exec = prg$
grd2: $t_1 \in TIME \wedge (t < t_1) \wedge (t_1 - t) \geq sigma$
grd3: $ctrlV \notin evade\_value_1 \implies evt\_trig_1(T(t) \mapsto (t_1 - t) \mapsto ctrlV) = TRUE$
grd4: $ctrlV \notin evade\_value_2 \implies evt\_trig_2(T(t) \mapsto (t_1 - t) \mapsto ctrlV) = TRUE$
THEN
act1: $t := t1$
act2: $exec := plant$
END

Figure 6.16: Specific Heater Progress.

The continuous part of the *Smart Heating* system is described by the event *Plant* depicted by Figure 6.17. Event *Plant* replaces the generic ordinary differential equation e by that associated with the *Smart Heating* system, $\frac{dT}{dt} = ctrlV$. It also adds the safety property specified by the guard *grd*6 of Figure 6.7. In Figure 6.18, we instantiate the guards, *grd*3 and *grd*4, to check that the values *temp* and $-temp$ respectively satisfy the safety envelop formulas $safe_1$ and $safe_2$ defined for this case study (*Rule_ET1_M_5*). Since the discrete behavior of the *Smart Heating* system is described by two evade modes, two events *Ctrl_evade_1* and *Ctrl_evade_2* are respectively defined as depicted by Figures 6.19 and 6.20. Event *Ctrl_evade_1* is linked to the safety property $safe_1$ and event *Ctrl_evade_1* is linked to $safe_2$ (*Rule_ET1_M_6*).

6.1.3 Instantiating the Generic TimeTriggered Model

The specific *TimeTriggered* model instantiates the generic *TimeTriggered* model (see Section 5.4.2) by defining a specific *SpecificTimeTriggered_Ctx1* context and a specific *SpecificTimeTriggered_M1* machine. *SpecificTimeTriggered_Ctx1* context depicted by

EVENT Plant
REFINES Plant
ANY $T1$
WHERE
grd1 : $exec = plant$
grd2 : $T1 \in [0,t] - dom(T) \rightarrow \mathbb{R}^+$
grd3 : $ode(f\_evol\_plantV(ctrlV), T1(t), t) \in DE(\mathbb{R})$
grd4 : $Solvable([0,t] - dom(T), ode(f\_evol\_plantV(ctrlV), T1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), T1(t), t),$
$[0,t] - dom(T), [0,t] - dom(T), T1)$
grd6 : $\forall xx \cdot xx \in dom(T1) \implies prop(T1(xx)) = TRUE$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), T1(t), t)$
THEN
act1 : $T := T \Leftarrow T1$
act2 : $exec := ctrl$
END

Figure 6.17: Specific Event_Heater Plant.

EVENT Ctrl_normal
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{temp, -temp\}$
grd3 : $nCtrlV = temp \implies safe_1(T(t) \mapsto temp) = TRUE$
grd4 : $nCtrlV = -temp \implies safe_2(T(t) \mapsto -temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Figure 6.18: Specific Heater Ctrl_normal.

EVENT Ctrl_evade_1
WHERE
grd1 : $exec = ctrl$
grd2 : $safe_1(T(t) \mapsto temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := temp$
END

Figure 6.19: Specific Heater Ctrl_evade_1.

EVENT Ctrl_evade_2
WHERE
grd1 : $exec = ctrl$
grd2 : $safe_2(T(t) \mapsto -temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := -temp$
END

Figure 6.20: Specific Heater Ctrl_evade_2.

Figure 6.21 extends the specific *SpecificEventTriggered_Ctx1* and instantiates the generic *TimeTriggered_Ctx* (see Figure 5.11) by applying the following two rules:

- *Rule_TT_Ctx_1*: defines the control period *epsilon* and its related properties as specified by $axm1$ of Figure 6.21.
- *Rule_TT_Ctx_2*: specifies for each sub-formula $safe_i$ defined in the specific *SpecificEventTriggered_Ctx1* context, a formula $safeEpsilon_i$ by taking into account the control period *epsilon*; $SafeEpsformi$ represents the formula to be modeled (see $axm2$ and $axm3$ of Figure 6.21).

CONTEXT SpecificTimeTriggered_Ctx1
EXTENDS SpecificEventTriggered_Ctx1
CONSTANTS $epsilon$, $safeEpsilon_1$, ..., $safeEpsilon_k$
AXIOMS
axm1: $epsilon \in TIME \land sigma \leq epsilon \land\ 0 < epsilon$
axm2: $\bigwedge_{i=1..k} safeEpsilon_i \in (S \times \mathbb{R}) \rightarrow BOOL$
axm3: $\bigwedge_{i=1..k} safeEpsilon_i = (\lambda(y_1, ..., y_n) \mapsto ctrlV \cdot (y_1, ..., y_n) \in S \land ctrlV \in \mathbb{R} \mid SafeEpsform_i)$
END

Figure 6.21: SpecificTimeTriggered_Ctx1.

The specific *EventTriggered* model associated with the *Smart Heating* system is refined to obtain a specific *TimeTriggered* model composed of a specific context, *Time_Heater_ctx*, and a specific machine, *Time_Heater_M*. Figure 6.22 depicts the different elements of *Time_Heater_ctx*. Acoording to *Rule_TT_Ctx_2*, two safety envelops taking into account the control period *epsilon* are defined (see $axm4$ and $axm5$).

CONTEXT Time_Heater_Ctx
EXTENDS Event_Heater_Ctx
CONSTANTS $epsilon$, $safeEpsilon_1$, $safeEpsilon_2$
AXIOMS
axm1: $epsilon \in TIME \land sigma \leq epsilon \land 0 < epsilon$
axm2: $safeEpsilon_1 \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm3: $safeEpsilon_2 \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm4: $safeEpsilon_1 = (\lambda\, T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T + ctrlV \times epsilon \leq T\_max))$
axm5: $safeEpsilon_2 = (\lambda\, T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T + ctrlV \times epsilon \geq T\_min))$
END

Figure 6.22: Time_Heater_Ctx.

The specific *SpecificTimeTriggered_M1* machine refines the specific *SpecificEventTriggered_M1* machine following the rules *Rule_TT_M_i* defined below:

- *Rule_TT_M_1*: instantiates the invariants $inv1$ and $inv3$ of Figure 6.23 by replacing the generic continuous variables *plantV* by the specific ones $specPlantV_i$. Moreover, the invariant $inv2$ of Figure 6.23 is specified for each sub-formula $safeEpsilon_i$.
- *Rule_TT_M_2*: adds the following property to the event *Progress* of Figure 6.11, $(t_1 - t) \leq epsilon$, to guarantee that the controller reacts at least every *epsilon* time. Moreover, it removes the guard $grd3$ of the event *Progress* (see Figure 6.24) to give rise to refinement proof obligations between the generic and the specific models.

INVARIANTS

inv1 : $\exists\, t1 \cdot t1 \in TIME \wedge \bigwedge_{i=1..n} dom(specPlantV_i) = [0, t1] \wedge (t - t1) \leq epsilon$
$\wedge(exec \neq plant \implies t1 = t) \wedge (exec = plant \implies t > t1)\wedge$
$(\bigwedge_{i=1..k} ctrlV \notin evade\_value_i \wedge$
$exec = plant \implies safeEpsilon_i((specPlantV_1(t1), ...,$
$specPlantV_n(t1)) \mapsto ctrlV) = TRUE)$

inv2 : $\bigwedge_{i=1..k} ctrlV \notin evade\_value_i \wedge exec = prg \implies safeEpsilon_i($
$(specPlantV_1(t), ..., specPlantV_n(t) \mapsto ctrlV) = TRUE$

inv3 : $\forall t1, t2 \cdot t1 \in TIME \wedge t2 \in TIME \wedge dom(specPlantV_i) = [0, t1] \wedge$
$dom(specPlantV_i) = [0, t2] \implies t1 = t2$

Figure 6.23: SpecificTimeTriggered_M1 INVARIANTS.

EVENT Progress **REFINES** Progress
ANY t_1
WHERE
grd1 : $exec = prg$
grd2 : $t_1 \in TIME \wedge (t < t_1) \wedge (t_1 - t) \geq sigma \wedge (t_1 - t) \leq epsilon$
THEN
act1 : $t := t_1$
act2 : $exec := plant$
END

Figure 6.24: SpecificTimeTriggered_M1 Progress.

- *Rule_TT_M_3*: refines the event *Plant* described in Figure 6.12 by adding the parameters *lastTime* and *epsilon*1 as well as their properties specified by the guards *grd*2, *grd*3 and *grd*4 of Figure 6.25. Moreover, for each continuous parameter $specPlant1_i$, a solution of the differential equation that describes its behavior is defined (see Sol_i in *grd*5 of Figure 6.25). Guard *grd*6 of Figure 6.25 is defined to ensure that the specific parameters $specPlant1_i$ represents a solution of the $ode(f\_evol\_plantV(ctrlV), specPlant1_i, t)$ using the operator *solutionOf* defined in [10].

EVENT Plant
REFINES Plant
ANY $specPlant1_1$, ... , $specPlant1_n$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \wedge \bigwedge_{i=1..n} dom(specPlantV_i) = [0, lastTime]$
grd3 : $t > lastTime \wedge \bigwedge_{i=1..n} lastTime \in dom(specPlantV_i)$
grd4 : $epsilon1 = (t - lastTime)$
grd5 : $\bigwedge_{i=1..n} specPlant1_i = (\lambda\, t1 \cdot t1 \in TIME \wedge t1 > lastTime \wedge t1 \leq t \mid Sol_i)$
grd6 : $solutionOf([0, t] - dom(plantV), ([0, t] - dom(plantV))$
$\lhd V1, ode(f\_evol\_plantV(ctrlV), (specPlant1_1(t), ..., specPlant1_n(t)), t))$
...
THEN
act1 : $specPlantV_i := specPlantV_i \lhd\!\!+ specPlant1_i$
act2 : $exec := ctrl$
END

Figure 6.25: SpecificTimeTriggered_M1 Plant.

- *Rule_TT_M_4*: replaces each formula $safe_i$ by the associated formula $safeEpsilon_i$ in the events *Ctrl_normal* (see Figure 6.13) and *Ctrl_evade* (see Figure 6.14).

Figure 6.26 depicts the *INVARIANTS* part of the *Time_Heater_M* machine. The invariants specified in Figure 6.23 are instantiated by replacing $specPlantV_i$ by T, the $safeEpsilon_i$ formulas by, $safeEpsilon_1$ and $safeEpsilon_2$, and the $evade\_value_i$ by, $evade\_value_1$ and $evade\_value_2$ (*Rule_TT_M_1*). The solution obtained from solving the differential equation that describes the continuous behavior of the *Smart Heating* system is: $ctrlV \times epsilon1 + T(lastTime)$. This solution is added in the event *Plant* (see Figure 6.27), (*Rule_TT_M_3*). The events *Ctrl_normal*, *Ctrl_evade_1* and *Ctrl_evade_2* are refined by replacing the formulas $safe_1$ and $safe_2$ respectively by formula $safeEpsilon_1$ and $safeEpsilon_2$ (see Figures 6.28, 6.29 and 6.30).

INVARIANTS

inv1 : $\exists t1 \cdot t1 \in TIME \land dom(T) = [0, t1] \land (t - t1) \leq epsilon \land$
$(exec \neq plant \implies t1 = t) \land (exec = plant \implies t > t1) \land$
$(ctrlV \notin evade\_value_1 \land exec = plant \implies safeEpsilon_1(T(t1) \mapsto ctrlV) = TRUE)$
$\land (ctrlV \notin evade\_value_2 \land exec = plant \implies safeEpsilon_2(T(t1) \mapsto ctrlV) = TRUE)$
inv2 : $ctrlV \notin evade\_value_1 \land exec = prg \implies safeEpsilon_1(T(t) \mapsto ctrlV) = TRUE$
inv3 : $ctrlV \notin evade\_value_2 \land exec = prg \implies safeEpsilon_2(T(t) \mapsto ctrlV) = TRUE$
inv4 : $\forall t1, t2 \cdot t1 \in TIME \land t2 \in TIME \land dom(T) = [0, t1] \land dom(T) = [0, t2] \implies t1 = t2$

Figure 6.26: Time_Heater_M INVARIANTS.

EVENT Plant **REFINES** Plant
ANY $T1$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \land dom(T) = [0, lastTime]$
grd3 : $t > lastTime \land lastTime \in dom(T)$
grd4 : $epsilon1 = (t - lastTime)$
grd5 : $T1 = (\lambda\, t1 \cdot t1 \in TIME \land t1 > lastTime \land t1 \leq t \mid ctrlV \times epsilon1 + T(lastTime))$
grd6 : $ode(f\_evol\_plantV(ctrlV), T1(t), t) \in DE(\mathbb{R})$
grd7 : $Solvable([0, t] - dom(V), ode(f\_evol\_plantV(ctrlV), T1(t), t))$
grd8 :
$solutionOf([0, t] - dom(T), ([0, t] - dom(T)) \lhd T1, ode(f\_evol\_plantV(ctrlV), T1(t), t))$
THEN
$act1 : T := T \Leftarrow T1$
$act2 : exec := ctrl$
END

Figure 6.27: Time_Heater_M Plant.

EVENT Ctrl_normal **REFINES** Ctrl_normal
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{temp, -temp\}$
grd3 : $nCtrlV = temp \implies safeEpsilon_1(T(t) \mapsto temp) = TRUE$
grd4 : $nCtrlV = -temp \implies safeEpsilon_2(T(t) \mapsto -temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Figure 6.28: Time_Heater_M Ctrl_normal.

EVENT Ctrl_evade_1 **REFINES** Ctrl_evade_1
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilon_1(T(t) \mapsto temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := temp$
END

Figure 6.29: Time_Heater_M Ctrl_evade_1.

EVENT Ctrl_evade_2 **REFINES** Ctrl_evade_2
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilon_2(T(t) \mapsto -temp) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := -temp$
END

Figure 6.30: Time_Heater_M Ctrl_evade_2.

6.2 Instantiation from the EventTriggered Level

The instantiation starts by directly refining the generic *EventTriggered* model. The obtained model is composed of a specific context named, *SpecificEventTriggered_Ctx2*, and a specific machine named, *SpecificEventTriggered_M2*. This model is then refined in order to design a specific *TimeTriggered* model that instantiates the generic *TimeTriggered* model as depicted by Figure 6.31. The specific *TimeTriggered* model is also composed of a specific context called, *SpecificTimeTriggered_Ctx2*, and a specific machine called, *SpecificTimeTriggered_M2*, obtained following the same rules defined in Section 6.1.3.

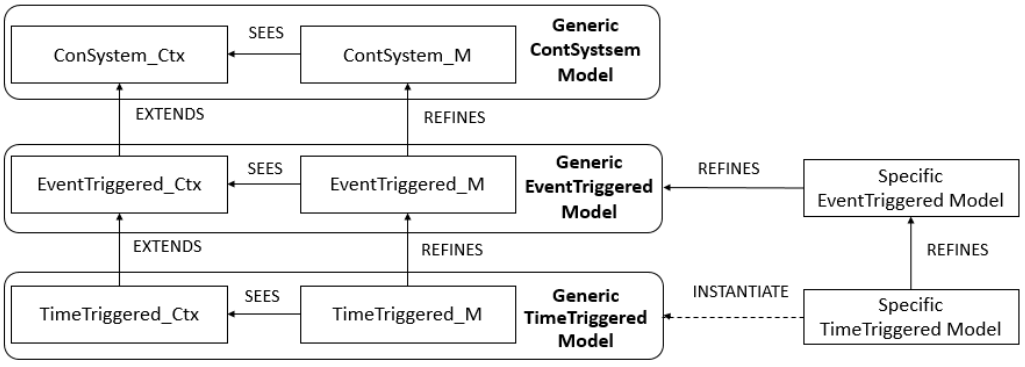

Figure 6.31: Second Strategy: Instantiation from the *EventTriggered* Level.

6.2.1 Instantiating the Generic EventTriggered Context

The specific context *SpecificEventTriggered_Ctx2* depicted by Figure 6.32 extends the generic *EventTriggered_Ctx* by applying the following rules:

- *Rule_ET2_Ctx_1*: defines all the constant parameters of the modeled system and their related properties as well as the initial values of the continuous variables by applying the rules *Rule_CS_Ctx_1*, *Rule_CS_Ctx_2* and *Rule_CS_Ctx_3* (see $axm1$, $axm2$ and $axm3$ of Figure 6.32).
- *Rule_ET2_Ctx_2*: instantiates the safety envelop of the modeled system as well as the event trigger formula defined in the generic *EventTriggered_Ctx*. If the system is composed of multiple safety envelops and event trigger formulas, these formulas should be declared in the *CONSTANTS* clause and their definitions should be added in the *AXIOMS* clause (see $axm4$, $axm5$, ... and $axm9$ of Figure 6.32). This rule also defines the set of evade values and set of possible values of the controlled variable val_i (see $axm11$ and $axm12$ of Figure 6.32). Moreover, it instantiates the safety property using the constant *prop* defined in the generic *EventTriggered_Ctx* (see $axm13$ of Figure 6.32) .
- *Rule_ET2_Ctx_3*: defines the differential equations of the modeled system by instantiating the function f_evol_plantV, where *ode* is defined in S and built using the right parts of the specific ODEs (see $axm10$ of Figure 6.32).

CONTEXT SpecificEventTriggered_Ctx2
EXTENDS EventTriggered_Ctx
CONSTANTS Cst_1 , ... , Cst_l , $specPlantVinit_1$, ... , $specPlantVinit_n$, $safe_1$, ... , $safe_k$, evt_trig_1 , ... , evt_trig_k , $evade\_value_1$, ... , $evade\_value_k$, val_1 , ... , val_k
AXIOMS
axm1: $(specPlantVinit_1, ..., specPlantVinit_n) = plantVinit$
axm2: $\bigwedge_{i=1..n} specPlantVinit_i \in \mathbb{R}$
axm3: $\bigwedge_{i=1..l} prt_i$
axm4: $safe = (\lambda\ (y_1, ..., y_n) \mapsto x \cdot (y_1, ..., y_n) \in S \wedge x \in \mathbb{R} \mid Safeform)$
axm5: $evt\_trig = (\lambda (y_1, ..., y_n) \mapsto t1 \mapsto x \cdot (y_1, ..., y_n) \in S \wedge$
$t1 \in TIME \wedge x \in \mathbb{R} \mid Trigform)$
axm6: $\bigwedge_{i=1..k} safe_i \in (S \times \mathbb{R}) \to BOOL$
axm7: $\bigwedge_{i=1..k} evt\_trig_i \in (S \times TIME) \times \mathbb{R} \to BOOL$
axm8: $\bigwedge_{i=1..k} safe_i = (\lambda\ (y_1, ..., y_n) \mapsto x \cdot (y_1, ..., y_n) \in S \wedge x \in \mathbb{R} \mid Safeform_i)$
axm9: $\bigwedge_{i=1..k} evt\_trig_i = (\lambda (y_1, ..., y_n) \mapsto t1 \mapsto x$
$\cdot (y_1, ..., y_n) \in S \wedge t1 \in TIME \wedge x \in \mathbb{R} \mid Trigform_i$
axm10: $\forall x \cdot x \in \mathbb{R} \implies (f\_evol\_plantV(x) = (\lambda t \mapsto (y_1, ..., y_n)$
$\cdot t \in TIME \wedge (y_1, ..., y_n) \in S \mid ode)$
axm11: $\bigwedge_{i=1..k} evade\_value_i \subseteq \mathbb{R}$
axm12: $\bigwedge_{i=1..z} val_i \in \mathbb{R}$
axm13: $prop = (\lambda (y_1 \mapsto .... \mapsto y_n) \cdot (y1 \mapsto .... \mapsto yn) \in \mathbb{R}^n \mid bool(PropForm))$
END

Figure 6.32: SpecificEventTriggered_Ctx2.

Figure 6.33 depicts the *SpecificEventTriggered_Ctx2* associated with the *Smart Heating* case study (Rules *Rule_ET2_Ctx_1*, *Rule_ET2_Ctx_2* and *Rule_ET2_Ctx_3*). *Event_Heater_Ctx2* redefines all the constants and properties defined in the context *Specific_Heater_Ctx* described in Figure 6.3 and context *SpecificEventTriggered_Ctx1* described in Figure 6.8. Moreover, context *SpecificEventTriggered_Ctx1* extends the generic context *EventTriggered_Ctx*.

6.2.2 Instantiating the Generic EventTriggered Machine

The specific *SpecificEventTriggered_M2* machine directly refines the generic *EventTriggered_M* machine by instantiating the following two rules:

CONTEXT Event_Heater_Ctx2
EXTENDS EventTriggered_Ctx
CONSTANTS $T0$, T_max, T_min, $safe_2$, evt_trig_2, $evade\_value_2$, $temp$
AXIOMS
axm1: $T0 = plantVInit$
axm2: $T0 \in RRealPlus$
axm3: $T0 < T\_max \land T0 > T\_min \land T\_max \in \mathbb{R} \land$
$T\_max > T\_min \land T\_min \in \mathbb{R} \land T\_min > 0$
axm4: $safe = (\lambda T \mapsto x \cdot T \in \mathbb{R} \land x \in \mathbb{R} \mid bool(T < T\_max))$
axm5: $evt\_trig = (\lambda\, T \mapsto t1 \mapsto x \cdot T \in \mathbb{R}$
$\land t1 \in TIME \land x \in \mathbb{R} \mid bool(T + x \times t1 \leq T\_max))$
axm6: $safe_2 \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm7: $evt\_trig_2 \in (\mathbb{R} \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm8: $safe_2 = (\lambda\, T \mapsto x \cdot T \in \mathbb{R} \land x \in \mathbb{R} \mid bool(T > T\_min))$
axm9: $evt\_trig_2 = (\lambda T \mapsto t1 \mapsto x \cdot T \in \mathbb{R}$
$\land t1 \in TIME \land x \in \mathbb{R} \mid bool(T + x \times t1 \geq T\_min))$
axm10: $\forall x \cdot x \in \mathbb{R} \implies (f\_evol\_plantV(x) = (\lambda t \mapsto T \cdot t \in TIME \land T \in \mathbb{R} \mid x))$
axm11: $evade\_value = \{-temp\}$
axm12: $evade\_value_2 \subseteq \mathbb{R} \land evade\_value_2 = \{temp\}$
axm13: $temp \in \mathbb{R} \land temp > 0$
axm14: $prop = (\lambda\, T \cdot T \in \mathbb{R} \mid bool(T \leq T\_max \land T \geq T\_min))$.
END

Figure 6.33: Event_Heater_Ctx2.

- *Rule_ET2_M_1*: defines the *INVARIANTS* part of the *SpecificEventTriggered_M2* machine. It specifies the possible values of *ctrlV* (see *inv*1 of Figure 6.34). It also defines n specific continuous variables and replaces the generic continuous variable *plantV* by these variables using the *bind* operator (see *inv*2 and *inv*3 of Figure 6.34). Moreover, it instantiates *inv*3 for each specific continuous variable and adds the invariants *inv*4 and *inv*5. Moreover, it defines the system safety property using the formula *prop* (see *inv*6 of Figure 6.34).

- *Rule_ET2_M_2*: events *Progress*, *Plant*, *Ctrl_normal* and *Ctrl_evade* are instantiated following the same instructions described respectively by rules *Rule_ET1_M_3*, *Rule_ET1_M_4*, *Rule_ET1_M_5* and *Rule_ET1_M_6*. The only difference is in the event *Plant*. In this second strategy the generic ODE e was already replaced using the function *f_evol_plantV* in the generic *EventTriggered_M* described in 5.4.1. The continuous variable *plantV* is replaced by the specific ones. Moreover, the events *Ctrl_normal* and *Ctrl_evade* are refined by replacing the continuous parameters by the specific ones.

INVARIANTS
inv1: $ctrlV \in setVal$
inv2: $\bigwedge_{i=1..n} specPlantV_i \in [0,t] \nrightarrow RReal$
inv3: $plantV = bind(specPlantV_1, bind(specPlantV_2, bind(...)))$
inv4: $\bigwedge_{i=1..n} exec \neq plant \implies dom(specPlantV_i) = [0,t]$
inv5: $exec = plant \implies t \notin dom(plantV)$
inv6: $\forall w \cdot w \in dom(bind(specPlantV_1,$
$bind(specPlantV_2, bind(...)))) \implies prop(specPlantV_1(w), ...,$
$specPlantV_n(w)) = TRUE$

Figure 6.34: SpecificEventTriggered_M2 INVARIANTS.

6.3 Discussion

The approach depicted by Figure 6.1 allows building the model step by step starting by specifying the continuous aspects of the system. Therefore, to instantiate by refinement the abstract model *ContSystem*, we define the continuous variables of the system and model the continuous evolution by refining the event *Plant*. This permits to cope with the proof complexity by decomposing the proof obligations, such that in the abstract model we only deal with the proof obligations related to the continuous aspects of the system and in the refined model we will have the proof obligations related to the safety properties of the controlled system. In the approach depicted by Figure 6.31, instantiating by refinement the *EventTriggered* model requires introducing both the continuous and discrete aspects of the system. Therefore, in addition to the continuous variables and their ODEs, we need to define the controller discrete states as well as the safety properties of the system, this generates more POs that are also more complex to discharge.

	Modeling	**Proof Obligations**
First strategy: instantiation from the ContSystem level	Allows modeling hybrid systems step by step, which simplifies the design of complex case studies in Event-B.	Since the Instantiation of POs is not yet supported by Rodin, we have to prove POs already discharged in the generic model. However, there is an ANR project, EBRP, whose objective is to develop an extension of Rodin that supports this concept.
Second strategy: instantiation from the EventTriggered level	Event-B does not support the continuous aspects of hybrid systems. In the case of complex case studies, the introduction of all the system details, make its modeling in Event-B more difficult.	Event-B refinement preserves the safety properties, so a concrete machine inherits from all the properties proved in an abstract one. Therefore, we do not need to redo the POs of the higher level .
What we retain	The more we start from an abstract level, the more we gain in simplicity of modeling.	The more we start from a concrete level, the more we gain in simplicity of POs.

Figure 6.35: Main Differences between the two Strategies.

Figure 6.35 summarizes the main differences between the two strategies. Note that the POs generated to prove the correctness of the *Specific TimeTriggered* model are similar to those generated for the *Generic TimeTriggered* model. In that case, it suffices to apply the same proof scripts used to prove the generic ones. This applies for the *Specific EventTriggered* model that instantiates the *Generic EventTriggered* model, its POs are similar to those already discharged in the generic model. Therefore, the instantiation of the generic models should allow reusing the already discharged proofs. Since the instantiation of POs is not yet supported by RODIN, we have to prove POs already discharged in the generic model. The reuse of proofs by instantiation is the subject of the ANR project, *EBRP* [9], whose objective is to develop an extension of RODIN to support such a concept.

6.4 Conclusion

In this chapter, we define a set of rules that are used to apply our generic models on specific case studies. Two approaches for instantiating the generic approach were defined. The first approach refines the abstract model *ContSystem* and then instantiates the generic *Event* and *Time-Triggered* models. The second one proposes to directly refine the generic *EventTriggered* model and then instantiate the generic *TimeTriggered* model. We have also considered the strategy that consists in starting by refining the *TimeTriggered* model but the proofs were more complex since it is more difficult to prove safety properties on *TimeTriggered* models than on *EventTriggered* models, as previously explained.

Chapter 7

Interfacing EVENT-B with SAGEMATH

Contents

Event-B is a formal method designed for modeling and proving the correctness of discrete systems. It does not support the resolution of ordinary differential equations for proving the correctness of hybrid systems. To deal with this limitation, we interface the Rodin tool with the differential equation solver, SageMath (System for Algebra and Geometry Experimentation) [6].

Section 7.1 introduces a correct-by-construction approach, using Event-B and its refinement strategy, for solving linear ordinary diffential equations. Section 7.2 describes the development process of a plugin that permits to call SageMath from Rodin. Hybrid systems whose behavior is described by nonlinear ordinary equations are treated in Section 7.3. Last, Section 7.4 concludes the chapter with a discussion on the proposed approaches.

7.1 Solving Linear ODEs in Event-B

In this section, we introduce a correct-by-construction approach to deal with the resolution of linear ODEs in Event-B. The approach follows the development schema depicted in Figure 7.1. It extends the approach introduced in Chapter 5 by adding by refinement a new generic model called *TimeTriggeredDesolve_M* which introduces a function named *B_desolve* to model exact solutions of ordinary differential equations in Event-B.

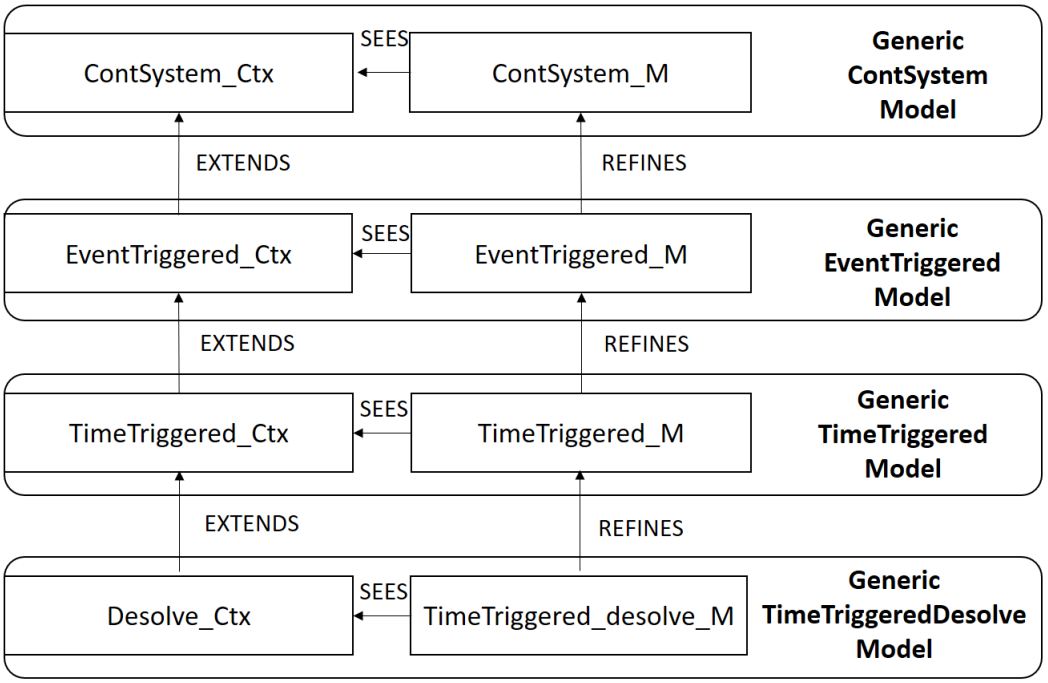

Figure 7.1: Generic Event-B specification with the *B_desolve* function.

The proposed approach defines a new set named *PROP* in the context *EventTriggered_Ctx*. This set allows linking the safety property with the formulas needed to model the different behaviors of the modeled system. We make the assumption that the safety property is in conjunctive normal form ($\bigwedge_{i=1..n} p_i$) and that, for each sub-formula p_i, $event\_trig_i$, $safe_i$, $safeEpsilon_i$ and a set of evade values $evade\_values_i$ for *ctrlV* are specified.

$PROP = \bigcup_{i=1..n} \{p_i\}$

Instead of modelling the controller part with two events, *Ctrl_normal* and *Ctrl_evade* as presented in Chapter 5, the controller is modeled by a single event *Ctrl*. This event checks, for each property p_i, that the safety envelop is true if the chosen value *value* does not belong to the set of evade values of p_i (see *grd3* in Figure 7.3). *TimeTriggeredDesolve* model then refines the new generic model *TimeTriggered* which applies the same modifications defined in the new model *EventTriggered*. *TimeTriggeredDesolve* introduces the function *B_desolve* defined to model analytical solutions of ODEs in Event-B. It is composed of an Event-B context named *Desolve_Ctx* and a machine named *TimeTriggered_desolve_M*.

7.1.1 Context Desolve_Ctx

Context *Desolve_Ctx* extends the context *TimeTriggered_Ctx* by introducing the generic function *B_desolve* which returns a function of type $\mathbb{R} \rightarrow \mathbb{R}$ that represents the solutions of a given continuous variable. Introducing this function in our generic approach allows us

CONTEXT EventTriggered_Ctx
EXTENDS ContSystem_Ctx
SETS $EXEC$, $PROP$
CONSTANTS $prop\_safe$, $prop\_evt\_trig$, $ctrl$, $plant$, prg, f_evol, f_evol_plantV, $prop\_evade\_values$
AXIOMS
axm1: $prop\_safe \in PROP \to ((S \times \mathbb{R}) \to BOOL)$
axm2: $prop\_evt\_trig \in PROP \to ((S \times TIME) \times \mathbb{R} \to BOOL)$
axm3: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm4: $f\_evol \in \mathbb{R} \to S$
axm5: $f\_evol\_plantV \in (\mathbb{R} \to (TIME \times S \to S))$
axm6: $\forall\, ctrlV \cdot ctrlV \in \mathbb{R} \implies (f\_evol\_plantV(ctrlV) = (\lambda\, t \mapsto plantV \cdot t \in TIME \land plantV \in S \mid f\_evol(ctrlV)))$
axm7: $prop\_evade\_values \in PROP \to P1(\mathbb{R})$
END

Figure 7.2: CONTEXT EventTriggered_Ctx.

EVENT Ctrl
ANY $value$
WHERE
grd1: $exec = ctrl$
grd2: $value \in \mathbb{R}$
grd3: $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies (prop\_safe(x))(plantV(t), value) = TRUE)$
THEN
act1: $ctrlV := value$
act2: $exec := prg$
END

Figure 7.3: EventTriggered Ctrl.

to prove the safety properties of hybrid systems in a *TimeTriggered* system, which was not possible with the generic approach previously introduced. Moreover, this function serves to establish the link between our Event-B models and the differential equations solver SageMath.

$$B\_desolve \in (\mathbb{N} \times \mathbb{R} \times (\text{TIME} \to \mathbb{R}) \times \text{TIME} \times (\text{TIME} \times \mathbb{R})) \to (\mathbb{R} \to \mathbb{R})$$

- 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.
- *the independent variable*, represented by a discrete variable, is typed by $TIME$.
- the last parameter denotes the initial values of both the independent variable and the unknown function.

As stated before, the main objective of the *TimeTriggeredDesolve* model is to prove the safety properties in a *TimeTriggered* model. Therefore, instead of defining these properties in the *EventTriggered* model as described in 5.4.3, we define the constant *prop*, used to model the safety properties of a given system, in the *Desolve_Ctx* context, see $axm2$ in 7.4.

CONTEXT Desolve_Ctx
EXTENDS TimeTriggered_Ctx
CONSTANTS B_desolve, prop
AXIOMS
axm1: $B\_desolve \in (\mathbb{N} \times \mathbb{R} \times (\text{TIME} \to \mathbb{R}) \times \text{TIME} \times (\text{TIME} \times \mathbb{R})) \to (\mathbb{R} \to \mathbb{R})$
axm2: $prop \in S \to BOOL$
axm3: $prop(plantVInit) = TRUE$
END

Figure 7.4: Context Desolve_Ctx.

7.1.2 Machine TimeTriggered_desolve_M

Machine *TimeTriggered_desolve_M* refines the machine *TimeTriggered_M* by using the function *B_desolve* in the *Plant* event to specify the generic solution of the generic continuous variable *plantV*. Event *Plant* of the *TimeTriggered* model is refined to calculate the value of *plant*1 during the period from *lastTime* to *t* using the function *B_desolve*, which is specified by *grd*3 in Figure 7.5. This guard is used to link the abstract event to its refinement. It strengthens the guards *grd*6 and *grd*7 that aim at modeling the ordinary differential equation solution using the operators of the theory *DiffEq*. The parameters *dvar*, *ivar* and *ics* of the predefined function *desolve* of SageMath are represented respectively by the dependent variable *plantV*, the independent variable *t* and the initial values of *plantV* and *t*. The parameter *lastTime* is introduced to represent the last progression of time at which *plantV* has been calculated. The solution of a given ordinary differential equation is calculated from *lastTime* to *t* in order not to overwrite the old values of the continuous variable *plantV*. Note that, in the case of a system with two or more continuous variables, we replace the generic parameter *plant*1 by *n* parameters using the operator *bind*, where *n* represents the number of the continuous variables. For each parameter a function *B_desolve* is defined to obtain the exact solution of the associated continuous variable.

EVENT Plant_time_desolve
REFINES Plant_time
ANY *plant*1, *lastTime*
WHERE
grd1: $exec = plant$
$grd2 : lastTime \in TIME \land dom(plantV) = [0, lastTime]$
$grd3 : plant1 = B\_desolve(1 \mapsto ctrlV \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)))$
grd4: $plant1 \in [0, t] - dom(plantV) \to S$
grd5: $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd6: $Solvable([0, t] - dom(plantV), ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd7: $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$[0, t] - dom(plantV), [0, t] - dom(plantV), plant1)$
grd8: $\forall xx \cdot xx \in dom(plant1) \Rightarrow prop(plant1(xx)) = TRUE$
THEN
act1: $plantV := plantV \lhd\!\!\!- plant1$
act2: $exec := prg$
END

Figure 7.5: TimeTriggeredDesolve Plant.

7.1.3 Correctness of the specification

Table 7.2 gives the statistics of the POs generated for the correctness of our generic models. It is noticeable that 47% of them were automatically discharged. These POs include the

Table 7.1: RODIN Proof Statistics for the Generic Models

Generic_Models	**Total**	**Automatic**	**Interactive**
ContSystem_M	8	1	7
EventTriggered_M	19	11	8
TimeTriggered_M	2	1	1
TimeTriggered_desolve_M	5	3	2

correctness of the events that specify the progression of time and those that specify the progression of the physical and the discrete parts and also the POs that verify the type of the variables. The POs related to the guards feasibility and well-definedness have been interactively discharged under RODIN. Comparing tables 5.1 and 7.2, we observe that the number of the POs generated has been reduced due to the use of the constant $PROP$ which allows defining a single event to describe the discrete part. Therefore, the POs related to the event $Ctrl\_evade$ have been removed and added to those of the event $Ctrl$.

To prove the correctness of the *TimeTriggeredDesolve* model, RODIN has generated five proof obligations, three of them were automatically discharged. The remaining proof obligation $PO1$ is a well-definedness proof obligation which aims at proving that the guard $grd3$ of 7.5, 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 $B\_desolve$. For this purpose, 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 $Plant\_time\_desolve$ 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: $lastTime \in dom(plantV) \land plantV \in \mathbb{R} \pfun \mathbb{R} \land 1 \mapsto ctrlV \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)) \in dom(B\_desolve) \land B\_desolve \in \mathbb{N} \times \mathbb{R} \times P(\mathbb{R} \times \mathbb{R}) \times \mathbb{R} \times (\mathbb{R} \times \mathbb{R}) \pfun P(\mathbb{R} \times \mathbb{R})$

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

7.1.4 Instantiating the Generic TimeTriggeredDesolve Model

To design specific systems following the generic approach, we instantiate the generic *TimeTriggeredDesolve* model by replacing the generic continuous variable $plantV$ by that or those associated with the specific system. The function $B\_desolve$ is then instantiated by the specific parameters of the modeled case study. The specific safety property is expressed as an invariant, in the specific instantiated machine, by the following formula, $\forall x \cdot x \in dom(plantV) \Rightarrow safetyProperty$, where $plantV$ denotes the continuous variables of the system and $safetyProperty$ is the specific safety property. This formula expresses that the safety property of the system shall be satisfied in the time interval $[0, t]$ which denotes the domain of $plantV$. Moreover, the instantiation consists in valuing the set $PROP$ and the different constants as follows:

$prop\_X = \bigcup_{i=1..n} \{p_i \mapsto X_i\}$

where $X \in \{event\_trig, safe, safeEpsilon, evade\_values\}$.

7.2 A tool for supporting the approach

In order to implement our approach, we built a new RODIN plug-in, called SAGEMATH plug-in, that interfaces the RODIN platform with the computer algebra system SAGEMATH to calculate the solutions of ODEs. Solving ODEs is needed in two steps of the proof activity: for proving the safety property and for proving 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 $B\_desolve(...)$ in order to replace it by the solution of the corresponding ODE.

7.2.1 The general process

The general process is composed of three main steps: (1) calling SAGEMATH from RODIN, (2) solving ordinary differential equations and (3) using the result returned in RODIN (see Figure 7.6). In the first step, an input field that allows calling SAGEMATH from RODIN appears when the current PO contains the terms $B\_desolve$. The second step consists in calling a predefined script generated systematically from the function $B\_desolve(...)$. The last step consists in translating the result of SAGEMATH into the specific EVENT-B language using the theory of reals. This result is added as an hypothesis to prove the current PO. More details on these steps are provided in the next subsections.

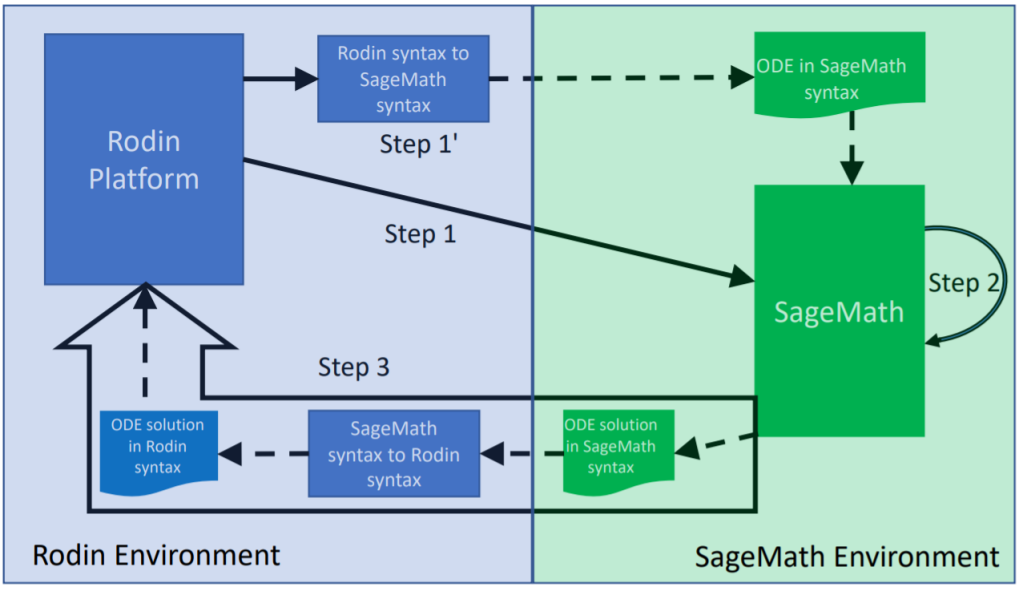

Figure 7.6: The General Process.

7.2.2 Calling SAGEMATH from RODIN (Step1)

To call SAGEMATH from RODIN, a button called *sagemath* has been added in the proof window using an Eclipse plug-in. The button is made available on a hypothesis/goal when this later contains the call to a function $B\_desolve(...)$. 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 plug-in. To implement the SAGEMATH plug-in using Eclipse IDE, the following Java classes have been defined: *SageTacticProvider*, *SageApplication* and *SageTactic*. Appendix E illustrates the description of the different stages that constitute the main scenario of use of the SAGEMATH plug-in. Figure 7.7 shows the sequence diagram associated with this development. For a RODIN PO containing a call to the $B\_desolve(...)$

function, the *getPossibleApplications* method implemented in the *SageTacticProvider* Java class creates a new instance of the Java class *SageApplication* which displays the *sagemath* button to the user using the method *getHyperlinkLabel()*. When the *sagemath* button is clicked by the user, the *getTactic()* method implemented in the *SageApplication* Java class creates a new instance of the Java class *SageTactic* that opens the platform SAGEMATH using an instance of the *ProcessBuilder* Java class. The three following subsections give more details for the Java classes that have been developed.

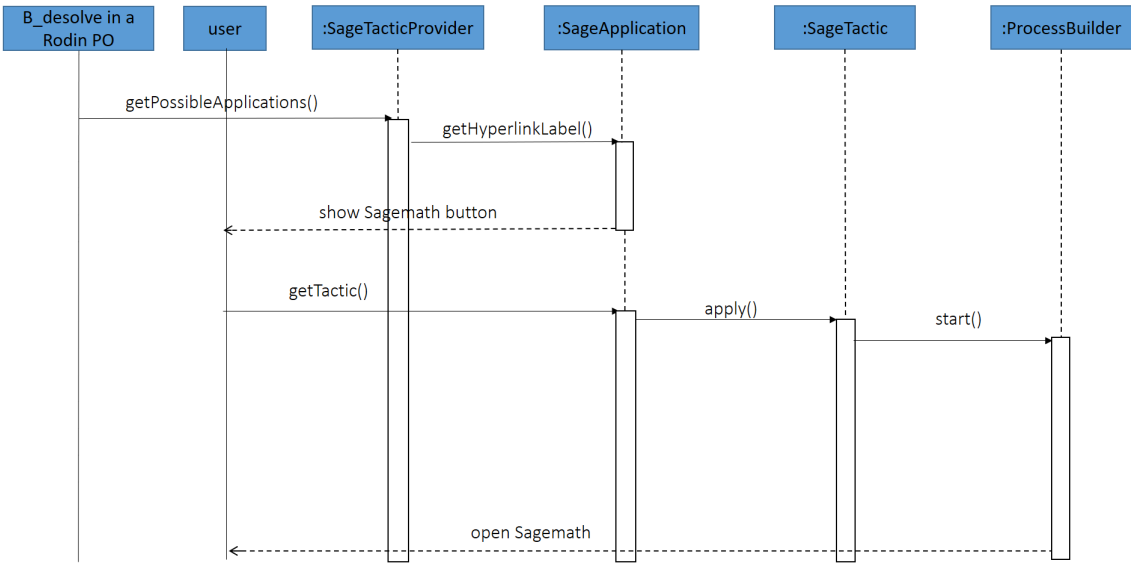

Figure 7.7: The Sequence Diagram of the SAGEMATH Plug-in.

7.2.2.1 SageTacticProvider Class

This Java class implements the method *getPossibleApplications* to check the presence of *B_ desolve* in each proof obligation and returns an instance of the *SageApplication* class. Function *getPossibleApplications* uses two main predicates as depicted by Figure 7.8, *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 tag, the left and right parts of the formula *pred*, are equal to those of the predicate that contains *B_ desolve*, we return the list of tactic that can be applied as a list of instances of *SageApplication* or null otherwise. This is repeated for each node of the proof tree.

```
public List<ITacticApplication> getPossibleApplications(
        IProofTreeNode node, Predicate hyp, String globalInput) {

    boolean isGoal = hyp == null;
    Predicate pred = ( isGoal ? node.getSequent().goal() : hyp);
```

Figure 7.8: Function *getPossibleApplications*.

7.2.2.2 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* as depicted by Figure 7.9. 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.

```
public Object apply(IProofTreeNode ptNode, IProofMonitor pm) {

    |

    ProcessBuilder pb = new ProcessBuilder("cmd", "/c",
            "C:\\Users\\Meryem\\AppData\\Roaming\\Microsoft\\Windows"
            + "\\Start Menu\\Programs\\SageMath 9.2\\SageMath 9.2.lnk");

    pb.redirectErrorStream(true);
     try
        {

            Process p = pb.start();
```

Figure 7.9: Calling SAGEMATH Using *ProcessBuilder*.

7.2.2.3 SageApplication Class

This Java class establishes the link between the checking of the presence of the function *B_ desolve* in the current proof obligation and the call to SAGEMATH. It implements in particular two methods of the abstract interface *IPositionApplication*:

- 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.

7.2.3 Solving ODEs in SAGEMATH (Step1' and Step2)

A SAGEMATH script is systematically generated from the EVENT-B function *B_ desolve*, with all the parameters necessary to execute the SAGEMATH predefined function *desolve*, Step 1' in Figure 7.6. According to the structure and the nature of the differential equation to be solved, a specific SAGEMATH script is defined. 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 a given hybrid system. A script is executed in SAGEMATH using the following command: $load('scriptName.sage')$. Figure 7.10 represents the script that solves a differential equation of type $T' = ctrlV$. This script is generated from the formula $B\_desolve(1 \mapsto ctrlV \mapsto T \mapsto t \mapsto (lastTime \mapsto T(lastTime)))(x)$, where:

- Line 1 is generated using the second parameter of *B_ desolve* and it specifies the right part of an ordinary differential equation.
- Line 2 is generated using the forth parameter of *B_ desolve* and it specifies the definition of the independent variable t.
- Line 3 is generated using the third parameter of *B_ desolve* and it specifies the definition of the continuous variable represented by T. The definition of this variable must always be after the definition of the independent variable.
- Line 4 is generated using the first part of the parameter $lastTime \mapsto T(lastTime)$ and it represents the last progression of time from which we calculate the values of the continuous variable.
- Line 5 represents the call to the SAGEMATH predefined function *desolve*. The first parameter 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*.

- Lines $6-8$ generate a text file, named "sageresult.txt", which store the result of the differential equation specified by *sol* in Line 5.

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

Figure 7.10: Script for an Ordinary Differential Equation of Type $T' = ctrlV$.

To solve an equation of the form $T' = ctrlV * t + c$, another script is defined. Line 5 is replaced by two lines: the first line is to introduce the constant c: $c = var('c')$ and the second one to replace formula $diff(T,t,1) == ctrlV$ by formula $diff(T,t,1) == ctrlV * t + c$.

7.2.4 Using SageMath Results in Rodin (Step3)

To discharge a proof obligation that includes a call to the function $B\_desolve()$ by using the result returned by SageMath, the call must be replaced by its solution *sol* returned by SageMath and written in the text file *sageresult.txt*. For this purpose, the predicate $(B\_desolve() = sol')$ is added as an additional hypothesis for the current PO. where sol' is a rewritten of *sol* according to the syntax of the theory of reals used in the project. Basically, the theory of reals 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 $ctrlV \times lastTime + T(lastTime)$ is rewrittten into $plus(times(ctrlV \mapsto lastTime) \mapsto T(lastTime))$.

7.3 Solving Nonlinear ODEs in Event-B

There are two types of methods for solving ODEs: analytical methods and numerical methods. Analytical methods use a set of theorems to obtain an exact solution for a given ordinary differential equation. For example, the computer algebra SageMath provides the predefined function, *desolve*(), that uses analytical methods to find analytical solutions for ODEs. However most differential equations cannot be solved exactly. Therefore, we must rely on numerical methods to obtain approximate solutions or use approximation techniques to transform an equation into an equivalent equation with an exact solution. For example, we can use linearization techniques to transform a nonlinear differential equation into a linear differential equation and then apply analytical methods for linear differential equations. The obtained solution is thus an approximate solution for the original one.

7.3.1 The Generic Approach

Figure 7.11 represents our proposed approach for modelling and verifying safety properties of nonlinear differential equations in Event-B. The approach consists of two sub-approaches depending of the order of the ODE: if the given differential equation is a first order ODE, then we use the numerical function *desolve_rk4()* defined in SageMath to find approximate solutions for ODEs. This method can only be applied to solve first order ODEs. Otherwise,

we linearize (if it is possible) the differential equation and use the approach that model the function *desolve()* in EVENT-B.

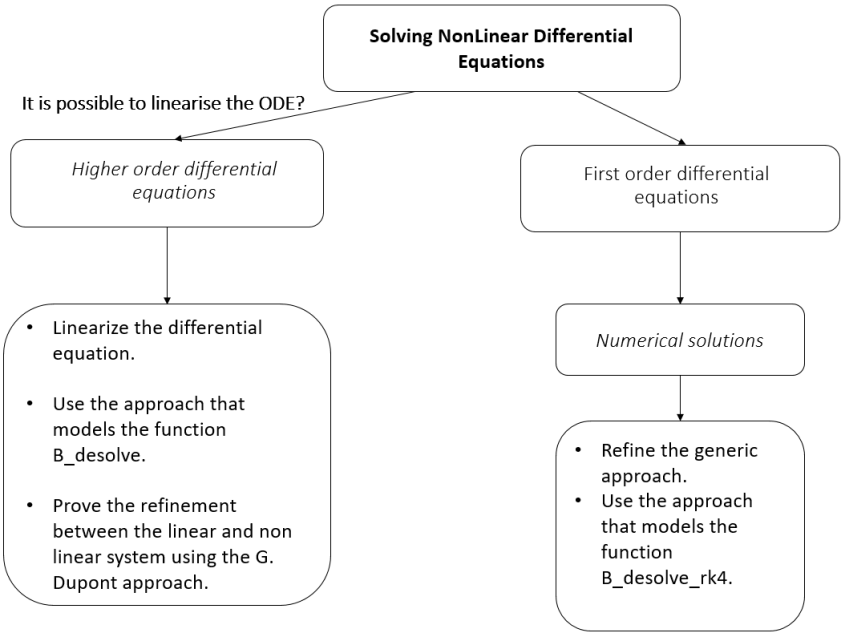

Figure 7.11: Solving Nonlinear Differential Equations using EVENT-B.

For verifying nonLinear systems in EVENT-B, we propose to prove their safety properties per time interval, i.e for a given interval $[t1, t2]$ we prove that its associated $plantV(t1)$ and $plantV(t2)$, obtained using the function $desolve\_rk4$, preserve the system safety properties, assuming the monotony of the values returned by $desolve\_rk4$ $(\forall\, t \cdot t \in [t1, t2] \Rightarrow (plantV(t1) \leq plantV(t) \leq plantV(t2)) \vee (plantV(t2) \leq plantV(t) \leq plantV(t1))$. For this purpose, we define an EVENT-B function, named $B\_desolve\_rk4$, that models approximate resolutions in EVENT-B. This function is defined in the context *Desolverk4* (see Figure 7.13) that extends the generic context *TimeTriggered_Ctx*. This context defines also the constant *prop* that specifies the system safety properties and that must be true for the initial value of the continuous variable $plant0$ at the instant t_0 $(axm3)$.

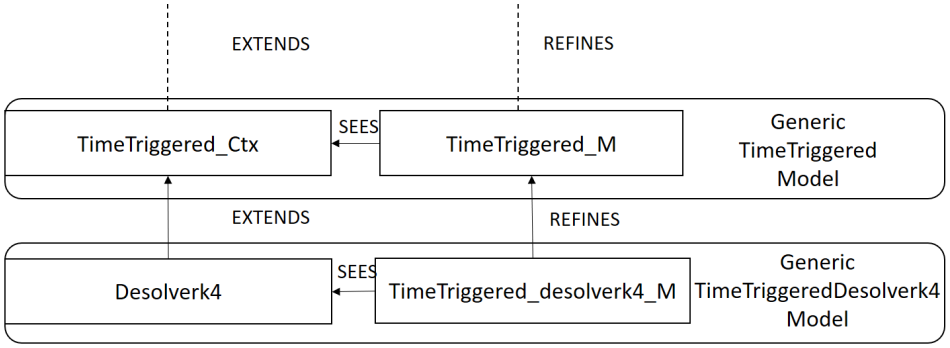

Figure 7.12: Generic EVENT-B specification for Approximate Solutions.

$B\_desolve\_rk4$ returns a function of type $\mathbb{R} \nrightarrow \mathbb{R}$ that represents the values of the continuous variables specified in the generic models by $plantV$ (see $axm1$).

$$B\_desolve\_rk4 \in \mathbb{R} \times (TIME \nrightarrow \mathbb{R}) \times TIME \times$$

$$(TIME \times \mathbb{R}) \times (TIME \times TIME) \rightarrow (\mathbb{R} \nrightarrow \mathbb{R})$$

- the first parameter, element of $\mathbb{R}$, represents the right term of the ordinary differential equation.
- the second parameter specifies the unknown function, represented by the continuous variable ($plantV$), is typed by $TIME \nrightarrow \mathbb{R}$.
- the third parameter is used to specify independent variable, represented by the discrete variable (time t), is typed by $TIME$.
- the fourth parameter represents the initial values of both the independent variable t and the unknown function $plantV$.
- the last parameter is used to specify the interval denoted $[t1, t2]$ for which we calculate the values of $plantV$.

CONTEXT Desolverk4
EXTENDS TimeTriggered_Ctx
CONSTANTS $B\_desolve\_rk4$, $prop$
AXIOMS
axm1: $B\_desolve\_rk4 \in \mathbb{R} \times (TIME \nrightarrow \mathbb{R}) \times TIME \times (TIME \times \mathbb{R}) \times (TIME \times TIME) \rightarrow (\mathbb{R} \nrightarrow \mathbb{R})$
axm2: $prop \in \mathbb{R} \rightarrow BOOL$
axm3: $prop(plantVInit) = TRUE$
END

Figure 7.13: CONTEXT Desolverk4.

The associated machine $TimeTriggered\_desolverk4\_M$ refines the generic machine $TimeTriggered\_M$ by adding the guard $grd6$ (see Figure 7.14) to the generic event $Plant\_time$. $grd6$ specifies that the new value of $plantV$ is equal to the result of the Event-B function $B\_desolve\_rk4$. This function takes as parameters the right term of the differential equation that depends on the controlled variable $ctrlV$, the continuous variable $plantV$, the independent variable t, the initial conditions of t and $plantV$ and the interval $[t1, t2]$. The initial conditions are represented by the couple $lastTime$ and $plantV(lastTime)$ and the interval $[t1, t2]$ is equal to $[lastTime, t]$, so $desolve\_rk4$ will return the values of $plantV$ between $lastTime$ and $max(lastTime, t) = t$, where t represents the current instant of control. $grd8$ specifies that the values returned by $desolve\_rk4$ must be monotonous.

EVENT Plant_time_desolverk4
REFINES Plant_time
ANY $plant1$, $lastTime$
WHERE
grd6: $plant1 = B\_desolve\_rk4(f\_evol(ctrlV) \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)) \mapsto (lastTime \mapsto t))$
grd7: $\forall xx \cdot xx \in dom(plant1) \Rightarrow prop(plant1(xx)) = TRUE$
grd8: $\forall tt \cdot tt \in [lastTime, t] \Rightarrow (plant1(tt) \mapsto plant1(lastTime) \in geq \wedge plant1(tt) \mapsto plant1(t) \in leq) \wedge (plant1(tt) \mapsto plant1(lastTime) \in leq \vee plant1(tt) \mapsto plant1(t) \in geq)$
THEN
act1: $plantV := plantV \lhd\!\!+ plant1$
act2: $exec := ctrl$
END

Figure 7.14: Event Plant_time_desolverk4.

7.3.2 Choosing the Interval $[t1, t2]$

In mathematics, an autonomous system or autonomous differential equation is a system of ODEs which does not explicitly depend on the independent variable. When the variable is time, they are also called *time-invariant* systems. Many laws in physics, where the independent variable is usually assumed to be time, are expressed as autonomous systems. The solutions to the autonomous equation are monotonic functions. In particular, the first order autonomous equations cannot have periodic solutions. This guarantees that between two *TimeTriggered* sensors updates the function *plantV* is monotone as long as the controller has not yet updated the controlled variable *ctrlV*. The longest time between these updates is bounded by the symbolic duration *epsilon*. For example in the case of the *Smart Heating* system, the ordinary differential equation is expressed using the controlled variable $T' = ctrlV$ with $ctrlV \in \{temp, -temp\}$. If the controller choose to increase the temperature between the instant $t1$ and the instant $t2$ i.e $T' = temp$, then the values of T in $[t1, t2]$ are represented by monotonic function (the same for $T' = -temp$). In that case, to prove the system safety property in $[t1, t2]$ we just need to prove that $T(t2)$ satisfies the safety property $t \leq T_{max}$ ($t \geq T_{min}$).

To specify *TimeTriggered* sensors updates in Event-B, we calculate the new values of *plantV* using $B\_desolverk4$ in the interval $[lastTime, t]$. The controllers of such system must be triggered when the normal mode specified by the normal values, declared as a parameter of the event $Ctrl\_desolverk4$ (see Figure 7.15), satisfy the safety envelope within the period t to $t+epsilon$. t represents the current instant of control and $t+epsilon$ represents the next period of control (see $grd5$). Moreover, the results returned by $B\_desolverk4$ for the evade values including the normal values must satisfy the safety envelope between t and $t + 2 * epsilon$ in order to guarantee that the physical system does not exceed the safety property within two periods of control ($grd6$).

EVENT Ctrl_desolverk4
REFINES Ctrl
ANY *value*
WHERE
grd5 : $\forall x, tt \cdot x \in PROP \land tt \in [t, t + epsilon] \Rightarrow (value \notin prop\_evade\_values(x) \Rightarrow$
$(prop\_safeEpsilon(x))(B\_desolve\_rk4(f\_evol(value) \mapsto plantV \mapsto t \mapsto (t \mapsto plantV(t))$
$\mapsto (t \mapsto (t + epsilon)))(tt) \mapsto value) = TRUE)$
grd6 : $\forall x, tt \cdot x \in PROP \land tt \in [t, t + 2 \times epsilon] \Rightarrow (prop\_safeEpsilon(x))$
$(B\_desolve\_rk4(f\_evol(value) \mapsto plantV \mapsto t \mapsto$
$(t \mapsto B\_desolve\_rk4(f\_evol(value) \mapsto plantV \mapsto t \mapsto (t \mapsto plantV(t))$
$\mapsto (t \mapsto (t + epsilon)))(t)) \mapsto (t \mapsto (t + 2 \times epsilon))))(tt) \mapsto value) = TRUE)$
THEN
act1 : $ctrlV := value$
act2 : $exec := prg$
END

Figure 7.15: Event Ctrl_desolverk4.

7.3.3 Discussion

In this section, we presented a generic approach for modeling nonlinear hybrid systems using the predefined function *desolve_rk4*. The solutions returned by *desolve_rk4* are approximated solutions, they do not represent the exact behavior of the continuous variables of hybrid systems. Therefore, the proof of safety properties of nonlinear systems is more complicated to that of linear hybrid systems. Table 7.2 gives the statistics of the proof obligations generated to ensure the correctness of the Nonlinear generic models of our approach. As stated before, the modifications brought on the *Event-* and *Time-Triggered* models reduced the number of proof obligations compared to table 5.1.

Table 7.2: RODIN Proof Statistics for the Nonlinear Generic Models

Generic_Models	**Total**	**Automatic**	**Interactive**
ContSystem_M	8	1	7
EventTriggered_M	19	11	8
TimeTriggered_M	2	1	1
TimeTriggered_desolverk4_M	7	3	4

For the nonlinear hybrid systems, 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:

$$\forall tt \cdot tt \in [lastTime, t] \Rightarrow (plant1(tt) \geq plant1(lastTime) \wedge$$

$$plant1(tt) \leq plant1(t)) \vee (plant1(tt) \leq plant1(lastTime) \wedge plant1(tt) \geq plant1(t))$$

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

7.4 Conclusion

This chapter has presented a proof-based approach that 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 plugin, that establishes the link between RODIN and SAGEMATH. Our approach can be compared to the approach of the differential refinement logic dR$\mathcal{L}$ that requires mastering the syntax of other tools to validate the proof phase. For example it requires mastering the syntax of KeYmaera and KeYmaera X, that interface the theorem prover Mathematica, to prove the safety property of an *Event-Triggered* systems. Unlike dR$\mathcal{L}$, our proposed approach uses end-to-end the formal method EVENT-B to take advantage of its supported tools and its refinement strategy, always by coupling Rodin and SAGEMATH. To cope with the complexity of the system, the EVENT-B specification consists of three generic models: *EventTriggered*, *TimeTriggered* and *TimeTriggeredDesolve* that introduces the function *B_desolve* to model the call to a differential equations solver. The proposed approach extends the generic approach introduced in the previous chapter by adding a new generic set that permits to reduce the number of proof obligations. To model hybrid systems of which the continuous part is described by nonlinear ordinary differential equations, we introduce a new generic model that refines the *Time-Triggered* model by defining a new predefined function of SAGEMATH, *desolve_rk4*, that returns approximated solutions of first order ODEs. So far, we have been unable to find a first-order nonlinear case study to which the generic approach can be applied. To prove the feasibility of our general approach, we plan as a future work to add some complex properties to the linear case studies used in this work and then use the *B_desolve_rk4* function to find approximate solutions.

Chapter 8

Application

Contents

This chapter describes the application of our generic approaches for modeling the *Stop Sign*, the *Water Tank* and the *Smart Heating* case studies presented in Chapter 4. The rules described in Chapter 6 are used to demonstrate how a system specified with two continuous variables and a single safety property is instantiated, the case of the *Stop Sign* system. They are also used to describe how a system specified with two safety envelops, two set of evade values, two event trigger formulas is instantiated, the case of the *Water Tank* and *Smart Heating* systems.

Section 8.1 describes the modeling of the *Stop Sign* case study, for which we chose to directly model the controlled system by refining the *EventTriggered* model. For the *Water Tank* case study, we chose to start with an abstract model that refines the generic *ContSystem* model which is described in Section 8.2. Then, Section 8.3 discusses the proof activity resulting from the application of our generic approach on specific case studies. Section 8.4 describes the instantiation of the *Smart Heating* case study which starts by refining the generic *TimeTriggeredDesolve* model to illustrate the use of the function *B_desolve* on a

specific case study. Last, we describe briefly in Section 8.5 the modeling of the nonlinear case study, the *Inverted Pendulum*.

8.1 Stop Sign Models

The modeling of the *Stop Sign* case study follows the schema depicted by Figure 8.1, the whole models are available in Appendix B.

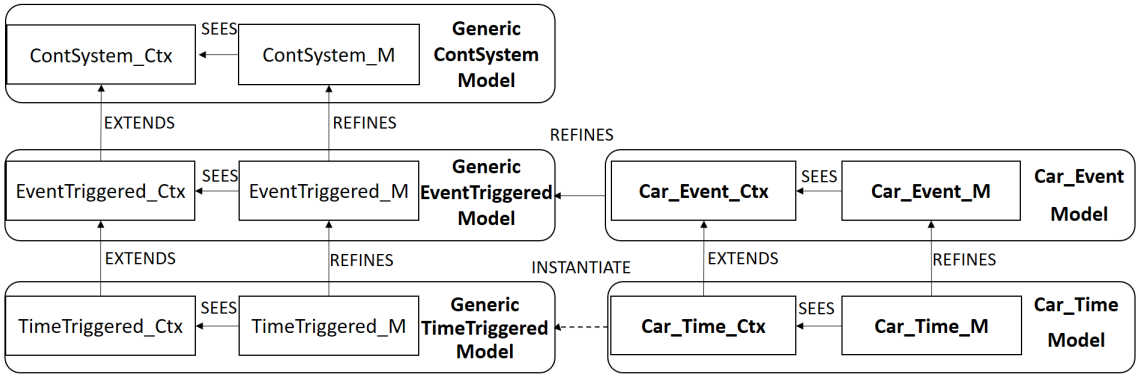

Figure 8.1: Architecture of the EVENT-B model of the Stop Sign.

8.1.1 Stop Sign EventTriggered Model

The instantiation starts by refining the generic *EventTriggered* model to obtain the specific *Car_Event* model represented by the *Car_Event_M* machine that sees the *Car_Event_Ctx* context depicted by Figure 8.2. Context *Car_Event_Ctx* is obtained following the set of rules *Rule_ET2_Ctx_i* defined in Section 6.2. As stated in Section 4.1, the continuous behavior of the *Stop Sign* case study is described by two state variables p and v. Following *Rule_ET2_Ctx_1*, the constant S is instantiated in the specific *ContSystem_Ctx* by $\mathbb{R} \times \mathbb{R}$. The constant *plantVInit* is replaced by the constants *pinit* and *vinit* that respectively specify the initial position and the initial velocity, see $axm1 - 2$. $axm3$ specifies the properties of the initial values *pinit* and *vinit* as well the stop sign constant SP. It also guarantees that the system is initially safe.

The *Stop Sign* case study is composed of a single safety envelop which guarantees that the car will never exceed the SP limit. This safety property is associated with an event trigger formula and a set of evade values instantiated following *Rule_ET2_Ctx_2*, see axioms $axm4$, $axm5$ and $axm7$. The value of *evt_trig* is defined by considering the distance traveled by the car during a time period of x $(1/2 \times ctrlV \times x^2 + v \times x)$ plus the distance traveled from the time the car must brake to stop just before SP $(v^2/2B)$. $axm6$ specifies the specific ODEs associated with the case study, $\frac{dp}{dt} = v$ and $\frac{dv}{dt} = ctrlV$, using the function *f_evol_plantV* (*Rule_ET2_Ctx_3*), where $ctrlV$ represents the discrete measurement and p and v represent the continuous measurements. $axm8$ and $axm9$ specify the properties of *val_i*, the maximum limit of acceleration A and the maximum limit of braking B, which are used to specify the *normal* and *evade* modes. Finally, the *Stop Sign* safety property, $p \leq SP$, is specified in $axm10$ using the constant *prop*.

The specific machine *Car_Event_M* refines the generic machine *EventTriggered_M* by following the rules, *Rule_ET2_M_1* and *Rule_ET2_M_2* described in Section 6.2. We replace the generic state variable *plantV* by the physical state variables associated with the *Stop Sign* case study, p and v. This substitution is done using the operator *bind* of the

CONTEXT Car_Event_Ctx
EXTENDS EventTriggered_Ctx
CONSTANTS SP, $pinit$, $vinit$, A, B
AXIOMS
axm1: $pinit \in \mathbb{R}^+ \land vinit \in \mathbb{R}^+$
axm2: $(pinit, vinit) = plantVInit$
axm3: $pinit \leq SP \land SP \in \mathbb{R} \land 0 < SP \land pinit + \frac{vinit \times vinit}{2 \times B} \leq SP$
axm4: $safe = (\lambda(p \mapsto v) \mapsto ctrlV \cdot (p \mapsto v) \in (\mathbb{R} \times \mathbb{R}) \land ctrlV \in \mathbb{R} \mid bool(p + v^2/2B < SP))$
axm5: $evt\_trig = (\lambda(p \mapsto v) \mapsto (t1 \mapsto ctrlV) \cdot (p \mapsto v) \in (\mathbb{R} \times \mathbb{R}) \land$
$t1 \in TIME \land ctrlV \in \mathbb{R} \mid bool(p + 1/2 \times ctrlV \times t1^2 + v \times t1 + v^2/2B \leq SP))$
axm6: $\forall ctrlV \cdot ctrlV \in \mathbb{R} \implies (f\_evol\_plantV(ctrlV) =$
$(\lambda\, t \mapsto (p \mapsto v) \cdot t \in TIME \land (p \mapsto v) \in (\mathbb{R} \times \mathbb{R}) \mid (v \mapsto ctrlV)))$
axm7: $evade\_value = \{-B, 0\}$
axm8: $A \in \mathbb{R} \land 0 < A$
axm9: $B \in \mathbb{R} \land 0 < B$
axm10: $prop = (\lambda\, p.p \in \mathbb{R} \mid bool(p \leq SP))$.
END

Figure 8.2: Context Car_Event_Ctx.

DiffEq theory in the *INVARIANTS* clause, see $inv3$ [1] in Figure 8.3. The *INVARIANTS* clause also contains the possible values of the controlled variable $ctrlV$ ($inv1$) the definition of p and v ($inv2$) as well as the properties necessary to facilitate the proof phase and which instantiate $inv4$ and $inv5$ of Figure 6.34.

INVARIANTS
inv1: $ctrlV \in \{0, -B, A\}$
inv2: $p \in [0,t] \nrightarrow \mathbb{R} \land v \in [0,t] \nrightarrow \mathbb{R}^+ \land dom(v) = dom(p)$
inv3: $plantV = bind(p, v)$
inv4: $exec \neq plant \implies dom(p) = [0,t] \land dom(v) = [0,t]$
inv5: $exec = plant \implies t \notin dom(plantV)$
inv6: $\forall x \cdot x \in dom(p) \implies prop(p(x)) = TRUE$

Figure 8.3: Stop Sign EventTriggered INVARIANTS.

The transitions between states of the hybrid automaton of Figure 4.2 are represented in our approach by the event *Plant_event_car* (Figure 8.4) which describes the evolution of the physical part represented by the state variables p and v. This event is obtained following *Rule_ET2_M_2*. It refines the *Plant* event of the generic model (Figure 5.9) by exhibiting a witness that replaces the evolution of the generic state parameter $plant1$ by that of the parameters $p1$ and $v1$. These parameters represent respectively the evolution of the position and the velocity from the last moment until the current time t. Then the values of the functions p and v are overridden by those of the parameters $p1$ and $v1$ in $act1$ and $act2$.

Rule_ET2_M_2 also instantiates the events *Ctrl_normal* and *Ctrl_evade*. The *Stop Sign* case study has three discrete states, *Accelerating* state which corresponds to the *normal* mode, *Braking* and *Stopped* states which correspond to the *evade* mode. The transition from *Braking* to *Accelerating* states is modeled using the event *Ctrl_Acceleration_car* (Figure 8.5) which refines the generic *Ctrl_normal* event by replacing *plantV* by p and v and the *nrml_value* by A in the formula $safe$. The transition from *Accelerating* to *Braking* or *Stopped* states is modeled by the event *Ctrl_Deceleration_car* (Figure 8.6) that replaces the value of *evade_val* by $-B$ if $v(t) > 0$ or by 0 if $v(t) = 0$.

As it is easier to prove the safety property at this level, we express it as an invariant $inv6$ of Figure 8.3, where we specified $prop(p(x))$ in the context *Car_Event_Ctx* (see $axm10$).

[1] $bind(p, v) = p \mapsto v$

EVENT Plant_event_car **REFINES** Plant
ANY $p1$, $v1$
WHERE
grd1 : $exec = plant$
grd2 : $p1 \in [0,t] - dom(p) \rightarrow \mathbb{R} \land\ v1 \in [0,t] - dom(p) \rightarrow \mathbb{R}^+$
grd3 : $ode(f\_evol\_plantV(ctrlV), (p1(t), v1(t)), t) \in DE(\mathbb{R} \times \mathbb{R})$
grd4 : $Solvable([0,t] - dom(bind(p,v)), ode(f\_evol\_plantV(ctrlV), bind(p1,v1)(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV),$
$(bind(p1,v1))(t), t), [0,t] - dom(bind(p,v)), [0,t] - dom(bind(p,v)), bind(p1,v1))$
grd6 : $\forall xx \cdot xx \in dom(p1) \implies prop(p1(xx)) = TRUE$
WITH $plant1 : plant1 = bind(p1,v1)$
THEN
act1 : $p := p \lhd p1$
act2 : $v := v \lhd v1$
act3 : $exec := ctrl$
END

Figure 8.4: Event Plant_event_car.

EVENT Ctrl_Acceleration_car **REFINES** Ctrl_normal
WHERE
grd1 : $exec = ctrl$
grd2 : $safe(bind(p,v)(t), A) = TRUE$
WITH $nrml\_value : nrml\_value = A$
THEN
act1 : $ctrlV := A$
act2 : $exec := prg$
END

Figure 8.5: Event Ctrl_Acceleration_car.

EVENT Ctrl_Deceleration_car **REFINES** Ctrl_evade
ANY $evade\_val$
WHERE
grd1 : $exec = ctrl$
grd2 : $evade\_val \in evade\_value$
grd3 : $v(t) > 0 \implies evade\_val = -B$
grd4 : $v(t) = 0 \implies evade\_val = 0$
THEN
act1 : $ctrlV := evade\_val$
act2 : $exec := prg$
END

Figure 8.6: Event Ctrl_Deceleration_car.

To be sure that this invariant will be preserved by the event *Plant_event_car*, we added the guard *grd*6 (see Figure 8.4) to ensure that the new value of the position $p1$ verifies the system safety property. This guard is necessary at this level because we do not consider the solution of the system ODEs yet. It will be removed in the *TimeTriggered* model and refinement proofs (*GRD* type) will be generated to ensure that the actual value of $p1$ does satisfy it (Section 8.1.3).

8.1.2 Stop Sign TimeTriggered Model

We refine the *Car_ Event* model to produce the *Car_ Time* model. Machine *Car_ Time_ M* sees the context *Car_ Time_ Ctx* (Figure 8.7) which is an instantiation of *TimeTriggered_ Ctx* context. Context Car_Time_Ctx is obtained following the two rules, *Rule_ TT_ Ctx_ 1* and *Rule_ TT_ Ctx_ 2*, represented in Section 6.1.3. It defines the formula $safeEpsilon$, $axm2 - axm3$, as the longest position of the car after an acceleration phase of *epsilon* unit of time followed by a braking phase until it stops [2]:

CONTEXT Car_Time_Ctx **EXTENDS** Car_Event_Ctx
CONSTANTS epsilon , safeEpsilon
AXIOMS
axm1 : $epsilon \in TIME \land sigma \leq epsilon \land\ 0 < epsilon$
axm2 : $safeEpsilon \in (\mathbb{R} \times \mathbb{R}) \times \mathbb{R} \rightarrow BOOL$
axm3 : $safeEpsilon = \lambda(p \mapsto v) \mapsto ctrlV \cdot (p \mapsto v) \in (\mathbb{R} \times \mathbb{R}) \land ctrlV \in \mathbb{R} \mid p+$
$(A \times epsilon^2/2 + v \times epsilon) + v^2/2B + (A^2 \times epsilon^2/2B) + (A \times v \times epsilon/B) < SP$
END

Figure 8.7: Context Car_Time_Ctx.

Machine *Car_ Time_ M* is an instantiation of the generic machine *TimeTriggered_ M*. The instantiation is done following the set of rules *Rule_ TT_ M_ i* described in Section 6.1.3. The *INVARIANTS* clause (Figure 8.8) defines the properties needed to ensure that during a control period *epsilon* the system remains safe as described by *Rule_ TT_ M_ 1*. $inv1$, $inv2$ and $inv3$ (Figure 6.23) are instantiated by replacing the generic parameters by those associated with the *Stop Sign* case study. Moreover, we added two invariants, $inv4$ and $inv5$, which are specific to the *Stop Sign* case study. $inv4$ is a bit special. It ensures that the safety property is satisfied between two control periods i.e the evolution of the physical part represented by $p(t1)$ and $v(t1)$ must preserve this safety property. $inv5$ is defined to guarantee that the car is stopped when the acceleration is equal to zero.

INVARIANTS
$inv1 : \exists\, t1 \cdot t1 \in TIME \land dom(p) = [0, t1] \land t - t1 \leq epsilon \land (exec \neq plant \implies t1 = t) \land$
$(exec = plant \implies t > t1) \land (ctrlV \notin evade\_value \land exec = plant$
$\implies safeEpsilon((p(t1) \mapsto v(t1)) \mapsto A) = TRUE)$
$inv2 : \forall\, t1 \cdot (t1 \in TIME \land dom(p) = [0, t1] \implies p(t1) + \frac{v(t1) \times v(t1)}{2B} \leq SP)$
$inv3 : ctrlV \notin evade\_value \land exec = prg \implies safeEpsilon((p(t) \mapsto v(t)) \mapsto A) = TRUE$
$inv4 : \forall\, t1 \cdot t1 \in TIME \land dom(p) = [0, t1] \land ctrlV = 0 \land exec \neq ctrl \implies v(t1) = 0$
$inv5 : \forall t1, t2 \cdot t1 \in TIME \land t2 \in TIME \land dom(p) = [0, t1] \land dom(p) = [0, t2] \implies t1 = t2$

Figure 8.8: Stop Sign TimeTriggered Invariants.

Following *Rule_ TT_ M_ 2*, we refine the event *Progress* associated with the machine *Car_ Event_ M* to add the control period *epsilon*, such as $t1 - t \leq epsilon$ (see Figure 8.9), and remove the following guard: $ctrlV \notin evade\_value \implies evt\_trig((bind(p, v)(t), (t1 - t), ctrlV) = TRUE$. Removing such a guard gives rise to a proof obligation to ensure that it is satisfied, see Section 8.1.3. This proof obligation corresponds to POs *PO2_ dRℒ* and *PO3_ dRℒ* of dRℒ.

The continuous part is modeled by the event *Plant_ time_ car* (see Figure 8.10). *Plant_ time_ car* refines the event *Plant_ event_ car* to calculate the values of $p1$ and $v1$ during the period from *lastTime* to t where *lastTime* denotes the last moment at which p and v have been calculated with $dom(p) = [0, lastTime]$ (*Rule_ TT_ M_ 3*). The parameter $epsilon1$ is defined to represent the maximum duration of the car movement

[2] During the acceleration phase: the position increases by $(A \times epsilon^2/2 + v \times epsilon)$, the speed becomes equal to $(A \times epsilon + v)$. In the braking phase, the car stops in $(t_1 = (A \times epsilon + v)/B)$ units of time. So, the final position of the car is equal to $p + (A \times epsilon^2/2 + v \times epsilon) + (-Bt_1{}^2 + t_1((A \times epsilon + v)/B)$.

EVENT Progress **REFINES** Progress
ANY $t1$
WHERE
grd1 : $exec = prg$
grd2 : $t1 \in TIME \wedge (t < t1 \wedge t1 - t \geq sigma)$
grd3 : $(t1 - t) \leq epsilon$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Figure 8.9: Car Time Progress.

until its stop. In the *evade* mode ($ctrlV = -B$), the car is allowed to move forward during a period $epsilon1$ equal to the maximum of $\frac{v(lastTime)}{B}$ (the moment at which its speed becomes null) and $(t - lastTime)$. Otherwise, this period is equal to $(t - lastTime)$. As stated before, the guard $grd6$ of the $Plant\_event\_car$ has been removed, instead a proof obligation is generated to ensure that it is induced by the actual values of $p1$ and $v1$, see Section 8.1.3. The actual values of $p1$ and $v1$ are calculated by guards $grd6$ and $grd7$ that specify the results of solving the ODEs, $\frac{dp}{dt} = v(t)$ and $\frac{dv}{dt} = ctrlV$ taking in consideration the period of control $epsilon1$. The result of solving $\frac{dv}{dt} = ctrlV$ is $v(t) = ctrlV \times epsilon1 + v(lastTime)$. Substituting v by this value in $\frac{dp}{dt} = v$ gives $p(t) = \frac{1}{2} \times ctrlV \times epsilon1^2 + v(lastTime) \times epsilon1 + p(lastTime)$. Guard $grd10$ instantiates the $grd6$ defined in Figure 6.25 by replacing the generic parameters by the specific ones.

EVENT Plant_time_car **REFINES** Plant_event_car
ANY $p1$, $v1$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \wedge\ dom(p) = [0, lastTime]$
grd3 : $lastTime \in dom(p) \wedge\ lastTime \in dom(v)$
grd4 : $ctrlV = -B \implies (t - lastTime \leq \frac{v(lastTime)}{B}$
$\implies epsilon1 = t - lastTime) \wedge (t - lastTime > \frac{v(lastTime)}{B} \implies epsilon1 = \frac{v(lastTime)}{B})$
grd5 : $ctrlV \in \{0, A\} \implies epsilon1 = t - lastTime$
grd6 : $p1 = (\lambda t1 \cdot t1 \in TIME \wedge t1 > lastTime \wedge t1 \leq t | (p(lastTime)$
$+(\frac{1}{2} \times (ctrlV \times (epsilon1^2)))) + (v(lastTime) * epsilon1))$
grd7 : $v1 = (\lambda t1 \cdot t1 \in TIME \wedge t1 > lastTime \wedge t1 \leq t\ | (ctrlV * epsilon1) + v(lastTime))$
grd8 : $ode(f\_evol\_plantV(ctrlV), (p1(t) \mapsto v1(t)), t) \in DE(\mathbb{R} \times \mathbb{R})$
grd9 : $Solvable([0, t] - dom(bind(p, v)), ode(f\_evol\_plantV(ctrlV), bind(p1, v1)(t), t))$
grd10 : $solutionOf([0, t] - dom(bind(p, v)), ([0, t] - dom(bind(p, v))) \lhd bind(p1, v1),$
$ode(f\_evol\_plantV(ctrlV), (p1(t), v1(t)), t))$
THEN
act1 : $p := p \Leftarrow p1$ act2 : $v := v \Leftarrow v1$
act3 : $exec := ctrl$
END

Figure 8.10: Event Plant_time_car.

We refine the event $Ctrl\_Acceleration\_car$ to instantiate the event $Ctrl\_normal\_time$ (Figure 5.13) by replacing the formula $safe((p(t), v(t)), A)$ by the formula $SafeEpsilon$ ($Rule\_TT\_M\_4$). Let us remark, that the guard related to the satisfaction of $safe$ has been removed to give rise to a proof obligation that verifies that $safeEpsilon$ induces $Safe$. Event $Ctrl\_Deceleration\_car$ remains as defined in machine Car_Event_M.

EVENT Ctrl_Acceleration_car_time
REFINES Ctrl_Acceleration_car
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilon(p(t), v(t), A) = TRUE$
THEN
act1 : $ctrlV := A$
act2 : $exec := prg$
END

Figure 8.11: Event Ctrl_Acceleration_car_time.

8.1.3 Correctness of the Specification

Table 8.1: RODIN proof statistics for the Stop Sign system

Specific_Models	Total	Automatic	Interactive
Car_Event_M	44	17	27
Car_Time_M	42	13	29

Table 8.1 gives the statistics of the POs generated for modeling the *Stop Sign* case study. All the POs are discharged using the automatic/interactive provers of the RODIN platform and the theories of [10]. It is noticeable that 34% of them were automatically discharged. The POs that are independent from any specific case study are simply discharged by replying the same proof script defined in the proof of the generic models. These POs are related to the correctness of the event Progress, i.e feasibility of the event, and also the type of the variables. Moreover, the POs related to the guards feasibility and well-definedness are much easier to discharge than those related to ensure the system safety property and also that concerning the preservation of formula $safe$ that we detail hereafter.

Proof of the safety property: Recall that the safety property has been expressed as an invariant in the *Car_Event_M* machine: $\forall x \cdot x \in dom(p) \implies prop(p(x)) = TRUE$, where $prop(p(x)) = p(x) \leq SP$. This invariant generates the following proof obligation for the event *Plant_event_car* that updates the value of the variable p:

$$\forall x \cdot x \in dom(p \lhd p1) \implies (p \lhd p1)(x) \leq SP$$

Since at this level the ordinary differential equation has not been resolved yet, the concrete value of $p1$ is not known. For this purpose, we added the guard $grd6$ in the event *Plant_event_car*. This guard states that the concrete value of $p1$ should be such that the future position of the car is always before the stop signal SP. Thus at this level, the safety property is proved under the guard $grd6$. This guard is removed by refinement in the *Car_Time_M* machine and gives thus rise to a proof obligation that verifies the concrete value of $p1$, obtained after the resolution of the ordinary differential equations ($grd6$ and $grd7$ of the event *Plant_time_car*), does satisfy $grd6$ of *Plant_event_car*. In that way, we definitely proved the safety property as we establish that the guard $grd6$ of the event *Plant_event_car* is true. To prove this, we added the following invariants in *Car_Time_M* machine:

- When the variable $ctrlV$ is updated to be equal to A, then $safeEpsilon$ is verified,

where t_1 is such that $dom(p) = [0, t_1]$.

$$exec = plant \wedge ctrlV \neq evade\_value \implies safeEpsilon((p(t_1), v(t_1)), A) = TRUE$$

- According to [16], at any moment t_1, the position of the car permits to brake before SP. In fact $v(t_1)^2/2B$ denotes the maximum distance that the car can cover when it enters into the braking phase. Indeed, when the car brakes at the instant t_1, it continues to move forwards during $(V(t_1)/B)$ units of time. So, the car will cover a distance equal to $(-1/2 \times B \times (V(t_1)/B)^2 + V(t_1) \times (V(t_1)/B) = v(t_1)^2/2B$

$$\forall t_1.t_1 \in \mathbb{R}^+ \wedge dom(p) = [0, t_1] \implies p(t_1) + v(t_1)^2/2B \leq SP$$

Preservation of the predicate $safe$ by refinement The generic modeling of the event $Ctrl\_normal\_time$ (Figure 5.13) contains a guard to check that $safe$ is fulfilled, $grd3$ of this event, in addition to its dual guard related to $safeEpsilon$. As stated before, for a specific application, $grd3$ is skipped which gives rise to a refinement proof obligation to ensure that the removed guard can be induced from that of the event $Ctrl\_normal\_time$. In other words, this means that the guard of the event $Ctrl\_normal\_time$ must be stronger than that of the event $Ctrl\_normal\_event$. This refinement proof obligation corresponds to $PO1\_dR\mathcal{L}$ of dR$\mathcal{L}$.

$$\begin{aligned} &... \wedge safeEpsilon(plantV(t), ctrlV) = TRUE \\ &\implies safe(plantV(t), nrml\_value) = TRUE \end{aligned}$$

Let us note that such proof obligations have been interactively discharged under Rodin thanks to different provers like SMT and AtelierB provers but also the inference rules described in the 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. For instance, the formula $a \leq c$ under the hypotheses $a \leq b \wedge b \leq c$, with a, b, c denoting real expressions, cannot be discharged automatically and requires the intervention of the user that must explicitly apply the transitivity rules included in the theory of reals. To speed up the proof activity, it would be interesting to develop an automatic prover around the theory of reals whose objective is to automatically apply the existing inference rules to produce new hypotheses. For the above example, hypothesis $a \leq c$ should be automatically inferred by applying the transitivity inference rule. The development of such a prover is one of the objectives of the EBRP project [9].

Preservation of evt_trig by refinement Recall that the generic modeling of the event $Progress$, in the timed model, contains a guard to ensure that time progresses without going beyond the safety envelope boundaries evt_trig. This guard is omitted by instantiation in the $TimeTriggered$ model and the following proof obligation is generated instead it corresponds to $PO2\_dR\mathcal{L}$ and $PO3\_dR\mathcal{L}$ of dR$\mathcal{L}$:

$$exec = prg \implies evt\_trig((bind(p, v))(t), t_1 - t, ctrlV)$$

To prove the above PO, we have added and proved the following invariant that states that before making the time progress, if the normal mode is chosen, then the system is safe:

$$ctrlV \notin evade\_value \wedge exec = prg \implies safeEpsilon((p(t) \mapsto v(t)) \mapsto ctrlV) = TRUE$$

Let us remark that expressions of evt_trig and $safeEpsilon$ are very similar: $safeEpsilon$ depends on $epsilon$ while evt_trig depends on $(t_1 - t)$. By rewriting $(evt\_trig = E_1 \leq SP)$ and $(safeEpsilon = E_2 < SP)$, it suffices to prove that $E_1 \leq E_2$ with the hypothesis that $(t_1 - t \leq epsilon)$.

8.2 Water Tank Models

To model the *Water Tank* case study in EVENT-B, we proceed in three refinement steps (Figure 8.12). We start by refining the abstract generic *ContSystem* model to obtain the *Abstract_ Tank* model. In the second step, we produce the *Tank_ Event* model together with the safety property by refining the *Abstract_ Tank* model and instantiating the generic *EventTriggered* model. The last step consists in instantiating the generic *TimeTriggered* model and refining the *Tank_ Event* model to obtain the *Tank_ Time* model. the whole models are available in Appendix C.

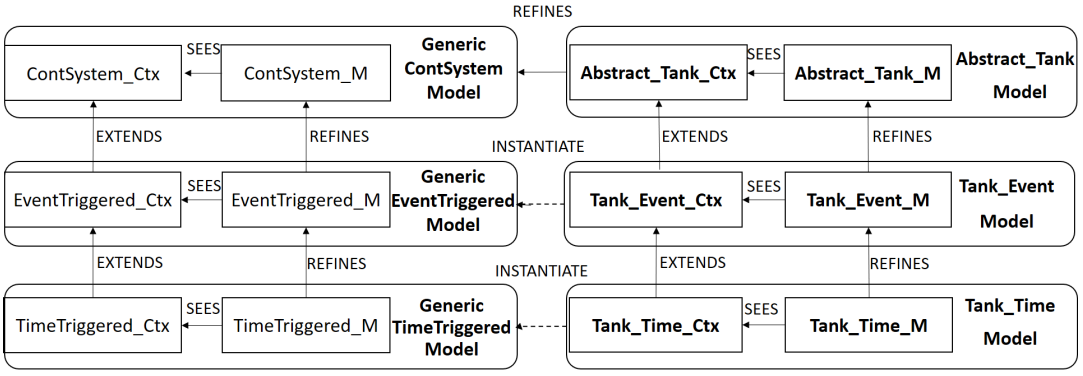

Figure 8.12: Architecture of the EVENT-B Model of the Water Tank.

8.2.1 Abstract Water Tank Model

To model the abstract *Water Tank* model in EVENT-B, we refine the generic abstract *ContSystem* model. *Abstract_ Tank* model consists of an abstract context, *Abstract_ Tank_ Ctx*, and an abstract machine, *Abstract_ Tank_ M*. Context *Abstract_ Tank_ Ctx* (Figure 8.13) is obtained following the rules *Rule_ CS_ Ctx_ i* (Section 6.1.1). It defines all the properties of the *Water Tank* case study such as the property $0 < V\_low < V\_high$, see $axm1 - axm3$. $axm4 - axm5$ specify the ordinary differential equation that describes the evolution of the water level using the specific function f_evol_V.

CONTEXT Abstract_Tank_Ctx
EXTENDS ContSystem_Ctx
CONSTANTS $V0$, V_high, V_low, f_evol_V
AXIOMS
axm1: $V0 \in \mathbb{R}^+$
axm2: $V0 = plantVInit$
axm3: $V\_high \in \mathbb{R} \land V\_low \in \mathbb{R} \land V0 < V\_high \land V0 > V\_low \land$
$V\_high > V\_low \land V\_low > 0$
axm4: $f\_evol\_V \in \mathbb{R} \rightarrow (TIME \times \mathbb{R} \rightarrow \mathbb{R})$
axm5:
$\forall\, ctrlV \cdot ctrlV \in \mathbb{R} \implies (f\_evol\_V(ctrlV) = (\lambda\, t \mapsto vol \cdot t \in TIME \land vol \in \mathbb{R} | ctrlV))$
END

Figure 8.13: Context Abstract_Tank_Ctx.

The *Abstract_ Tank_ M* machine refines the *ContSystem_ M* machine following the set of rules *Rule_ CS_ M_ i* (Section 6.1.1). It instantiates (renaming) the generic continuous variable *plantV* and the generic continuous parameter *plant*1 respectively by the specific continuous variable Vol and the specific parameter $Vol1$. Event *Water_ behave* (Figure

8.14) describes the evolution of the physical part by assigning the specific parameter $Vol1$ to the specific continuous variable Vol, see $act1$.

EVENT Water_behave **REFINES** Plant
ANY e, $Vol1$
WHERE
grd1: $e \in DE(\mathbb{R})$
grd2: $Solvable([0,t] - dom(Vol), e)$
grd3: $Vol1 \in [0,t] - dom(Vol) \rightarrow \mathbb{R}^+ \land AppendSolutionBAP(e, [0,t] - dom(Vol), [0,t] - dom(Vol), Vol1)$
WITH $plant1 : plant1 = Vol1$
THEN act1: $Vol := Vol \lhd\!\!+ Vol1$
END

Figure 8.14: Event Water_behave.

8.2.2 Water Tank EventTriggered Model

The procedure for modeling the *Water Tank* case study is similar to that described in Section 6.1.2 for modeling the *Smart Heating* system. *Tank_Event* model refines *Abstract_Tank* model by defining a more specific machine *Tank_Event_M* that refines the machine *Abstract_Tank_M*. Machine *Tank_Event_M* sees the context *Tank_Event_Ctx* (Figure 8.15) which adds new constants needed to model the interaction between the physical and the continuous parts of the *Water Tank* case study (*Rule_ET1_Ctx_1*). It also introduces the safety envelopes $safeFill$ and $safeEmp$ (*Rule_ET1_Ctx_2*).

CONTEXT Tank_Event_Ctx **EXTENDS** Abstract_Tank_Ctx
SETS EXEC
CONSTANTS $ctrl$, $plant$, prg, $safeFill$, $safeEmp$, $evt\_TrigFill$, $evt\_TrigEmp$, f_in, f_out, $evade\_valueFill$, $evade\_valueEmp$, $prop$
AXIOMS
axm1: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm2: $safeFill \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm3: $safeEmp \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm4: $evt\_TrigFill \in (\mathbb{R} \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm5: $evt\_TrigEmp \in (\mathbb{R} \times TIME) \times \mathbb{R} \rightarrow BOOL$
axm6: $safeFill = (\lambda\ vol \mapsto ctrlV \cdot vol \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(vol < V\_high))$
axm7: $safeEmp = (\lambda\ vol \mapsto ctrlV \cdot vol \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(vol > V\_low))$
axm8: $evt\_trigFill = (\lambda\ vol \mapsto t1 \mapsto ctrlV \cdot vol \in \mathbb{R} \land t1 \in TIME \land ctrlV \in \mathbb{R} \mid bool(vol + ctrlV \times t1 \leq V\_high))$
axm9: $evt\_trigEmp = (\lambda\ vol \mapsto t1 \mapsto ctrlV \cdot vol \in \mathbb{R} \land t1 \in TIME \land ctrlV \in \mathbb{R} \mid bool(vol + ctrlV \times t1 \geq V\_low))$
axm10: $f\_in \in \mathbb{R} \land f\_in > 0$
axm11: $f\_out \in \mathbb{R} \land f\_out > 0$
axm12: $evade\_valueFill \subseteq \mathbb{R} \land evade\_valueFill = \{-f\_out\}$
axm13: $evade\_valueEmp \subseteq \mathbb{R} \land evade\_valueEmp = \{f\_in\}$
axm14: $prop \in \mathbb{R} \rightarrow BOOL$
axm15: $prop = (\lambda vol \cdot vol \in \mathbb{R} | bool(vol \leq V\_high \land vol \geq V\_min))$
END

Figure 8.15: Context Tank_Event_Ctx.

As stated in Section 4.2, we express the safety property in a conjunctive normal form: $(V\_low \leq Vol \land Vol \leq V\_high)$. For the safety property $V\_low \leq Vol$ (resp. $Vol \leq V\_high$), the evade value is $\{f\_in\}$ (resp. $\{-f\_out\}$). Following *Rule_ET1_Ctx_3*,

Rule_ET1_Ctx_4 and *Rule_ET1_Ctx_5*, we define the following constants $safeEmp$, $evt\_trigEmp$, $safeFill$, $evt\_trigFill$, $evade\_valueFill$ and $evade\_valueEmp$ (axioms $axm6-9$ and $axm12-13$). Last $axm14-15$ are used to specify the *Water Tank* safety property by defining the constant *prop*.
Since the *Tank_Event* model instantiates the *EventTriggered* model, the *INVARIANTS* clause must include the system safety property specified using the constant *prop*. Moreover, we instantiate the invariants specified in Figure 6.10 by replacing the generic parameters by the specific ones and by following *Rule_ET1_M_2*. In the *Tank_Event* model, we refine the *Progress* event by adding the following guards (*Rule_ET1_M_3*):

$$evt\_trigFill(Vol(t) \mapsto (t_1 - t) \mapsto ctrlV) = TRUE$$
$$evt\_trigEmp(Vol(t) \mapsto (t_1 - t) \mapsto ctrlV) = TRUE.$$

Following the rules, *Rule_ET1_M_5* and *Rule_ET1_M_6*, we define three control events *Ctrl_normal* (Figure 8.16), *Ctrl_emptying* (Figure 8.17) and *Ctrl_filling* (Figure 8.18). Event *Ctrl_normal* represents the *normal* mode and is triggered when the safety envelopes $safeEmp$ and $safeFill$, associated to these properties, are satisfied which means that the system does not change its discrete state (*Filling* or *Emptying*) when the current water level evolves between V_high and V_low. Note that, as we only have two values f_in and $-f\_out$, we rewrite the condition $nCtrlV \notin \{-f\_out\}$ (resp. $nCtrlV \notin \{f\_in\}$) of $grd3$ (resp. $grd4$) into $nCtrlV = f\_in$ (resp. $nCtrlV = -f\_out$). The *evade* mode is refined by two events *Ctrl_emptying* and *Ctrl_filling*. *Ctrl_emptying* and *Ctrl_filling* can be triggered respectively when $safeEmp$ and $safeFill$ are verified.

EVENT Ctrl_normal
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{f\_in, -f\_out\}$
grd3 : $nCtrlV = f\_in \implies safeFill(Vol(t), f\_in) = TRUE$
grd4 : $nCtrlV = -f\_out \implies safeEmp(Vol(t), -f\_out) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Figure 8.16: Event Ctrl_normal.

EVENT Ctrl_emptying
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEmp(Vol(t), -f\_out) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := -f\_out$
END

Figure 8.17: Event Ctrl_emptying.

The continuous part is represented by the event *Plant_event_tank* (Figure 8.19) that refines the event *Water_behave*. It introduces the evolution of the water level using the continuous function *f_evol_V(ctrlV)* (*Rule_ET1_M_4*). Moreover, it specifies the safety property of the *Water Tank* case study $V\_low \leq Vol(t) \leq V\_high$ (see $grd6$), so the continuous part will be triggered iff the formula $prop(Vol1(xx)) \equiv Vol1(xx) \leq V\_high \wedge Vol1(xx) \geq V\_low$ is satisfied.

EVENT Ctrl_filling
WHERE
grd1 : $exec = ctrl$
grd2 : $safeFill(Vol(t), f\_in) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := f\_in$
END

Figure 8.18: Event Ctrl_filling.

EVENT Plant_event_tank
REFINES Water_behave
ANY $Vol1$
WHERE
grd1 : $exec = plant$
grd2 : $Vol1 \in [0,t] - dom(Vol) \rightarrow \mathbb{R}^+$
grd3 : $ode(f\_evol\_V(ctrlV), Vol1(t), t) \in DE(\mathbb{R})$
grd4 : $Solvable([0,t] - dom(Vol), ode(f\_evol\_V(ctrlV), Vol1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_V(ctrlV), Vol1(t), t),$
$[0,t] - dom(Vol), [0,t] - dom(Vol), Vol1)$
grd6 : $\forall xx \cdot xx \in dom(Vol1) \implies prop(Vol1(xx)) = TRUE$
WITH $e : e = ode(f\_evol\_V(ctrlV), Vol1(t), t)$
THEN
act1 : $Vol := Vol \Leftarrow Vol1$
act2 : $exec := prg$
END

Figure 8.19: Event Plant_event_tank.

8.2.3 Water Tank TimeTriggered Model

The *Tank_Time* model is obtained by refining the *Tank_Event* model. The static part of this model is represented by the context *Tank_Time_Ctx* (Figure 8.20). Context *Tank_Time_Ctx* extends the context *Tank_Event_Ctx* and instantiates the generic context *TimeTriggered_Ctx* (*Rule_TT_Ctx_1* and *Rule_TT_Ctx_2*). It adds the definition of the control period *epsilon*, see $axm1$, as well as the system safety properties where $safeEpsilonEmp$ and $safeEpsilonFill$ are defined in $axm4$ and $axm5$.

CONTEXT Tank_Time_Ctx
EXTENDS Tank_Event_Ctx
CONSTANTS $epsilon, safeEpsilonFill, safeEpsilonEmp$
AXIOMS
axm1 : $epsilon \in TIME \land 0 < epsilon \land sigma \leq epsilon$
axm2 : $safeEpsilonFill \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm3 : $safeEpsilonEmp \in (\mathbb{R} \times \mathbb{R}) \rightarrow BOOL$
axm4 : $safeEpsilonEmp = (\lambda\ vol \mapsto ctrlV \cdot vol \in \mathbb{R} \land ctrlV \in \mathbb{R} |$
$bool(ctrlV \times epsilon \geq V\_low))$
axm5 : $safeEpsilonFill = (\lambda\ vol \mapsto ctrlV \cdot vol \in \mathbb{R} \land ctrlV \in \mathbb{R} |$
$bool(ctrlV \times epsilon \leq V\_high))$
END

Figure 8.20: Context Tank_Time_Ctx.

The dynamic part of the *Tank_Time* model is represented by the *Tank_Time_M* machine which refines the *Tank_Event_M* machine. The *Tank_Time INVARIANTS*

clause (Figure 8.21) defines all the properties needed to preserve the system safety property during the control period *epsilon* (*Rule_ TT_ M_ 1*), see *inv*1. For each safety envelope $safeEpsilonFill$ and $safeEpsilonEmp$, it defines a set of properties that must be preserved during the progression of time, see $inv2 - inv3$.

INVARIANTS
inv1 : $\exists\, t1 \cdot t1 \in TIME \wedge dom(Vol) = [0, t1] \ \wedge t - t1 \leq epsilon \wedge (exec \neq plant \implies t1 = t) \wedge$
$(exec = plant \implies t > t1) \wedge (ctrlV \notin evade\_valueFill \wedge exec = plant \implies$
$safeEpsilonFill(Vol(t1) \mapsto ctrlV) = TRUE) \wedge (ctrlV \notin evade\_valueEmp \wedge exec = plant$
$\implies safeEpsilonEmp(Vol(t1) \mapsto ctrlV) = TRUE)$
inv2 :
$ctrlV \notin evade\_valueFill \wedge exec = prg \implies safeEpsilonFill(Vol(t) \mapsto ctrlV) = TRUE$
inv3 : $ctrlV \notin evade\_valueEmp \wedge exec = prg \implies$
$safeEpsilonEmp(Vol(t) \mapsto ctrlV) = TRUE$
inv4 : $\forall t1, t2 \cdot t1 \in TIME \wedge t2 \in TIMEdom(Vol) = [0, t1] \wedge dom(Vol) = [0, t2] \implies t1 = t2$

Figure 8.21: Tank_Time INVARIANTS.

The continuous part of the *Tank_ Time* model is described by the event *Plant_ time_ tank* (Figure 8.22). Event *Plant_ time_ tank* refines the event *Plant_ event_ tank* by adding the parameters $lastTime$ and $epsilon1$ (*Rule_ TT_ M_ 3*). The parameter $epsilon1$ represents the maximum duration of the water level until the next discrete state change. This period is equal to $t - lastTime$, see $grd4$. As stated before, to prove the safety property of a hybrid system we must obtain the solutions of its differential equations. The result of solving the ordinary differential equation that describes the continuous evolution of the water level Vol is represented by the $grd5$: $Vol1 = (ctrlV \times epsilon1) + Vol(lastTime)$. The discrete part of this model is represented by three events, (Figures 8.23, 8.24 and 8.25), that replace the safety envelops $safeFill$ and $safeEmp$ respectively by $safeEpsilonFill$ and $safeEpsilonEmp$ to take into account the duration *epsilon* in the evolution of Vol (*Rule_ TT_ M_ 4*).

EVENT Plant_time_tank
REFINES Plant_event_tank
ANY $Vol1$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \wedge dom(Vol) = [0, lastTime]$
grd3 : $t > lastTime \wedge lastTime \in dom(Vol)$
grd4 : $epsilon1 = t - lastTime$
grd5 : $Vol1 = (\lambda t1 \cdot t1 \in TIME \wedge t1 > lastTime \wedge t1 \leq t \mid$
$(ctrlV \times epsilon1) + Vol(lastTime))$
grd6 : $ode(f\_evol\_V(ctrlV), Vol1(t), t) \in DE(\mathbb{R})$
grd7 : $Solvable([0, t] - dom(Vol), ode(f\_evol\_V(ctrlV), Vol1(t), t))$
grd8 :
$solutionOf([0, t] - dom(Vol), ([0, t] - dom(Vol)) \lhd Vol1, ode(f\_evol\_V(ctrlV), Vol1(t), t))$

THEN
act1 : $Vol := Vol \lhd\!\!+ Vol1$
act2 : $exec := ctrl$
END

Figure 8.22: Event Plant_time_tank.

EVENT Ctrl_normal **REFINES** Ctrl_normal
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{f\_in, -f\_out)\}$
grd3 : $nCtrlV = f\_in \implies safeEpsilonFill(Vol(t) \mapsto f\_in) = TRUE$
grd4 : $nCtrlV = -f\_out \implies safeEpsilonEmp(Vol(t) \mapsto -f\_out) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Figure 8.23: Tank_Time Ctrl_normal.

EVENT Ctrl_emptying **REFINES** Ctrl_emptying
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilonEmp(Vol(t) \mapsto -f\_out) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := -f\_out$
END

Figure 8.24: Tank_Time Ctrl_emptying.

EVENT Ctrl_filling **REFINES** Ctrl_filling
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilonFill(Vol(t) \mapsto f\_in) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := f\_in$
END

Figure 8.25: Tank_Time Ctrl_filling.

8.2.4 Correctness of the Specification

Table 8.3 gives the statistics of the POs generated for modeling the *Water Tank* case study following the strategy depicted by Figure 6.1. All the POs are discharged using the RODIN platform and the theories of [10]. It is noticeable that 42% of them were automatically discharged. Most interactive proof obligations are related to the proof of refinement between the three specific models. To prove the refinement relation between the generic model *ContSystem* and *Abstract_Tank_M*, the following proof obliagtion is generated. This PO checks the compliance between the guards of the event *Plant* after replacing the generic variable $plantV$ by the specific one Vol.

$$Vol1 \in [0, t] - dom(plantV) \rightarrow \mathbb{R} \wedge$$

$$AppendSolutionBAP(e, [0, t] - dom(plantV), [0, t] - dom(plantV), Vol1)$$

Machine *Tank_Time_M* refines machine *Tank_Event_M* to introduce the control period *epsilon*, and replace the formulas $safeEmp$ and $safeFill$ respectively by $safeEpsilonEmp$ and $safeEpsilonFill$. Consequently, Rodin has generated the following proof obligations to prove the compliance between these two models. The first two POs are generated in the event *Ctrl_normal* for removing guards $grd3$ and $grd4$ in Figure 8.16. The third PO is

Table 8.2: RODIN proof statistics for the Water Tank system

Specific_Models	**Total**	**Automatic**	**Interactive**
Abstract_Tank_M	11	3	8
Tank_Event_M	38	24	14
Tank_Time_M	42	12	30

generated for the event *Ctrl_emtying* and the last one is generated for the event *Ctrl_filling* for removing the guard $grd2$ of both events.

$$nCtrlV = f\_in \implies safeFill(Vol(t) \mapsto f\_in) = TRUE$$

$$nCtrlV = -f\_out \implies safeEmp(Vol(t) \mapsto -f\_out) = TRUE$$

$$safeEmp(Vol(t) \mapsto -f\_out) = TRUE$$

$$safeFill(Vol(t) \mapsto f\_in) = TRUE$$

As stated in the generic *TimeTriggered* model, on specific case studies, the guards related to the formula *evt_trig* added in an *EventTriggered* model are removed, which give rise to the following refinement proof obligations that we have discharged using inference rules.

$$ctrlV \notin evade\_valueFill \implies$$

$$evt\_trigFill(Vol(t) \mapsto (t1 - t) \mapsto ctrlV) = TRUE$$

$$ctrlV \notin evade\_valueEmp \implies$$

$$evt\_trigEmp(Vol(t) \mapsto (t1 - t) \mapsto ctrlV) = TRUE$$

The system safety property is expressed as an invariant in the *Tank_Event_M*: $\forall x \cdot x \in dom(Vol) \implies Vol(x) \leq V\_high \wedge Vol(x) \geq V\_low$. Similarly to the *Stop Sign* case study, we use the invariant $inv1$ (Figure 8.21) to prove this safety property. This invariant states that before executing the *Plant_time_tank* event, $safeEpsilonFill$ (resp. $safeEpsilonEmp$) is satisfied if we are in the emptying (resp. filling) phase.

8.3 Discussion on the proof activity

From the different case studies that we have modeled and verified to prove the feasibility of our approach, the following conclusions can be drawn:

(1) Most generated POs are generic and do not depend on a specific case study.

(2) The POs that depend on the case study can be classified into four categories:

   (a) The proof of the safety property in the *EventTriggered* model;

   (b) The refinement of the event *Progress*, in the *TimeTriggered* model, preserves the guard *evt_trig*;

   (c) The refinement of the event *Plant_event* in the *TimeTriggered* model preserves the guard $grd6$ related to the safety property;

   (d) The refinement of the events *Ctrl_normal* and *Ctrl_evade*, in the *TimeTriggered* model, preserves the guards related to the predicate $safe$.

(3) 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.

(4) To discharge some proofs, the following generic invariants have been defined:

(a) In the *EventTriggered* model: an invariant to state that if the next event to execute is different from the *Plant* event, every variable x_i describing a physical element, such as the position and the velocity in the *Stop Sign* case study, should be defined until the current value of time t:

$$exec \neq plant \implies \bigwedge_{x_i}(dom(x_i) = [0, t])$$

(b) In the *EventTriggered* model: an invariant to state that if the next event to execute is the *Plant* event, no variable x_i describing a physical element is defined for the current value of time t:

$$exec = plant \implies \bigwedge_{x_i}(t \notin dom(x_i))$$

(c) In the machine corresponding to the *TimeTriggered* model of a specific application (in Car_Time and $Tank\_Time$ for instance): an invariant to state that if the next event to execute is the progress of the time($Progress\_time$), the system is safe if it is in a normal mode:

$$ctrlV \notin evade\_values \wedge exec = prg$$
$$\implies$$
$$safeEpsilon(plantV(t), ctrlV) = TRUE$$

8.4 The Smart Heating System Models

This section illustrates the approach presented in chapter 7 on a frequently used cyber-physical case study, the *Smart Heating* system. For that purpose, we follow the schema depicted by Figure 8.26. The instantiation starts by refining the generic model *TimeTriggeredDesolve* to obtain the machine $Heater\_M$ that sees the context $Heater\_Ctx$. The whole models are available in Appendix D.

8.4.1 Context Heater_Ctx

The context $Heater\_Ctx$ (Figure 8.27) contains the following elements:

- $axm1-2$: define the initial value of the temperature, $T0$, the maximum and the minimum limits of the temperature, T_max and T_min, as well as the main properties of these constants.
- $axm3-5$: valuate the set $PROP$ by defining two formulas $p1$ and $p2$.
- $axm6$: specifies the safety property of the *Smart Heating* system, $T\_min \leq T \leq T\_max$.
- $axm7$: specifies the safety envelops of the system, $T \leq T\_max$ and $T \geq T\_min$, using the formulas $p1$ and $p2$.
- $axm8$: specifies the safety envelops taking into account the control period *epsilon*.

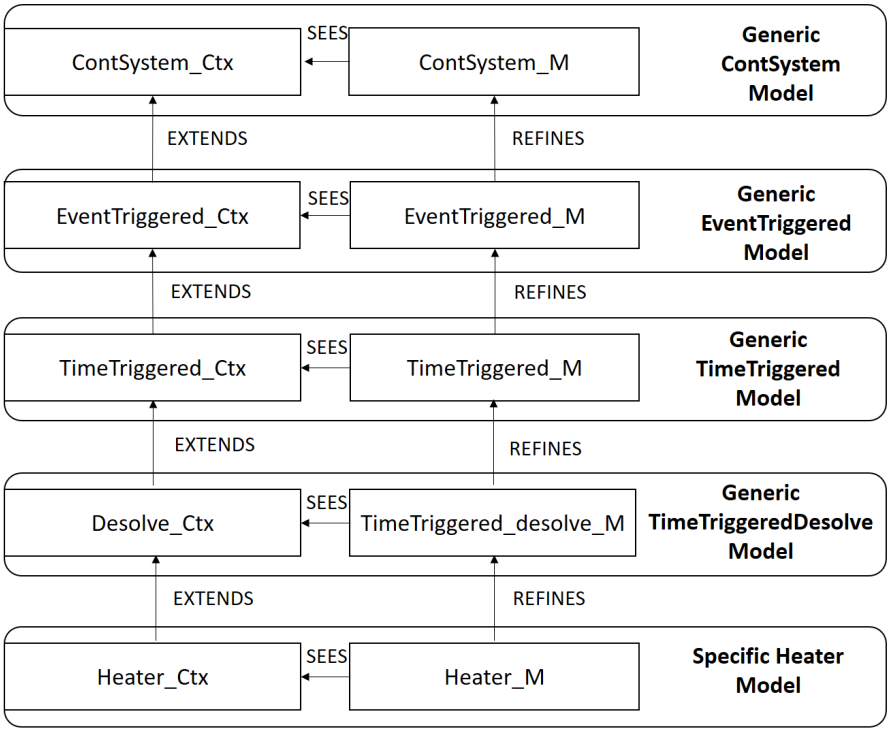

Figure 8.26: Architecture of the EVENT-B Model of the Smart Heating System.

- $axm9$: specifies the event trigger which guarantees that the physical part does not go beyond the boundaries of the safety envelope.
- $axm10 - 11$: specify the evade values of the system using the flow $temp$. The evade value associated with the On state is $-temp$ and that associated with the Off state is $temp$.

8.4.2 Machine Heater_M

The interaction between the discrete and continuous parts of the *Smart Heating* case study are described by the EVENT-B machine $Heater\_M$ which refines the generic machine $TimeTriggeredDesolve\_M$. The *INVARIANTS* clause (Figure 8.28) defines a set of properties that the system should satisfy. Invariant $inv1$ is defined to replace the generic continuous variable $plantV$ by the specific one represented by the temperature level T. Invariant $inv2$ specifies the possible values of the variable $ctrlV$. The most interesting invariants are $inv5$, $inv6$ and $inv7$. $inv5$ specifies the system safety property using the formula $prop$. $inv6$ is defined to guarantee that the temperature T does not exceed the limits T_max and T_min during the evolution of the physical part. $inv7$ specifies the same property as the invariant $inv6$ but this time during the execution of the event $Progress$.

The continuous part is represented by the discrete event $Thermostat\_plant$ (Figure 8.29) that refines the generic event $Plant\_time\_desolve$ by replacing the generic parameter $plant1$ with the exact solution obtained using SAGEMATH. For this purpose, we use the function $B\_desolve$ by valuating each of his parameter by the one associated with the *Smart Heating* case study. $grd3$ assigns the solution returned by the function $B\_desolve$ to the parameter $plant1$. $B\_desolve$ is used to obtain the values of the dependent variable T in the interval $[lastTime, t]$. $grd4$ specifies the exact solution of the ordinary differential equation $T' = ctrlV$. As stated before, the exact solution in SAGEMATH of differential equation for dependent variable is obtained using the predefined function $desolve$. This function is used in the script 7.10 presented in Section 7.2.3. This script returns the solution of $T' = ctrlV$

CONTEXT Heater_Ctx **EXTENDS** Desolve_Ctx
CONSTANTS $p1, p2, prop\_val, T\_max, T\_min, T0, temp$
AXIOMS
axm1: $T0 \in \mathbb{R}^+ \land T0 = plantVInit$
axm2: $T\_max \in \mathbb{R} \land T\_min \in \mathbb{R} \land T\_max > T\_min \land T\_min > 0 \land T0 \geq T\_min \land T0 \leq T\_max$
axm3: $prop\_val \in PROP \to P(\mathbb{R} \times BOOL)$
axm4: $PROP = \{p1, p2\}$
axm5: $prop\_val = \{p1 \mapsto (\lambda\, T \cdot T \in \mathbb{R} \mid bool(T\_min \leq T)), p2 \mapsto (\lambda\, T \cdot T \in \mathbb{R} \mid bool(T \leq T\_max))\}$
axm6: $prop = (\lambda\, T \cdot T \in \mathbb{R} \mid bool((prop\_val(p1))(t) = TRUE \land (prop\_val(p2))(t) = TRUE))$
axm7: $prop\_safe = \{p1 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T < T\_max)), p2 \mapsto (\lambda T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T > T\_min))\}$
axm8: $prop\_safeEpsilon = \{p1 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T + (ctrlV \times epsilon) \leq T\_max)), p2 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in \mathbb{R} \land ctrlV \in \mathbb{R} \mid bool(T + (ctrlV \times epsilon) \geq T\_min))\}$
axm9: $prop\_evt\_trig = \{p1 \mapsto (\lambda\, T \mapsto t1 \mapsto ctrlV \cdot T \in \mathbb{R} \land t1 \in TIME \land ctrlV \in \mathbb{R} \mid bool(T + (ctrlV \times t1) \leq T\_max)), p2 \mapsto (\lambda\, T \mapsto t1 \mapsto ctrlV \cdot T \in \mathbb{R} \land t1 \in TIME \land ctrlV \in \mathbb{R} \mid bool(T + (ctrlV \times t1) \geq T\_min))\}$
axm10: $temp \in \mathbb{R} \land temp > 0$
axm11: $prop\_evade\_values = \{p1 \mapsto \{-temp\}, p2 \mapsto \{temp\}\}$
END

Figure 8.27: CONTEXT Heater_Ctx.

INVARIANTS
inv1: $T = plantV \land ran(T) \subseteq \mathbb{R}$
inv2: $ctrlV \in \{temp, -temp\}$
inv3: $exec \neq plant \implies dom(T) = [0, t]$
inv4: $exec = plant \implies t \notin dom(T)$
inv5: $\forall x \cdot x \in dom(T) \implies prop(T(x)) = TRUE$
inv6: $\exists\, t1 \cdot t1 \in TIME \land dom(T) = [0, t1] \land t - t1 \leq epsilon \land (exec \neq plant \implies t1 = t) \land (exec = plant \implies t > t1) \land (\forall x \cdot x \in PROP \land ctrlV \notin prop\_evade\_values(x) \land exec = plant \implies (prop\_safeEpsilon(x))(T(t1) \mapsto ctrlV) = TRUE)$
inv7: $\forall x \cdot x \in PROP \land ctrlV \notin prop\_evade\_values(x) \land exec = prg \implies (prop\_safeEpsilon(x))(T(t) \mapsto ctrlV) = TRUE$
inv8: $\forall t1, t2 \cdot t1 \in TIME \land t2 \in TIME \land dom(T) = [0, t1] \land dom(T) = [0, t2] \implies t1 = t2$

Figure 8.28: Heater_M INVARIANTS.

in the language of SageMath. This solution is then translated in the Event-B language to be used in the *proof* phase. The discrete part is represented by the event *Ctrl* (Figure 8.30) that refines the generic event *Ctrl* by replacing the generic continuous variable *plantV* by T in *grd*3. Moreover, the event uses the formula $prop\_safeEpsilon$ to check that the chosen *value* satisfies the safety envelop during the control period *epsilon*.

8.4.3 Correctness of the specification

To ensure the correctness of the developed models, a set of proof obligations are produced. These proof obligations aim at verifying that the different refinements are correct and the safety properties are verified on the system. It is noticeable that 54% of them were automatically discharged. These POs include the correctness of the events that specify the progression of time and those that specify the progression of the physical and the discrete parts and also the POs that verify the type of the variables. The POs related to the guards

EVENT Thermostat_plant **REFINES** Plant_time_desolve
ANY $lastTime$, $plant1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \land dom(T) = [0, lastTime]$
grd3 : $plant1 = B\_desolve(1 \mapsto ctrlV \mapsto T \mapsto t \mapsto (lastTime \mapsto T(lastTime)))$
grd4 : $B\_desolve(1 \mapsto ctrlV \mapsto T \mapsto t \mapsto (lastTime \mapsto T(lastTime))) =$
$(\lambda\, t1 \cdot t1 \in TIME \land t1 > lastTime \land t1 \leq t \mid T(lastTime) +$
$(ctrlV \times -lastTime) + (ctrlV \times t1))$
grd5 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(RReal)$
grd6 : $Solvable([0, t] - dom(T), ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd7 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV),$
$plant1(t), t), [0, t] - dom(T), [0, t] - dom(T), plant1)$
THEN
act1 : $T := T \lhd\!\!+ plant1$
act2 : $exec := ctrl$
END

Figure 8.29: Event Thermostat_plant.

EVENT Ctrl
REFINES Ctrl
ANY $value$
WHERE
grd1 : $exec = ctrl$
grd2 : $value \in \{temp, -temp\}$
grd3 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies$
$prop\_safeEpsilon(x))(T(t) \mapsto value) = TRUE)$
THEN
act1 : $ctrlV := value$
act2 : $exec := prg$
END

Figure 8.30: Thermostat Ctrl.

feasibility and well-definedness have been interactively discharged under Rodin. Among these proof obligations, we can cite those related to the elimination of the guards during the refinement of events. These proof obligations are specified using the set $PROP$. For example, Rodin has generated the following proof obligation due to removing the guard related to the formula evt_trig in the event $Progress$:

$$\forall\, x \cdot x \in PROP \implies (ctrlV \notin prop\_evade\_values(x) \implies$$
$$prop\_evt\_trig(x)(plantV(t) \mapsto (t1 - t) \mapsto ctrlV) = TRUE)$$

To discharge such proof obligations, we needed to add invariants that translate implicit properties on the system. These invariants specify that the system is safe if the controller has chosen a value for $ctrlV$ that does not belong to the sets of evade values, see guard

Table 8.3: Rodin Proof Statistics for the Smart Heating System

Specific_Models	**Total**	**Automatic**	**Interactive**
Heater_M	48	21	27

$grd6$ and $grd7$ in Figure 8.28.

$$plant1 = B\_desolve(1 \mapsto ctrlV \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)))$$

The above proof obligation was generated in the event *Thermostat_plant* to prove the compliance between the generic model *TimeTriggeredDesolve* and the specific one. This PO was discharged by replacing *B_desolve* by the results of solving the associated ordinary differential equation using the interface between SageMath and Rodin.

8.5 Modeling NonLinear Case Studies

As stated before, in case of higher order nonlinear ODEs, we use approximation techniques to transform an equation into an equivalent equation of another type. We can use linearization techniques to transform a nonlinear differential equation into a linear differential equation and then apply our generic approach that model the predefined function *desolve* in Event-B. For example, the *Inverted Pendulum* is a non linear hybrid system whose continuous behavior is described by a second order ordinary differential equation:

$$f_{NonLin}((\theta, \dot{\theta}), u) = (\dot{\theta}, u * cos(\theta) + \frac{g}{l})$$

Therefore, the functions used by SageMath to obtain approximate solutions, such as *desolve_rk4*, can not be applied since they only treat first order differential equations. In that case, we must use linearisation methods to linearise the differential equation of this case study and then apply our generic approach that models the function *desolve* in Event-B. Then we use the approach presented in [59] which allows proving the refinement relation between a linear and a nonlinear systems. To linearise such a case study, we assume that the angle θ is small enough, so we can approximate $\sin(\theta)$ and $\cos(\theta)$ such that $|\sin(\theta) - \theta| = 0 \Rightarrow sin(\theta) = \theta$ and $|\cos(\theta) - \theta| = 1 \Rightarrow cos(\theta) = 1$. Assuming this condition holds, it is possible to approximate f_{NonLin} to a simpler form, so-called linearised: $v = \ddot{\theta} - \frac{g}{l}\theta$, with v is an adequate linear control command linked to the non linear controlled variable u after linearisation. u is replaced by v so that the values of the continuous variables obtained by solving the linear system are an approximation of those of the non-linear system. Therefore, f_{NonLin} is replaced by:

$$f_{Lin}((\theta, \dot{\theta}), v) = (\dot{\theta}, v + \frac{g}{l}\theta)$$

The continuous measurements of this system are represented by the angle θ and the angular velocity $\dot{\theta}$ represented in our approach by two partial functions defined as follows: $\theta \in [0, t] \nrightarrow \mathbb{R}$ and $\dot{\theta} \in [0, t] \nrightarrow \mathbb{R}^+$. The discrete behavior is represented by the discrete variable v defined in the linearised system to replace the non linear controlled variable u. To model this system in Event-B, we must define a specific model that refines the *TimeTriggeredDesolve* model. The associated context specifies the definition of the main constants of the system such as the length l of the rigid rod and the intensity g. It also specifies the linearised differential equation using the function f_evol_plantV. The discrete part of this model is represented by a discrete event defined in the associated machine and which updates the value of v to stabilise the rod in its vertical position.

In the case of nonlinear system whose continuous part is described by a first order nonlinear, we directly refine the model *TimeTriggeredDesolverk4*. The instantiation on a specific case study is similar to that described for instantiating the *TimeTriggeredDesolve* model on the *Smart Heating* system. Let us remark that for the nonlinear 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:

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

Having this property as verified, the proof of a safety property comes down to prove it for the lower and/or the upper bounds. So far we have failed to find a nonlinear system whose continuous behavior is described by a first order nonlinear differential equation. Therefore, as future work we plane to introduce more complexity on the modeled case studies to use the function *B_ desolve_ rk4* in order to get approximate solutions.

8.6 Conclusion

In this chapter, we have experimented our generic approach on three representative case studies, *Stop Sign*, *Water Tank* and *Smart Heating* systems. The *Stop Sign* case study is described by a single safety property which involves a single safety envelop, a single event trigger formula and a single evade mode. To model this case study, we chose to directly refine the generic *EventTriggered* model which already provides the definition of a single safety envelope, an event trigger formula and a set of evade values. While in the case of the *Water Tank* case study, the safety property is defined as a conjunction of two sub-formulas which requires the definition of two safety envelopes, two event trigger formulas and two set of evade values. For this purpose, we chose to introduce an abstract model of the *Water Tank* system by refining the generic *ContSystem* model. Therefore, each case study illustrating one strategy. The first strategy consists in refining directly the generic *EventTriggered* model. The second one consists in starting by instantiating by refinement the generic *ContSystem* model. All the proof obligations of the two specific models have been discharged using the Rodin platform and the theories of [10]. It is worth noting that the models and the instantiation rules being generic, a tool can be developed to automate the instantiation.

The approach that introduces the function *B_ desolve* is applied on another linear case study, the *Smart Heating* system. Using the interface between SageMath and Rodin, we obtain exact solutions of ordinary differential equations which facilitates the proof of the safety properties of hybrid systems. We admit that the chosen case study is a simple hybrid system with a linear ODE but it served us well to describe the different steps for applying our generic approach. Using SageMath, we can deal with more complex ODEs by modeling the function *desolve_ rk4* which solves nonlinear ODEs in the case of first order ODEs or using the linearisation methods which is the case of the *Inverted Pendulum*. The strength of our approach comparing to other proof-based approaches is that it proves the safety properties of hybrid systems in the more concrete model of CPSs, *TimeTriggered* model, using Event-B and its supported tool Rodin. Without solving ODEs, our models were abstract and did not allow proving the safety properties of hybrid systems.

Chapter 9

Conclusion

Cyber-physical systems allow interactions with the physical world using a network of sensors and actuators. Cyber-physical systems are often represented by their common mathematical model of hybrid systems which combine continuous dynamics represented by differential equations with discrete dynamics. The interaction between the software part and the physical world makes the verification of cyber-physical systems an intellectual challenge. To address this challenge, several formal methods have been proposed which can be grouped into two categories: *Model-checking based approaches* and *Proof-based approaches*. *Model-checking based approaches* use hybrid automata and algorithmic analysis methods to model and verify CPSs. An example of model-checker for CPSs is *Hytech* which is the first model checker that implements the reachability analysis for hybrid systems. The main limitation of these approaches is that the reachability is not decidable for non linear hybrid systems. *Proof-based approaches* use formal methods and deductive verification to model and verify CPSs. Their key feature is that they support the description of any category of hybrid systems.

We distinguish two main types of cyber-physical systems, *Event-Triggered* systems where sensors have a continuous access to the measurement of continuous behaviors, and *Time-Triggered* systems where sensors take periodic measurements of continuous behaviors. In an *Event-Triggered* model, the system evolves continuously until a particular event triggers the controller, while in a *Time-Triggered* model, the controller is periodically triggered to control the system. An *Event-Triggered* model is easy to verify and difficult (if not impossible) to implement whereas *Time-Triggered* model is difficult to verify and easy to implement.

Modeling the physical part of cyber-physical systems requires the manipulation of differential equations which are equations that involve a set of functions, as unknown variables, and their derivatives. In this thesis, we are interesting in modelling systems where the continuous behavior is described by ordinary differential equations (ODEs). An ODE is the relationship between a single independent variable x, an unknown function y, and its derivative at a point x. Methods to resolve ordinary differential equations fall into two categories : analytical (symbolic) methods and numerical methods. However, it is not always possible to obtain exact solutions of ordinary differential equations, thus the use of numerical methods or approximation techniques to obtain approximate solutions.

9.1 Contribution

In this thesis, we presented a correct-by-construction proof-based approach for modeling and verifying hybrid systems using the Event-B formal method and its refinement technique. The proposed approach is based on modeling and verifying the relationship between the *Event* and *Time-Triggered* systems using Event-B. It defines two generic models for these systems as described in dR$\mathcal{L}$ [5]. The generic *Event-Triggered* model describes the

interaction between the physical and the discrete parts using a discrete variable *exec* that can take as value, *prg* which specifies the progression of time, *ctrl* which represents the discrete part and *plant* which represents the continuous part. The generic *Time-Triggered* model introduces the notion of control period represented by the constant *epsilon*, so the controller reacts at least every *epsilon* time. We have also introduced a more abstract level, the generic *ContSystem* model, that specifies the continuous aspects of hybrid systems, adapted from the approach introduced by Dupond et al [10]. This permits to cope with the proof complexity by decomposing the proof obligations, such that in the abstract model we only deal with the proof obligations related to the continuous aspects of the system and in the refined model we will have the proof obligations related to the safety properties of the controlled system. To instantiate these generic models, we have introduced two strategies: the first one consists in starting by instantiating by refinement the *Generic ContSystem* model. The second one consists in refining directly the *Generic EventTriggered* model. We have applied our approach to two case studies, each one illustrating one strategy. All the proof obligations of the generic models and the two specific models have been discharged using the Rodin platform and the theories introduced in [10]. It is worth noting that as the models and the instantiation rules are generic, a tool can be developed to automate the instantiation.

The generic approach is still at an abstract level regarding the solutions of differential equations. For this purpose, we extend the generic approach by making use of the SageMath solver. We propose a tool-supported approach which combines the Event-B formal method with the differential equation solver SageMath. This is achieved by implementing a plug-in to Rodin that permits to call SageMath. The interface between Event-B with the differential equation solver SageMath is done by modeling and implementing the call of two predefined functions regarding the type of ODE: *B_ desolve* to obtain exact solutions and *B_ desolve_ rk4* to obtain approximate solutions.

9.2 Future Work

To demonstrate the usability of our approach, we have tested it on three representative case studies, *Stop Sign*, *Water Tank* and *Smart Heating*. In the three case studies, the differential equations that represent the evolution of their physical parts are linear and can be easily solved. To handle more elaborated differential equations we plan to model other cyber-physical case studies.

Our approach is still at an abstract level. It does not take into account the delay between the sending of continuous measurements by the sensors and their processing by the controller as well as the delay between the sending of actions by the controller and their execution. For this purpose, we plan to define more specific models using the Event-B refinement to consider these delays. By doing that, we are getting closer to our goal of reducing the gap between a verified model of a cyber-Physical system and its implementation.

We admit that the chosen case studies are simple hybrid systems with linear ODEs but they served us well 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. The strength of our approach comparing to other proof-based approaches is that it proves that the more concrete model of CPSs, *TimeTriggered* model, preserves the safety properties of hybrid systems using Event-B and its supported tool Rodin.

The work described in this thesis presents a first step that will facilitate the treatment of complex hybrid systems using Event-B. For this purpose, we plan to apply our approach on more complex case studies especially on nonlinear case studies that admit approximate solutions to prove the feasibility of our generic approach that models the function *B_ desolve_ rk4*.

We defined a set of theorems, generic axioms, and generic invariants identified from case studies to prove the safety properties of cyber-physical systems. In order to easily use these

theorems, axioms and invariants during the proof stage, we plan to develop an Event-B theory using the theory plugin.

For a discrete system Event-B, we can use the animator ProB [19], a B-method constraint solver and model checker, to ensure that the modeled system behaves as expected. As a future work, we propose to extend the reasoning of ProB to develop an animator that simulates the behavior of CPSs by using the solutions of ordinary differential equations obtained with the *SageMath* plugin.

The *SageMath* plugin is still in early development. There are some manual and interactive steps. For example, scripts must run manually in SageMath. In future work, we plan to automate these steps into a 100% automated tool.

Bibliography

[1] M. Afendi, R. Laleau, A. Mammar, Modelling hybrid programs with event-b, in: A. Raschke, D. Méry, F. Houdek (Eds.), Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings, Vol. 12071 of Lecture Notes in Computer Science, Springer, 2020, pp. 139–154. doi:10.1007/978-3-030-48077-6_10.
URL https://doi.org/10.1007/978-3-030-48077-6_10

[2] E. A. Lee, Cyber physical systems: Design challenges, in: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA, IEEE Computer Society, 2008, pp. 363–369. doi:10.1109/ISORC.2008.25.
URL https://doi.org/10.1109/ISORC.2008.25

[3] H. Kopetz, Event-triggered versus time-triggered real-time systems, in: A. I. Karshmer, J. Nehmer (Eds.), Operating Systems of the 90s and Beyond, International Workshop, Dagstuhl Castle, Germany, July 8-12, 1991, Proceedings, Vol. 563 of Lecture Notes in Computer Science, Springer, 1991, pp. 87–101. doi:10.1007/BFb0024530.
URL https://doi.org/10.1007/BFb0024530

[4] A. Platzer, A complete uniform substitution calculus for differential dynamic logic, J. Autom. Reason. 59 (2) (2017) 219–265. doi:10.1007/s10817-016-9385-1.
URL https://doi.org/10.1007/s10817-016-9385-1

[5] S. M. Loos, A. Platzer, Differential refinement logic, in: M. Grohe, E. Koskinen, N. Shankar (Eds.), Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016, ACM, 2016, pp. 505–514. doi:10.1145/2933575.2934555.
URL https://doi.org/10.1145/2933575.2934555

[6] P. Zimmermann, A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, et al., Computational mathematics with SageMath, SIAM, 2018.

[7] R. G. Sanfelice, D. A. Copp, P. Nanez, A toolbox for simulation of hybrid systems in matlab/simulink: hybrid equations (hyeq) toolbox, in: C. Belta, F. Ivancic (Eds.), Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA, ACM, 2013, pp. 101–106. doi:10.1145/2461328.2461346.
URL https://doi.org/10.1145/2461328.2461346

[8] P. Zuliani, A. Platzer, E. M. Clarke, Bayesian statistical model checking with application to simulink/stateflow verification, in: K. H. Johansson, W. Yi (Eds.), Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010, ACM, 2010, pp. 243–252. doi:10.1145/1755952.1755987.
URL https://doi.org/10.1145/1755952.1755987

[9] EBRP ANR Project, https://www.irit.fr/EBRP/.

[10] G. Dupont, Y. A. Ameur, M. Pantel, N. K. Singh, Proof-based approach to hybrid systems development: Dynamic logic and event-b, in: M. J. Butler, A. Raschke, T. S. Hoang, K. Reichl (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings, Vol. 10817 of Lecture Notes in Computer Science, Springer, 2018, pp. 155–170. doi:10.1007/978-3-319-91271-4_11.
URL https://doi.org/10.1007/978-3-319-91271-4_11

[11] S. Engell, R. Paulen, M. A. Reniers, C. Sonntag, H. Thompson, Core research and innovation areas in cyber-physical systems of systems - initial findings of the cpsos project, in: C. Berger, M. R. Mousavi (Eds.), Cyber Physical Systems. Design, Modeling, and Evaluation - 5th International Workshop, CyPhy 2015, Amsterdam, The Netherlands, October 8, 2015, Proceedings, Vol. 9361 of Lecture Notes in Computer Science, Springer, 2015, pp. 40–55. doi:10.1007/978-3-319-25141-7_4.
URL https://doi.org/10.1007/978-3-319-25141-7_4

[12] F. Mallet, E. Villar, F. Herrera, Marte for cps and cpsos, in: Cyber-Physical System Design from an Architecture Analysis Viewpoint, Springer, 2017, pp. 81–108.

[13] J. Lunze, F. Lamnabhi-Lagarrigue, Handbook of hybrid systems control: theory, tools, applications, Cambridge University Press, 2009.

[14] L. P. Carloni, R. Passerone, A. Pinto, A. L. Sangiovanni-Vincentelli, Languages and tools for hybrid systems design, Found. Trends Electron. Des. Autom. 1 (1/2) (2006). doi:10.1561/1000000001.
URL https://doi.org/10.1561/1000000001

[15] E. L. Ince, Ordinary differential equations, Courier Corporation, 1956.

[16] J. Quesel, S. Mitsch, S. M. Loos, N. Aréchiga, A. Platzer, How to model and prove hybrid systems with keymaera: a tutorial on safety, Int. J. Softw. Tools Technol. Transf. 18 (1) (2016) 67–91. doi:10.1007/s10009-015-0367-0.
URL https://doi.org/10.1007/s10009-015-0367-0

[17] J.-R. Abrial, The B-book: assigning programs to meanings, Vol. 1, Cambridge university press Cambridge, 1996.

[18] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[19] J. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in event-b, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466. doi:10.1007/s10009-010-0145-y.
URL https://doi.org/10.1007/s10009-010-0145-y

[20] I. Maamria, M. J. Butler, Rewriting and well-definedness within a proof system, in: E. Komendantskaya, A. Bove, M. Niqui (Eds.), Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010, Vol. 5 of EPiC Series, EasyChair, 2010, pp. 55–71. doi:10.29007/b7wc.
URL https://doi.org/10.29007/b7wc

[21] M. Noro, T. Takeshima, Risa/asir - a computer algebra system, in: P. S. Wang (Ed.), Proceedings of the 1992 International Symposium on Symbolic and Algebraic Computation, ISSAC '92, Berkeley, CA, USA, July 27-29, 1992, ACM, 1992, pp. 387–396. doi:10.1145/143242.143362.
URL https://doi.org/10.1145/143242.143362

[22] S. Wolfram, The Mathematica book, 5th edn. wolfram media, champaign (2003).

[23] I. Tech, Introduction to maple (1993).

[24] C. Bischof, B. Lang, A. Vehreschild, Automatic differentiation for matlab programs, in: PAMM: Proceedings in Applied Mathematics and Mechanics, Vol. 2, Wiley Online Library, 2003, pp. 50–53.

[25] W. Bosma, J. J. Cannon, C. Playoust, The magma algebra system I: the user language, J. Symb. Comput. 24 (3/4) (1997) 235–265.

[26] A. Heck, A. Heck, Introduction to MAPLE, Vol. 1993, Springer, 1993.

[27] T. A. Henzinger, P. Ho, H. Wong-Toi, HYTECH: A model checker for hybrid systems, Int. J. Softw. Tools Technol. Transf. 1 (1-2) (1997) 110–122. doi:10.1007/s100090050008.
URL https://doi.org/10.1007/s100090050008

[28] G. Frehse, Phaver: Algorithmic verification of hybrid systems past hytech, in: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings, Vol. 3414 of Lecture Notes in Computer Science, Springer, 2005, pp. 258–273. doi:10.1007/978-3-540-31954-2_17.
URL https://doi.org/10.1007/978-3-540-31954-2_17

[29] E. Asarin, T. Dang, O. Maler, The d/dt tool for verification of hybrid systems, in: E. Brinksma, K. G. Larsen (Eds.), Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings, Vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 365–370. doi:10.1007/3-540-45657-0_30.
URL https://doi.org/10.1007/3-540-45657-0_30

[30] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, O. Maler, Spaceex: Scalable verification of hybrid systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 379–395. doi:10.1007/978-3-642-22110-1_30.
URL https://doi.org/10.1007/978-3-642-22110-1_30

[31] X. Chen, E. Ábrahám, S. Sankaranarayanan, Flow*: An analyzer for non-linear hybrid systems, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Vol. 8044 of Lecture Notes in Computer Science, Springer, 2013, pp. 258–263. doi:10.1007/978-3-642-39799-8_18.
URL https://doi.org/10.1007/978-3-642-39799-8_18

[32] M. Fränzle, C. Herde, T. Teige, S. Ratschan, T. Schubert, Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure, J. Satisf. Boolean Model. Comput. 1 (3-4) (2007) 209–236. doi:10.3233/sat190012.
URL https://doi.org/10.3233/sat190012

[33] S. Kong, S. Gao, W. Chen, E. M. Clarke, dreach: δ-reachability analysis for hybrid systems, in: C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Vol. 9035 of Lecture Notes in Computer Science, Springer, 2015, pp. 200–205. doi:10.1007/978-3-662-46681-0_15.
URL https://doi.org/10.1007/978-3-662-46681-0_15

[34] H. Wong-Toi, The synthesis of controllers for linear hybrid automata, in: Proceedings of the 36th IEEE Conference on Decision and Control, Vol. 5, IEEE, 1997, pp. 4607–4612.

[35] X. Chen, E. Ábrahám, S. Sankaranarayanan, Taylor model flowpipe construction for non-linear hybrid systems, in: Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012, IEEE Computer Society, 2012, pp. 183–192. doi:10.1109/RTSS.2012.70.
URL https://doi.org/10.1109/RTSS.2012.70

[36] A. Platzer, J. Quesel, Keymaera: A hybrid theorem prover for hybrid systems (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings, Vol. 5195 of Lecture Notes in Computer Science, Springer, 2008, pp. 171–178. doi:10.1007/978-3-540-71070-7_15.
URL https://doi.org/10.1007/978-3-540-71070-7_15

[37] N. Fulton, S. Mitsch, J. Quesel, M. Völp, A. Platzer, Keymaera X: an axiomatic tactical theorem prover for hybrid systems, in: A. P. Felty, A. Middeldorp (Eds.), Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, Vol. 9195 of Lecture Notes in Computer Science, Springer, 2015, pp. 527–538. doi:10.1007/978-3-319-21401-6_36.
URL https://doi.org/10.1007/978-3-319-21401-6_36

[38] A. Platzer, Differential dynamic logic for hybrid systems, J. Autom. Reason. 41 (2) (2008) 143–189. doi:10.1007/s10817-008-9103-8.
URL https://doi.org/10.1007/s10817-008-9103-8

[39] D. Kozen, On hoare logic and kleene algebra with tests, ACM Trans. Comput. Log. 1 (1) (2000) 60–76.

[40] S. Lunel, S. Mitsch, B. Boyer, J. Talpin, Parallel composition and modular verification of computer controlled systems in differential dynamic logic, CoRR abs/1907.02881 (2019). arXiv:1907.02881.
URL http://arxiv.org/abs/1907.02881

[41] C. Zhou, J. Wang, A. P. Ravn, A formal description of hybrid systems, in: R. Alur, T. A. Henzinger, E. D. Sontag (Eds.), Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA, Vol. 1066 of Lecture Notes in Computer Science, Springer, 1995, pp. 511–530. doi:10.1007/BFb0020972.
URL https://doi.org/10.1007/BFb0020972

[42] H. Jifeng, From CSP to hybrid systems, in: A classical mind: essays in honour of CAR Hoare, 1994, pp. 171–189.

[43] C. A. R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677. doi:10.1145/359576.359585.
URL https://doi.org/10.1145/359576.359585

[44] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, L. Zou, A calculus for hybrid CSP, in: K. Ueda (Ed.), Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings, Vol. 6461 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–15. doi:10.1007/978-3-642-17164-2_1.
URL https://doi.org/10.1007/978-3-642-17164-2_1

[45] L. Zou, N. Zhan, S. Wang, M. Fränzle, S. Qin, Verifying simulink diagrams via a hybrid hoare logic prover, in: R. Ernst, O. Sokolsky (Eds.), Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013, IEEE, 2013, pp. 9:1–9:10. doi:10.1109/EMSOFT.2013.6658587.
URL https://doi.org/10.1109/EMSOFT.2013.6658587

[46] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, R. R. H. Schiffelers, Syntax and consistent equation semantics of hybrid chi, J. Log. Algebraic Methods Program. 68 (1-2) (2006) 129–210. doi:10.1016/j.jlap.2005.10.005.
URL https://doi.org/10.1016/j.jlap.2005.10.005

[47] S. Foster, J. J. H. y Munive, G. Struth, Differential hoare logics and refinement calculi for hybrid systems with isabelle/hol, in: U. Fahrenberg, P. Jipsen, M. Winter (Eds.), Relational and Algebraic Methods in Computer Science - 18th International Conference, RAMiCS 2020, Palaiseau, France, April 8-11, 2020, Proceedings [postponed], Vol. 12062 of Lecture Notes in Computer Science, Springer, 2020, pp. 169–186. doi:10.1007/978-3-030-43520-2_11.
URL https://doi.org/10.1007/978-3-030-43520-2_11

[48] A. Armstrong, V. B. F. Gomes, G. Struth, Building program construction and verification tools from algebraic principles, Formal Aspects Comput. 28 (2) (2016) 265–293. doi:10.1007/s00165-015-0343-1.
URL https://doi.org/10.1007/s00165-015-0343-1

[49] C. Morgan, Programming from Specifications, Prentice-Hall, Inc., 1990.

[50] J. J. H. y Munive, Verification components for hybrid systems, Arch. Formal Proofs 2019 (2019).
URL https://www.isa-afp.org/entries/Hybrid_Systems_VCs.html

[51] S. Boldo, F. Clément, J. Filliâtre, M. Mayero, G. Melquiond, P. Weis, Trusting computations: A mechanized proof from partial differential equations to actual program, Comput. Math. Appl. 68 (3) (2014) 325–352. doi:10.1016/j.camwa.2014.06.004.
URL https://doi.org/10.1016/j.camwa.2014.06.004

[52] S. Boldo, C. Lelay, G. Melquiond, Coquelicot: A user-friendly library of real analysis for coq, Math. Comput. Sci. 9 (1) (2015) 41–62. doi:10.1007/s11786-014-0181-1.
URL https://doi.org/10.1007/s11786-014-0181-1

[53] C. Marché, Jessie: an intermediate language for java and C verification, in: A. Stump, H. Xi (Eds.), Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007, ACM, 2007, pp. 1–2. doi:10.1145/1292597.1292598.
URL https://doi.org/10.1145/1292597.1292598

[54] J. Filliâtre, C. Marché, The why/krakatoa/caduceus platform for deductive program verification, in: W. Damm, H. Hermanns (Eds.), Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, Vol. 4590 of Lecture Notes in Computer Science, Springer, 2007, pp. 173–177. doi:10.1007/978-3-540-73368-3_21.
URL https://doi.org/10.1007/978-3-540-73368-3_21

[55] B. Bohrer, V. Rahli, I. Vukotic, M. Völp, A. Platzer, Formally verified differential dynamic logic, in: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, 2017, pp. 208–221.

[56] G. Babin, A formal approach for correct-by-construction system substitution, CoRR abs/1404.7513 (2014). arXiv:1404.7513.
URL http://arxiv.org/abs/1404.7513

[57] W. Su, J. Abrial, H. Zhu, Formalizing hybrid systems with event-b and the rodin platform, Sci. Comput. Program. 94 (2014) 164–202. doi:10.1016/j.scico.2014.04.015.
URL https://doi.org/10.1016/j.scico.2014.04.015

[58] M. J. Butler, J. Abrial, R. Banach, Modelling and refining hybrid systems in event-b and rodin, in: L. Petre, E. Sekerinski (Eds.), From Action Systems to Distributed Systems - The Refinement Approach, Chapman and Hall/CRC, 2016, pp. 29–42. doi:10.1201/b20053-5.
URL https://doi.org/10.1201/b20053-5

[59] G. Dupont, Y. A. Ameur, M. Pantel, N. K. Singh, An event-b based generic framework for hybrid systems formal modelling, in: B. Dongol, E. Troubitsyna (Eds.), Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings, Vol. 12546 of Lecture Notes in Computer Science, Springer, 2020, pp. 82–102. doi:10.1007/978-3-030-63461-2_5.
URL https://doi.org/10.1007/978-3-030-63461-2_5

[60] Z. Cheng, D. Méry, A refinement strategy for hybrid system design with safety constraints, in: J. C. Attiogbé, S. B. Yahia (Eds.), Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings, Vol. 12732 of Lecture Notes in Computer Science, Springer, 2021, pp. 3–17. doi:10.1007/978-3-030-78428-7_1.
URL https://doi.org/10.1007/978-3-030-78428-7_1

[61] R. Banach, M. J. Butler, S. Qin, N. Verma, H. Zhu, Core hybrid event-b I: single hybrid event-b machines, Sci. Comput. Program. 105 (2015) 92–123. doi:10.1016/j.scico.2015.02.003.
URL https://doi.org/10.1016/j.scico.2015.02.003

[62] A. Buga, A. Mashkoor, S. T. Nemes, K. Schewe, P. Songprasop, An event-b-based approach to hybrid systems engineering and its application to a hemodialysis machine case study, Comput. Lang. Syst. Struct. 54 (2018) 297–315. doi:10.1016/j.cl.2018.07.004.
URL https://doi.org/10.1016/j.cl.2018.07.004

[63] W. Windsteiger, Theorema 2.0: A system for mathematical theory exploration, in: ICMS, Vol. 8592 of Lecture Notes in Computer Science, Springer, 2014, pp. 49–52.

[64] C. Kaliszyk, F. Wiedijk, Certified computer algebra on top of an interactive theorem prover, in: Calculemus/MKM, Vol. 4573 of Lecture Notes in Computer Science, Springer, 2007, pp. 94–105.

[65] R. Y. Lewis, An extensible ad hoc interface between lean and mathematica, in: C. Dubois, B. W. Paleo (Eds.), Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017, Vol. 262 of EPTCS, 2017, pp. 23–37. doi:10.4204/EPTCS.262.4.
URL https://doi.org/10.4204/EPTCS.262.4

[66] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic, Vol. 2283, Springer Science & Business Media, 2002.

[67] C. Ballarin, K. Homann, J. Calmet, Theorems and algorithms: An interface between isabelle and maple, in: A. H. M. Levelt (Ed.), Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, ISSAC '95, Montreal, Canada, July 10-12, 1995, ACM, 1995, pp. 150–157. doi:10.1145/220346.220366.
URL https://doi.org/10.1145/220346.220366

[68] F. Immler, Verified reachability analysis of continuous systems, in: C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Vol. 9035 of Lecture Notes in Computer Science, Springer, 2015, pp. 37–51. doi:10.1007/978-3-662-46681-0_3.
URL https://doi.org/10.1007/978-3-662-46681-0_3

Appendices

Appendix A

Generic Models

A.1 Context ContSystem_Ctx

CONTEXT ContSystem_Ctx
CONSTANTS S, $TIME$, $sigma$, $plantVInit$
AXIOMS
axm1: $S = RReal^n$
axm2: $TIME = RRealPlus$
axm3: $sigma \in RRealPlus \land sigma \mapsto Rzero \in gt$
axm4: $plantVInit \in S$
END

A.2 Machine ContSystem_M

MACHINE ContSystem_M
SEES ContSystem_Ctx
VARIABLES t, $plantV$
INVARIANTS
inv1: $t \in TIME$
inv2: $plantV \in Closed2Closed(Rzero, t) \nrightarrow S$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
END

Progress $\widehat{=}$
THEN
act1: $t :\mid t' \in TIME \land (t \mapsto t' \in lt \land minus(t' \mapsto t) \mapsto sigma \in geq)$
END

Plant $\widehat{=}$

ANY e, $plant1$
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV), e)$
grd3: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \to S \wedge$
$AppendSolutionBAP(e, Closed2Closed(Rzero, t) \backslash dom(plantV),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1: $plantV := plantV \Leftarrow plant1$
END

END

A.3 Context EventTriggered_Ctx

CONTEXT EventTriggered_Ctx
EXTENDS ContSystem_Ctx
SETS $EXEC$
CONSTANTS $safe$, evt_trig, $ctrl$, $plant$, prg, f_evol, f_evol_plantV, $evade\_value$
AXIOMS
axm1: $safe \in (S \times RReal) \to BOOL$
axm2: $evt\_trig \in S \times TIME \times RReal \to BOOL$
axm3: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm4: $f\_evol \in RReal \to S$
axm5: $f\_evol\_plantV \in (RReal \to (TIME \times S \to S))$
axm6: $\forall ctrlV \cdot ctrlV \in RReal \implies (f\_evol\_plantV(ctrlV) =$
$(\lambda t \mapsto plantV \cdot t \in TIME \wedge plantV \in S | f\_evol(ctrlV)))$
axm7: $evade\_value \subseteq RReal \wedge evade\_value \neq \emptyset$
END

A.4 Machine EventTriggered_M

MACHINE EventTriggered_M
REFINES ContSystem_M
SEES EventTriggered_Ctx
VARIABLES t, $plantV$, $ctrlV$, $exec$
INVARIANTS
inv1: $ctrlV \in RReal$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \implies dom(plantV) = Closed2Closed(Rzero, t)$
inv4: $exec = plant \implies t \notin dom(plantV)$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$

act2 : $plantV := \{Rzero \mapsto plantVInit\}$
act3 : $ctrlV :\in RReal$
act4 : $exec := ctrl$
END

Progress $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1 : $exec = prg$
grd2 : $t1 \in TIME \wedge (t \mapsto t1 \in lt \wedge minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3 : $ctrlV \notin evade\_value \implies evt\_trig(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant $\widehat{=}$
REFINES Plant
ANY $plant1$
WHERE
grd1 : $exec = plant$
grd2 : $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
grd3 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd4 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV),$
$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$Closed2Closed(Rzero, t) \backslash dom(plantV),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), plant1(t), t)$
THEN
act1 : $plantV := plantV \lhd plant1$
act2 : $exec := ctrl$
END

Ctrl_normal $\widehat{=}$
ANY $nrml\_value$
WHERE
grd1 : $exec = ctrl$
grd2 : $nrml\_value \in RReal$
grd3 : $nrml\_value \notin evade\_value \implies safe(plantV(t) \mapsto nrml\_value) = TRUE$
THEN
act1 : $ctrlV := nrml\_value$
act2 : $exec := prg$
END

Ctrl_evade $\widehat{=}$
ANY *evade_val*
WHERE
grd1: $exec = ctrl$
grd2: $evade\_val \in evade\_value$
THEN
act1: $ctrlV := evade\_val$
act2: $exec := prg$
END

END

A.5 Context TimeTriggered_Ctx

CONTEXT TimeTriggered_Ctx
EXTENDS EventTriggered_Ctx
CONSTANTS *epsilon*, *safeEpsilon*
AXIOMS
axm1: $epsilon \in TIME \land sigma \mapsto epsilon \in leq$
axm2: $safeEpsilon \in (S \times RReal) \rightarrow BOOL$
axm3: $Rzero \mapsto epsilon \in lt$
END

A.6 Machine TimeTriggered_M

MACHINE TimeTriggered_M
REFINES EventTriggered_M
SEES TimeTriggered_Ctx
VARIABLES *t*, *plantV*, *ctrlV*, *exec*
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
act3: $ctrlV :\in RReal$
act4: $exec := ctrl$
END

Progress_time $\widehat{=}$
REFINES Progress
ANY *t1*
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \land (t \mapsto t1 \in lt \land minus(t1 \mapsto t) \mapsto sigma \in geq \land$
$minus(t1 \mapsto t) \mapsto epsilon \in leq)$

grd3 : $ctrlV \notin evade\_value \implies evt\_trig(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant $\widehat{=}$
REFINES Plant
ANY $plant1$
WHERE
grd1 : $exec = plant$
grd2 : $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
grd3 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd4 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV),$
$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1 : $plantV := plantV \Leftarrow plant1$
act2 : $exec := ctrl$
END

Ctrl_normal_time $\widehat{=}$
REFINES Ctrl_normal
ANY $nrml\_value$
WHERE
grd1 : $exec = ctrl$
grd2 : $nrml\_value \in RReal$
grd3 : $nrml\_value \notin evade\_value \implies safe(plantV(t) \mapsto nrml\_value) = TRUE$
grd4 : $nrml\_value \notin evade\_value \implies safeEpsilon(plantV(t) \mapsto nrml\_value) = TRUE$
THEN
act1 : $ctrlV := nrml\_value$
act2 : $exec := prg$
END

Ctrl_evade $\widehat{=}$
REFINES Ctrl_evade
ANY $evade\_val$
WHERE
grd1 : $exec = ctrl$
grd2 : $evade\_val \in evade\_value$
THEN
act1 : $ctrlV := evade\_val$
act2 : $exec := prg$
END

```
END
```

Appendices

Appendix B

Stop Sign Models

B.1 Context ContSystem_Ctx

CONTEXT ContSystem_Ctx
CONSTANTS S, $TIME$, $sigma$, $plantVInit$
AXIOMS
axm1: $S = RReal \times RReal$
axm2: $TIME = RRealPlus$
axm3: $sigma \in RRealPlus \land sigma \mapsto Rzero \in gt$
axm4: $plantVInit \in S$
END

B.2 Context Thoerems

CONTEXT Thoerems
AXIOMS
axm1: $\forall a, b, c, d \cdot a \mapsto b \in leq \land c \mapsto d \in leq \implies plus(a \mapsto c) \mapsto plus(b \mapsto d) \in leq$
axm2: $\forall a, b, c, d \cdot Rzero \mapsto a \in leq \land Rzero \mapsto b \in leq \land Rzero \mapsto c \in leq \land$
$Rzero \mapsto d \in leq \land a \mapsto b \in leq \land c \mapsto d \in leq \implies times(a \mapsto c) \mapsto times(b \mapsto d) \in leq$
axm3: $\forall a, b, c \cdot a \mapsto b \in leq \land b \mapsto c \in leq \implies a \mapsto c \in leq$
axm4: $\forall a, b \cdot a \in RReal \land b \in RReal \implies minus(times(a \mapsto a) \mapsto times(b \mapsto b)) =$
$times(plus(a \mapsto b) \mapsto minus(a \mapsto b))$
axm5: $\forall a \cdot a \in RReal \implies uminus(a) = minus(Rzero \mapsto a)$
axm6: $\forall a \cdot a \in RReal \implies a = plus(times(divide(Rone \mapsto Rtwo) \mapsto a)$
$\mapsto times(divide(Rone \mapsto Rtwo) \mapsto a))$
axm7: $\forall a, b \cdot a \in RReal \land b \in RReal \land times(a \mapsto b) \in RRealStar \implies$
$inverse(times(a \mapsto b)) = times(inverse(a) \mapsto inverse(b))$
END

B.3 Machine ContSystem_M

MACHINE ContSystem_M

SEES ContSystem_Ctx, Thoerems
VARIABLES t, $plantV$
INVARIANTS
inv1: $t \in TIME$
inv2: $plantV \in Closed2Closed(Rzero, t) \nrightarrow S$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
END

Progress $\widehat{=}$
THEN
act1: $t :| t' \in TIME \land (t \mapsto t' \in lt \land minus(t' \mapsto t) \mapsto sigma \in geq)$
END

Plant $\widehat{=}$
ANY e, $plant1$
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV), e)$
grd3: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
$\land AppendSolutionBAP(e, Closed2Closed(Rzero, t) \backslash dom(plantV),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1: $plantV := plantV \lhd\!\!+ plant1$
END

END

B.4 Context EventTriggered_Ctx

CONTEXT EventTriggered_Ctx
EXTENDS ContSystem_Ctx
SETS EXEC
CONSTANTS $safe$, evt_trig, $ctrl$, $plant$, prg, f_evol, f_evol_plantV, $evade\_value$
AXIOMS
axm1: $safe \in (S \times RReal) \rightarrow BOOL$
axm2: $evt\_trig \in S \times TIME \times RReal \rightarrow BOOL$
axm3: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm4: $f\_evol \in RReal \rightarrow S$
axm5: $f\_evol\_plantV \in (RReal \rightarrow (TIME \times S \rightarrow S))$
axm6: $\forall ctrlV \cdot ctrlV \in RReal \implies (f\_evol\_plantV(ctrlV) =$
$(\lambda\, t \mapsto plantV \cdot t \in TIME \land plantV \in S | f\_evol(ctrlV)))$

axm7: $evade\_value \subseteq RReal \land evade\_value \neq \emptyset$

END

B.5 Machine EventTriggered_M

MACHINE EventTriggered_M
REFINES ContSystem_M
SEES EventTriggered_Ctx
VARIABLES t, $plantV$, $ctrlV$, $exec$
INVARIANTS
inv1: $ctrlV \in RReal$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \implies dom(plantV) = Closed2Closed(Rzero, t)$
inv4: $exec = plant \implies t \notin dom(plantV)$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
act3: $ctrlV :\in RReal$
act4: $exec := ctrl$
END

Progress $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \land (t \mapsto t1 \in lt \land minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3: $ctrlV \notin evade\_value \implies evt\_trig(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
THEN
act1: $t := t1$
act2: $exec := plant$
END

Plant $\widehat{=}$
REFINES Plant
ANY $plant1$
WHERE
grd1: $exec = plant$
grd2: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
grd3: $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd4: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV),$
$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd5: $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$

$Closed2Closed(Rzero, t) \backslash dom(plantV), Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), plant1(t), t)$
THEN
act1: $plantV := plantV \Leftarrow plant1$
act2: $exec := ctrl$
END

Ctrl_normal $\widehat{=}$
ANY $nrml\_value$
WHERE
grd1: $exec = ctrl$
grd2: $nrml\_value \in RReal$
grd3: $nrml\_value \notin evade\_value \implies safe(plantV(t) \mapsto nrml\_value) = TRUE$
THEN
act1: $ctrlV := nrml\_value$
act2: $exec := prg$
END

Ctrl_evade $\widehat{=}$
ANY $evade\_val$
WHERE
grd1: $exec = ctrl$
grd2: $evade\_val \in evade\_value$
THEN
act1: $ctrlV := evade\_val$
act2: $exec := prg$
END

END

B.6 Context Car_Event_Ctx

CONTEXT Car_Event_Ctx
EXTENDS EventTriggered_Ctx
CONSTANTS SP, $pinit$, $vinit$, A, B
AXIOMS
axm1: (pinit, vinit)= plantVInit
axm2: $pinit \in RRealPlus \land vinit \in RRealPlus$
axm3: $pinit \mapsto SP \in leq \land SP \in RReal \land Rzero \mapsto SP \in lt \land$
$plus(pinit \mapsto divide(times(vinit \mapsto vinit) \mapsto times(Rtwo \mapsto B))) \mapsto SP \in leq$
axm4: $safe = (\lambda\ (p \mapsto v) \mapsto ctrlV \cdot (p \mapsto v) \in (RReal \times RReal) \land ctrlV \in RReal |$
$bool((plus(p \mapsto divide(times(v \mapsto v) \mapsto times(Rtwo \mapsto B))) \mapsto SP \in lt)))$
axm5: $evt\_trig = (\lambda\ (p \mapsto v) \mapsto t1 \mapsto ctrlV \cdot (p \mapsto v) \in (RReal \times RReal) \land t1 \in TIME \land$
$ctrlV \in RReal | bool((plus(plus(plus(p \mapsto times(divide(Rone \mapsto Rtwo) \mapsto$
$times(ctrlV \mapsto times(t1 \mapsto t1)))) \mapsto times(v \mapsto t1)) \mapsto$

$divide(times(v \mapsto v) \mapsto times(Rtwo \mapsto B))) \mapsto SP \in leq)))$

axm6: $\forall\, ctrlV \cdot ctrlV \in RReal \implies (f\_evol\_plantV(ctrlV) =$
$(\lambda\, t \mapsto (p \mapsto v) \cdot t \in TIME \wedge (p \mapsto v) \in (RReal \times RReal) | (v \mapsto ctrlV)))$

axm7: $evade\_value = \{uminus(B), Rzero\}$

axm8: $A \in RReal \wedge Rzero \mapsto A \in lt$

axm9: $B \in RReal \wedge Rzero \mapsto B \in lt$

axm10: $prop = (\lambda p \cdot p \in RReal | bool(p \mapsto SP \in leq))$

END

B.7 Machine Car_Event_M

MACHINE Car_Event_M
REFINES EventTriggered_M
SEES Car_Event_Ctx
VARIABLES t, $ctrlV$, $exec$, p, v
INVARIANTS

inv1: $ctrlV \in \{Rzero, uminus(B), A\}$

inv2: $p \in Closed2Closed(Rzero, t) \nrightarrow RReal \wedge v \in Closed2Closed(Rzero, t) \nrightarrow RRealPlus$
$\wedge dom(v) = dom(p)$

inv3: $plantV = bind(p, v)$

inv4: $exec \neq plant \implies dom(p) = Closed2Closed(Rzero, t) \wedge$
$dom(v) = Closed2Closed(Rzero, t)$

inv5: $exec = plant \implies t \notin dom(plantV)$

inv6: $\forall\, x \cdot x \in dom(p) \implies prop(p(x)) = TRUE$

EVENTS

INITIALISATION $\widehat{=}$
WITH $plantV' : plantV' = bind(p', v')$
THEN
act1: $t := Rzero$
act2: $p := \{Rzero \mapsto pinit\}$
act3: $v := \{Rzero \mapsto vinit\}$
act4: $ctrlV :\in RReal$
act5: $exec := ctrl$
END

Progress $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \wedge (t \mapsto t1 \in lt \wedge minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3: $ctrlV \notin evade\_value \implies$
$evt\_trig((bind(p, v))(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
THEN
act1: $t := t1$

act2 : $exec := plant$

END

Plant_event_car $\widehat{=}$

REFINES Plant

ANY $p1$, $v1$

WHERE

grd1 : $exec = plant$

grd2 : $p1 \in Closed2Closed(Rzero, t) \backslash dom(p) \rightarrow RReal \wedge$
$v1 \in Closed2Closed(Rzero, t) \backslash dom(v) \rightarrow RRealPlus$

grd3 : $ode(f\_evol\_plantV(ctrlV), (p1(t) \mapsto v1(t)), t) \in DE(RReal \times RReal)$

grd4 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(bind(p, v)),$
$ode(f\_evol\_plantV(ctrlV), bind(p1, v1)(t), t))$

grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), (bind(p1, v1))(t), t),$
$Closed2Closed(Rzero, t) \backslash dom(bind(p, v)),$
$Closed2Closed(Rzero, t) \backslash dom(bind(p, v)), bind(p1, v1))$

grd6 : $\forall xx \cdot xx \in dom(p1) \implies prop(p1(xx)) = TRUE$

WITH $plant1 : plant1 = bind(p1, v1)$

THEN

act1 : $p := p \lhd\!\!+ p1$

act2 : $v := v \lhd\!\!+ v1$

act3 : $exec := ctrl$

END

Ctrl_Acceleration_car $\widehat{=}$

REFINES Ctrl_normal

WHERE

grd1 : $exec = ctrl$

grd2 : $safe((bind(p, v))(t) \mapsto A) = TRUE$

WITH $nrml\_value : nrml\_value = A$

THEN

act1 : $ctrlV := A$

act2 : $exec := prg$

END

Ctrl_Deceleration_car $\widehat{=}$

REFINES Ctrl_evade

ANY $evade\_val$

WHERE

grd1 : $exec = ctrl$

grd2 : $evade\_val \in evade\_value$

grd3 : $v(t) \mapsto Rzero \in gt \implies evade\_val = uminus(B)$

grd4 : $v(t) = Rzero \implies evade\_val = Rzero$

THEN

act1 : $ctrlV := evade\_val$

act2 : $exec := prg$

END

END

B.8 Context Car_Time_Ctx

CONTEXT Car_Time_Ctx
EXTENDS Car_Event_Ctx
CONSTANTS $epsilon$, $safeEpsilon$
AXIOMS
axm1 : $epsilon \in TIME \land sigma \mapsto epsilon \in leq \land Rzero \mapsto epsilon \in lt$
axm2 : $safeEpsilon \in ((RReal \times RReal) \times RReal) \rightarrow BOOL$
axm3 : $safeEpsilon = (\lambda\ (p \mapsto v) \mapsto ctrlV \cdot (p \mapsto v) \in (RReal \times RReal) \land ctrlV \in RReal \mid$
$bool(plus(plus(p \mapsto plus(times(v \mapsto epsilon) \mapsto times(divide(Rone \mapsto Rtwo) \mapsto$
$times(A \mapsto times(epsilon \mapsto epsilon))))) \mapsto plus(plus(divide(times(v \mapsto v) \mapsto$
$times(Rtwo \mapsto B)) \mapsto divide(times(times(A \mapsto A) \mapsto times(epsilon \mapsto epsilon)) \mapsto$
$times(Rtwo \mapsto B))) \mapsto divide(times(A \mapsto times(epsilon \mapsto v)) \mapsto B))) \mapsto SP \in lt))$
END

B.9 Machine Car_Time_M

MACHINE Car_Time_M
REFINES Car_Event_M
SEES Car_Time_Ctx , Thoerems
VARIABLES t , $ctrlV$, $exec$, p , v
INVARIANTS
inv1 : $\exists\, t1 \cdot t1 \in TIME \land dom(p) = Closed2Closed(Rzero, t1) \land$
$minus(t \mapsto t1) \mapsto epsilon \in leq \land$
$(exec \neq plant \implies t1 = t) \land (exec = plant \implies t \mapsto t1 \in gt) \land$
$(ctrlV \notin evade\_value \land exec = plant \implies$
$safeEpsilon((p(t1) \mapsto v(t1)) \mapsto A) = TRUE)$
inv2 : $\forall\, t1 \cdot (t1 \in TIME \land dom(p) = Closed2Closed(Rzero, t1) \implies$
$plus(p(t1) \mapsto divide(times(v(t1) \mapsto v(t1)) \mapsto times(Rtwo \mapsto B))) \mapsto SP \in leq)$
inv3 : $ctrlV \notin evade\_value \land exec = prg \implies safeEpsilon((p(t) \mapsto v(t)) \mapsto A) = TRUE$
inv4 : $\forall\, t1 \cdot t1 \in TIME \land dom(p) = Closed2Closed(Rzero, t1) \land$
$ctrlV = Rzero \land exec \neq ctrl \implies v(t1) = Rzero$
inv5 : $\forall t1, t2 \cdot t1 \in TIME \land t2 \in TIME \land$
$dom(p) = Closed2Closed(Rzero, t1) \land dom(p) = Closed2Closed(Rzero, t2) \implies t1 = t2$

EVENTS
INITIALISATION $\widehat{=}$
THEN
act1 : $t := Rzero$
act2 : $p := \{Rzero \mapsto pinit\}$
act3 : $v := \{Rzero \mapsto vinit\}$

act4 : $ctrlV := Rzero$
act5 : $exec := ctrl$
END

Progress_time $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1 : $exec = prg$
grd2 : $t1 \in TIME \wedge (t \mapsto t1 \in lt \wedge minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3 : $minus(t1 \mapsto t) \mapsto epsilon \in leq$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant_time_car $\widehat{=}$
REFINES Plant_event_car
ANY $p1$, $v1$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \wedge dom(p) = Closed2Closed(Rzero, lastTime)$
grd3 : $lastTime \in dom(p) \wedge lastTime \in dom(v)$
grd4 : $ctrlV = uminus(B) \implies (minus(t \mapsto lastTime) \mapsto divide(v(lastTime) \mapsto B) \in leq$
$\implies epsilon1 = minus(t \mapsto lastTime)) \wedge (minus(t \mapsto lastTime) \mapsto$
$divide(v(lastTime) \mapsto B) \in gt \implies epsilon1 = divide(v(lastTime) \mapsto B))$
grd5 : $ctrlV \in \{Rzero, A\} \implies epsilon1 = minus(t \mapsto lastTime)$
grd6 : $p1 = (\lambda\, t1 \cdot t1 \in TIME \wedge t1 \mapsto lastTime \in gt \wedge t1 \mapsto t \in leq |$
$plus(plus(p(lastTime) \mapsto times(divide(Rone \mapsto Rtwo) \mapsto times(ctrlV \mapsto$
$times(epsilon1 \mapsto epsilon1)))) \mapsto times(v(lastTime) \mapsto epsilon1)))$
grd7 : $v1 = (\lambda\, t1 \cdot t1 \in TIME \wedge t1 \mapsto lastTime \in gt \wedge t1 \mapsto t \in leq |$
$plus(times(ctrlV \mapsto epsilon1) \mapsto v(lastTime)))$
grd8 : $ode(f\_evol\_plantV(ctrlV), (p1(t) \mapsto v1(t)), t) \in DE(RReal \times RReal)$
grd9 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(bind(p, v)),$
$ode(f\_evol\_plantV(ctrlV), bind(p1, v1)(t), t))$
grd10 : $solutionOf(Closed2Closed(Rzero, t) \backslash dom(bind(p, v)),$
$(Closed2Closed(Rzero, t) \backslash dom(bind(p, v))) \lhd bind(p1, v1),$
$ode(f\_evol\_plantV(ctrlV), bind(p1, v1)(t), t))$
THEN
act1 : $p := p \lhd\!\!+ p1$
act2 : $v := v \lhd\!\!+ v1$
act3 : $exec := ctrl$
END

Ctrl_Acceleration_car_time $\widehat{=}$
REFINES Ctrl_Acceleration_car

WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilon((p(t) \mapsto v(t)) \mapsto A) = TRUE$
THEN
act1 : $ctrlV := A$
act2 : $exec := prg$
END

Ctrl_Deceleration_car $\widehat{=}$
REFINES Ctrl_Deceleration_car
ANY evade_val
WHERE
grd1 : $exec = ctrl$
grd2 : $evade\_val \in evade\_value$
grd3 : $v(t) \mapsto Rzero \in gt \implies evade\_val = uminus(B)$
grd4 : $v(t) = Rzero \implies evade\_val = Rzero$
THEN
act1 : $ctrlV := evade\_val$
act2 : $exec := prg$
END

END

Appendices

Appendix C

Water Tank Models

C.1 Context ContSystem_Ctx

CONTEXT ContSystem_Ctx
CONSTANTS S, $TIME$, $sigma$, $plantVInit$
AXIOMS
axm1: $S = RReal$
axm2: $TIME = RRealPlus$
axm3: $sigma \in RRealPlus \land sigma \mapsto Rzero \in gt$
axm4: $plantVInit \in S$
END

C.2 Machine ContSystem_M

MACHINE ContSystem_M
SEES ContSystem_Ctx
VARIABLES t, $plantV$
INVARIANTS
inv1: $t \in TIME$
inv2: $plantV \in Closed2Closed(Rzero, t) \nrightarrow S$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
END

Progress $\widehat{=}$
THEN
act1: $t :\mid t' \in TIME \land (t \mapsto t' \in lt \land minus(t' \mapsto t) \mapsto sigma \in geq)$
END

Plant $\widehat{=}$

ANY e, $plant1$
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV), e)$
grd3: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S \wedge$
$AppendSolutionBAP(e, Closed2Closed(Rzero, t) \backslash dom(plantV),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1: $plantV := plantV \Leftarrow plant1$
END

END

C.3 Context Abstract_Tank_Ctx

CONTEXT Abstract_Tank_Ctx
EXTENDS ContSystem_Ctx
CONSTANTS $V0$, V_high, V_low, f_evol_V
AXIOMS
axm1: $V0 = plantVInit$
axm2: $V0 \in RRealPlus$
axm3 $V\_high \in RReal \wedge V\_low \in RReal \wedge V0 \mapsto V\_high \in lt \wedge$
$V0 \mapsto V\_low \in gt \wedge V\_high \mapsto V\_low \in gt \wedge V\_low \mapsto Rzero \in gt$
axm4: $f\_evol\_V \in RReal \rightarrow (TIME \times RReal \rightarrow RReal)$
axm5: $\forall ctrlV \cdot ctrlV \in RReal \implies$
$(f\_evol\_V(ctrlV) = (\lambda\, t \mapsto Vol \cdot t \in TIME \wedge Vol \in RReal | ctrlV))$
END

C.4 Machine Abstract_Tank_M

MACHINE Abstract_Tank_M
REFINES ContSystem_M
SEES Abstract_Tank_Ctx
VARIABLES t, Vol
INVARIANTS
inv1: $Vol \in Closed2Closed(Rzero, t) \nrightarrow RRealPlus$
inv2: $Vol = plantV$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $Vol := \{Rzero \mapsto V0\}$
END

Progress $\widehat{=}$

REFINES Progress
THEN
act1: $t : |t' \in TIME \land (t \mapsto t' \in lt \land minus(t' \mapsto t) \mapsto sigma \in geq)$
END

Water_behave
REFINES Plant
ANY $Vol1$, e
WHERE
grd1: $e \in DE(RReal)$
grd2: $Solvable(Closed2Closed(Rzero, t) \backslash dom(Vol), e)$
grd3: $Vol1 \in Closed2Closed(Rzero, t) \backslash dom(Vol) \rightarrow RRealPlus \land$
$AppendSolutionBAP(e, Closed2Closed(Rzero, t) \backslash dom(Vol),$
$Closed2Closed(Rzero, t) \backslash dom(Vol), Vol1)$
WITH $plant1 : plant1 = Vol1$
THEN
act1: $Vol := Vol \lhd\!\!+ Vol1$
END

END

C.5 Context Tank_Event_Ctx

CONTEXT Tank_Event_Ctx
EXTENDS Abstract_Tank_Ctx
SETS EXEC
CONSTANTS $ctrl$, $plant$, prg, $safeFill$, $safeEmp$, $evt\_trigFill$, $evt\_trigEmp$, f_in, f_out, $evade\_valueFill$, $evade\_valueEmp$, $prop$
AXIOMS
axm1: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm2: $safeFill \in (RReal \times RReal) \rightarrow BOOL$
axm3: $safeEmp \in (RReal \times RReal) \rightarrow BOOL$
axm4: $evt\_trigFill \in (RReal \times TIME) \times RReal \rightarrow BOOL$
axm5: $evt\_trigEmp \in (RReal \times TIME) \times RReal \rightarrow BOOL$
axm6: $safeFill = (\lambda\, vol \mapsto ctrlV \cdot vol \in RReal \land ctrlV \in RReal | bool(vol \mapsto V\_high \in lt))$
axm7: $safeEmp = (\lambda\, vol \mapsto ctrlV \cdot vol \in RReal \land ctrlV \in RReal | bool(vol \mapsto V\_low \in gt))$
axm8: $evt\_trigFill = (\lambda\, vol \mapsto t1 \mapsto ctrlV \cdot vol \in RReal \land t1 \in TIME \land ctrlV \in RReal |$
$bool(plus(vol \mapsto times(ctrlV \mapsto t1)) \mapsto V\_high \in leq))$
axm9: $evt\_trigEmp = (\lambda\, vol \mapsto t1 \mapsto ctrlV \cdot vol \in RReal \land t1 \in TIME \land ctrlV \in RReal |$
$bool(plus(vol \mapsto times(ctrlV \mapsto t1)) \mapsto V\_low \in geq))$
axm10: $f\_in \in RReal \land f\_in \mapsto Rzero \in gt$
axm11: $f\_out \in RReal \land f\_out \mapsto Rzero \in gt$
axm12: $evade\_valueFill \subseteq RReal \land evade\_valueFill = \{uminus(f\_out)\}$
axm13: $evade\_valueEmp \subseteq RReal \land evade\_valueEmp = \{f\_in\}$
axm14: $prop \in RReal \rightarrow BOOL$

axm15: $prop = (\lambda vol \cdot vol \in RReal | bool(vol \mapsto V\_high \in leq \wedge vol \mapsto V\_min \in geq))$

END

C.6 Machine Tank_Event_M

MACHINE Tank_Event_M
REFINES Abstract_Tank_M
SEES Tank_Event_Ctx
VARIABLES t, Vol, $ctrlV$, $exec$
INVARIANTS
inv1: $ctrlV \in \{f\_in, uminus(f\_out)\}$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \implies dom(Vol) = Closed2Closed(Rzero, t)$
inv4: $exec = plant \implies t \notin dom(Vol)$
inv5: $\forall x \cdot x \in dom(Vol) \implies Vol(x) \mapsto V\_high \in leq \wedge Vol(x) \mapsto V\_low \in geq$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $Vol := \{Rzero \mapsto V0\}$
act3: $ctrlV := f\_in$
act4: $exec := ctrl$
END

Progress $\widehat{=}$
REFINES Progress
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \wedge (t \mapsto t1 \in lt) \wedge minus(t1 \mapsto t) \mapsto sigma \in geq$
grd3: $ctrlV \notin evade\_valueFill \implies$
$evt\_trigFill(Vol(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
grd4: $ctrlV \notin evade\_valueEmp \implies$
$evt\_trigEmp(Vol(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE)$
THEN
act1: $t := t1$
act2: $exec := plant$
END

Plant_event_tank $\widehat{=}$
REFINES Water_behave
ANY $Vol1$
WHERE
grd1: $exec = plant$
grd2: $Vol1 \in Closed2Closed(Rzero, t) \backslash dom(Vol) \rightarrow RRealPlus$
grd3: $ode(f\_evol\_V(ctrlV), Vol1(t), t) \in DE(RReal)$

grd4 : $Solvable(Closed2Closed(Rzero, t)\backslash dom(Vol), ode(f\_evol\_V(ctrlV), Vol1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_V(ctrlV), Vol1(t), t),$
$Closed2Closed(Rzero, t)\backslash dom(Vol),$
$Closed2Closed(Rzero, t)\backslash dom(Vol), Vol1)$
grd6 : $\forall xx \cdot xx \in dom(Vol1) \implies prop(Vol1(xx)) = TRUE$

WITH $e : e = ode(f\_evol\_V(ctrlV), Vol1(t), t)$
THEN
act1 : $Vol := Vol \lhd\!\!+ Vol1$
act2 : $exec := ctrl$
END

Ctrl_normal $\widehat{=}$
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{f\_in, uminus(f\_out)\}$
grd3 : $nCtrlV = f\_in \implies safeFill(Vol(t) \mapsto f\_in) = TRUE$
grd4 : $nCtrlV = uminus(f\_out) \implies safeEmp(Vol(t) \mapsto uminus(f\_out)) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Ctrl_emptying $\widehat{=}$
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEmp(Vol(t) \mapsto uminus(f\_out)) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := uminus(f\_out)$
END

Ctrl_filling $\widehat{=}$
WHERE
grd1 : $exec = ctrl$
grd2 : $safeFill(Vol(t) \mapsto f\_in) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := f\_in$
END

END

C.7 Context Tank_Time_Ctx

CONTEXT Tank_Time_Ctx
EXTENDS Tank_Event_Ctx
CONSTANTS $epsilon$, $safeEpsilonFill$, $safeEpsilonEmp$
AXIOMS
axm1: $epsilon \in TIME \land Rzero \mapsto epsilon \in lt \land sigma \mapsto epsilon \in leq$
axm2: $safeEpsilonFill \in (RReal \times RReal) \rightarrow BOOL$
axm2: $safeEpsilonEmp \in (RReal \times RReal) \rightarrow BOOL$
axm3: $safeEpsilonFill = (\lambda\, vol \mapsto ctrlV \cdot vol \in RReal \land ctrlV \in RReal |$
$bool(plus(vol \mapsto times(ctrlV \mapsto epsilon)) \mapsto V\_high \in leq))$
axm4: $safeEpsilonEmp = (\lambda\, vol \mapsto ctrlV \cdot vol \in RReal \land ctrlV \in RReal |$
$bool(plus(vol \mapsto times(ctrlV \mapsto epsilon)) \mapsto V\_low \in geq))$
END

C.8 Machine Tank_Time_M

MACHINE Tank_Time_M
REFINES Tank_Event_M
SEES Tank_Time_Ctx, Theorems
VARIABLES t, Vol, $ctrlV$, $exec$
INVARIANTS
inv1: $\exists\, t1 \cdot t1 \in TIME \land dom(Vol) = Closed2Closed(Rzero, t1) \land$
$minus(t \mapsto t1) \mapsto epsilon \in leq \land$
$(exec \neq plant \implies t1 = t) \land (exec = plant \implies t \mapsto t1 \in gt) \land$
$(ctrlV \notin evade\_valueFill \land exec = plant \implies$
$safeEpsilonFill(Vol(t1) \mapsto ctrlV) = TRUE) \land (ctrlV \notin evade\_valueEmp \land$
$exec = plant \implies safeEpsilonEmp(Vol(t1) \mapsto ctrlV) = TRUE)$
inv2:
$ctrlV \notin evade\_valueFill \land exec = prg \implies safeEpsilonFill(Vol(t) \mapsto ctrlV) = TRUE$
inv3: $ctrlV \notin evade\_valueEmp \land exec = prg \implies$
$safeEpsilonEmp(Vol(t) \mapsto ctrlV) = TRUE$
inv4: $\forall\, t1, t2 \cdot t1 \in TIME \land t2 \in TIME \land$
$dom(Vol) = Closed2Closed(Rzero, t1) \land dom(Vol) = Closed2Closed(Rzero, t2) \implies t1 = t2$

EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $Vol := \{Rzero \mapsto V0\}$
act3: $ctrlV := f\_in$
act4: $exec := ctrl$
END

Progrss_time $\widehat{=}$
REFINES Progress
WHERE

grd1 : $exec = prg$
grd2 : $t1 \in TIME \land (t \mapsto t1 \in lt) \land (minus(t1 \mapsto t) \mapsto sigma \in geq) \land$
$(minus(t1 \mapsto t) \mapsto epsilon \in leq)$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant_time_tank $\widehat{=}$
REFINES Plant_event_tank
ANY $Vol1$, $lastTime$, $epsilon1$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \land dom(Vol) = Closed2Closed(Rzero, lastTime)$
grd3 : $t \mapsto lastTime \in gt \land lastTime \in dom(Vol)$
grd4 : $epsilon1 = minus(t \mapsto lastTime)$
grd5 : $Vol1 = (\lambda\, t1 \cdot t1 \in TIME \land t1 \mapsto lastTime \in gt \land t1 \mapsto t \in leq |$
$plus(times(ctrlV \mapsto epsilon1) \mapsto Vol(lastTime)))$
grd6 : $ode(f\_evol\_V(ctrlV), Vol1(t), t) \in DE(RReal)$
grd7 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(Vol), ode(f\_evol\_V(ctrlV), Vol1(t), t))$
grd8 : $solutionOf(Closed2Closed(Rzero, t) \backslash dom(Vol),$
$(Closed2Closed(Rzero, t) \backslash dom(Vol)) \lhd Vol1, ode(f\_evol\_V(ctrlV), Vol1(t), t))$
THEN
act1 : $Vol := Vol \lhd\!\!+ Vol1$
act2 : $exec := ctrl$
END

Ctrl_normal_time $\widehat{=}$
REFINES Ctrl_normal
ANY $nCtrlV$
WHERE
grd1 : $exec = ctrl$
grd2 : $nCtrlV \in \{f\_in, uminus(f\_out)\}$
grd3 : $nCtrlV = f\_in \implies safeEpsilonFill(Vol(t) \mapsto f\_in) = TRUE$
grd4 :
$nCtrlV = uminus(f\_out) \implies safeEpsilonEmp(Vol(t) \mapsto uminus(f\_out)) = TRUE$
THEN
act1 : $exec := prg$
act2 : $ctrlV := nCtrlV$
END

Ctrl_emptying $\widehat{=}$
REFINES Ctrl_emptying
WHERE
grd1 : $exec = ctrl$
grd2 : $safeEpsilonEmp(Vol(t) \mapsto uminus(f\_out)) = TRUE$

THEN
act1: $exec := prg$
act2: $ctrlV := uminus(f\_out)$
END

Ctrl_filling $\widehat{=}$
REFINES Ctrl_filling
WHERE
grd1: $exec = ctrl$
grd2: $safeEpsilonFill(Vol(t) \mapsto f\_in) = TRUE$
THEN
act1: $exec := prg$
act2: $ctrlV := f\_in$
END

END

Appendices

Appendix D

Smart Heating Models

D.1 Context ContSystem_Ctx

CONTEXT ContSystem_Ctx
CONSTANTS S, $TIME$, $sigma$, $plantVInit$
AXIOMS
axm1: $S = RReal$
axm2: $TIME = RRealPlus$
axm3: $sigma \in RRealPlus \land sigma \mapsto Rzero \in gt$
axm4: $plantVInit \in RReal$
END

D.2 Context Thoerems

CONTEXT Thoerems
AXIOMS
axm1: $\forall a, b, c, d \cdot a \mapsto b \in leq \land c \mapsto d \in leq \implies plus(a \mapsto c) \mapsto plus(b \mapsto d) \in leq$
axm2: $\forall a, b, c, d \cdot Rzero \mapsto a \in leq \land Rzero \mapsto b \in leq \land Rzero \mapsto c \in leq \land$
$Rzero \mapsto d \in leq \land a \mapsto b \in leq \land c \mapsto d \in leq \implies times(a \mapsto c) \mapsto times(b \mapsto d) \in leq$
axm3: $\forall a, b, c \cdot a \mapsto b \in leq \land b \mapsto c \in leq \implies a \mapsto c \in leq$
axm4: $\forall a, b \cdot a \in RReal \land b \in RReal \implies minus(times(a \mapsto a) \mapsto times(b \mapsto b)) =$
$times(plus(a \mapsto b) \mapsto minus(a \mapsto b))$
axm5: $\forall a \cdot a \in RReal \implies uminus(a) = minus(Rzero \mapsto a)$
axm6: $\forall a \cdot a \in RReal \implies a = plus(times(divide(Rone \mapsto Rtwo) \mapsto a)$
$\mapsto times(divide(Rone \mapsto Rtwo) \mapsto a))$
axm7: $\forall a, b \cdot a \in RReal \land b \in RReal \land times(a \mapsto b) \in RRealStar \implies$
$inverse(times(a \mapsto b)) = times(inverse(a) \mapsto inverse(b))$
END

D.3 Machine ContSystem_M

MACHINE ContSystem_M

SEES ContSystem_Ctx, Thoerems
VARIABLES t, $plantV$
INVARIANTS
inv1: $t \in TIME$
inv2: $plantV \in Closed2Closed(Rzero, t) \nrightarrow S$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
END

Progress $\widehat{=}$
THEN
act1: $t :| t' \in TIME \wedge (t \mapsto t' \in lt \wedge minus(t' \mapsto t) \mapsto sigma \in geq)$
END

Plant $\widehat{=}$
ANY e, $plant1$
WHERE
grd1: $e \in DE(S)$
grd2: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV), e)$
grd3: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
$\wedge AppendSolutionBAP(e, Closed2Closed(Rzero, t) \backslash dom(plantV),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1: $plantV := plantV \lhd\!\!+ plant1$
END

END

D.4 Context EventTriggered_Ctx

CONTEXT EventTriggered_Ctx
EXTENDS ContSystem_Ctx
SETS EXEC, PROP
CONSTANTS $prop\_safe$, $prop\_evt\_trig$, $ctrl$, $plant$, prg, f_evol, f_evol_plantV, $prop\_evade\_values$
AXIOMS
axm1: $prop\_safe \in PROP \rightarrow ((S \times RReal) \rightarrow BOOL)$
axm2: $prop\_evt\_trig \in PROP \rightarrow (S \times TIME \times RReal \rightarrow BOOL)$
axm3: $partition(EXEC, \{ctrl\}, \{plant\}, \{prg\})$
axm4: $f\_evol \in RReal \rightarrow S$
axm5: $f\_evol\_plantV \in (RReal \rightarrow (TIME \times S \rightarrow S))$
axm6: $\forall\, ctrlV \cdot ctrlV \in RReal \implies (f\_evol\_plantV(ctrlV) =$

$(\lambda\, t \mapsto plantV \cdot t \in TIME \land plantV \in S | f\_evol(ctrlV)))$
axm7: $prop\_evade\_values \in PROP \to P1(RReal)$
END

D.5 Machine EventTriggered_M

MACHINE EventTriggered_M
REFINES ContSystem_M
SEES EventTriggered_Ctx
VARIABLES t, $plantV$, $ctrlV$, $exec$
INVARIANTS
inv1: $ctrlV \in RReal$
inv2: $exec \in EXEC$
inv3: $exec \neq plant \implies dom(plantV) = Closed2Closed(Rzero, t)$
inv4: $exec = plant \implies t \notin dom(plantV)$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
act3: $ctrlV :\in RReal$
act4: $exec := ctrl$
END

Progress $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \land (t \mapsto t1 \in lt \land minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3: $\forall\, x \cdot x \in PROP \implies (ctrlV \notin prop\_evade\_values(x) \implies$
$(prop\_evt\_trig(x))(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
THEN
act1: $t := t1$
act2: $exec := plant$
END

Plant $\widehat{=}$
REFINES Plant
ANY $plant1$
WHERE
grd1: $exec = plant$
grd2: $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \to S$
grd3: $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd4: $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV),$

$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd5: $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$Closed2Closed(Rzero, t)\backslash dom(plantV), Closed2Closed(Rzero, t)\backslash dom(plantV), plant1)$
WITH $e : e = ode(f\_evol\_plantV(ctrlV), plant1(t), t)$
THEN
act1: $plantV := plantV \lhd\!\!+ plant1$
act2: $exec := ctrl$
END

Ctrl $\widehat{=}$
ANY *value*
WHERE
grd1: $exec = ctrl$
grd2: $value \in RReal$
grd3: $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies$
$(prop\_safe(x))(plantV(t) \mapsto value) = TRUE)$
THEN
act1: $ctrlV := value$
act2: $exec := prg$
END

END

D.6 Context TimeTriggered_Ctx

CONTEXT TimeTriggered_Ctx
EXTENDS EventTriggered_Ctx
CONSTANTS *epsilon*, *prop_safeEpsilon*
AXIOMS
axm1: $epsilon \in TIME \land sigma \mapsto epsilon \in leq$
axm2: $prop\_safeEpsilon \in PROP \to ((S \times RReal) \to BOOL)$
axm3: $Rzero \mapsto epsilon \in lt$
END

D.7 Machine TimeTriggered_M

MACHINE TimeTriggered_M
REFINES EventTriggered_M
SEES TimeTriggered_Ctx, Theorems
VARIABLES *t*, *plantV*, *ctrlV*, *exec*
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$

act3 : $ctrlV :\in RReal$
act4 : $exec := ctrl$
END

Progress_time $\widehat{=}$
REFINES Progress
ANY $t1$
WHERE
grd1 : $exec = prg$
grd2 : $t1 \in TIME \wedge (t \mapsto t1 \in lt \wedge minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3 : $\forall x \cdot x \in PROP \implies (ctrlV \notin prop\_evade\_values(x) \implies (prop\_evt\_trig(x))(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
grd4 : $t1 \in TIME \wedge (t \mapsto t1 \in lt) \wedge minus(t1 \mapsto t) \mapsto sigma \in geq \wedge minus(t1 \mapsto t) \mapsto epsilon \in leq$
THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant_time $\widehat{=}$
REFINES Plant
ANY $plant1$
WHERE
grd1 : $exec = plant$
grd2 : $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
grd3 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd4 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV), ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd5 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t), Closed2Closed(Rzero, t) \backslash dom(plantV), Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
THEN
act1 : $plantV := plantV \lhd\!\!+ plant1$
act2 : $exec := ctrl$
END

Ctrl $\widehat{=}$
REFINES Ctrl
ANY $value$
WHERE
grd1 : $exec = ctrl$
grd2 : $value \in RReal$
grd3 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies (prop\_safe(x))(plantV(t) \mapsto value) = TRUE)$
grd4 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies (prop\_safeEpsilon(x))(plantV(t) \mapsto value) = TRUE)$
THEN

act1: $ctrlV := value$
act2: $exec := prg$
END

END

D.8 Context Desolve

CONTEXT Desolve_Ctx
EXTENDS TimeTriggered_Ctx
CONSTANTS $B\_desolve$, $prop$
AXIOMS
axm1: $B\_desolve \in \mathbb{N} \times RReal \times (TIME \nrightarrow RReal) \times TIME \times (TIME \times RReal) \rightarrow (RReal \nrightarrow RReal)$
axm2: $prop \in S \rightarrow BOOL$
axm3: $prop(plantVInit) = TRUE$
END

D.9 TimeTriggered_desolve_M

MACHINE TimeTriggered_desolve_M
REFINES TimeTriggered_M
SEES Desolve, Theorems
VARIABLES t, $plantV$, $ctrlV$, $exec$
INVARIANTS
inv1: $\forall x \cdot x \in dom(plantV) \implies prop(plantV(x)) = TRUE$
EVENTS
INITIALISATION $\widehat{=}$
THEN
act1: $t := Rzero$
act2: $plantV := \{Rzero \mapsto plantVInit\}$
act3: $ctrlV :\in RReal$
act4: $exec := ctrl$
END

Progress_time $\widehat{=}$
REFINES Progress_time
ANY $t1$
WHERE
grd1: $exec = prg$
grd2: $t1 \in TIME \land (t \mapsto t1 \in lt \land minus(t1 \mapsto t) \mapsto sigma \in geq)$
grd3: $\forall x \cdot x \in PROP \implies (ctrlV \notin prop\_evade\_values(x) \implies (prop\_evt\_trig(x))(plantV(t) \mapsto minus(t1 \mapsto t) \mapsto ctrlV) = TRUE$
grd4: $t1 \in TIME \land (t \mapsto t1 \in lt) \land minus(t1 \mapsto t) \mapsto sigma \in geq \land minus(t1 \mapsto t) \mapsto epsilon \in leq$

THEN
act1 : $t := t1$
act2 : $exec := plant$
END

Plant_time_desolve $\widehat{=}$
REFINES Plant_time
ANY $plant1$, $lastTime$
WHERE
grd1 : $exec = plant$
grd2 : $lastTime \in TIME \land dom(plantV) = Closed2Closed(Rzero, lastTime)$
grd3 : $plant1 = B\_desolve(1 \mapsto ctrlV \mapsto plantV \mapsto t \mapsto (lastTime \mapsto plantV(lastTime)))$
grd4 : $plant1 \in Closed2Closed(Rzero, t) \backslash dom(plantV) \rightarrow S$
grd5 : $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(S)$
grd6 : $Solvable(Closed2Closed(Rzero, t) \backslash dom(plantV),$
$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$
grd7 : $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$Closed2Closed(Rzero, t) \backslash dom(plantV), Closed2Closed(Rzero, t) \backslash dom(plantV), plant1)$
grd8 : $\forall xx \cdot xx \in dom(plant1) \implies prop(plant1(xx)) = TRUE$
THEN
act1 : $plantV := plantV \lhd plant1$
act2 : $exec := ctrl$
END

Ctrl $\widehat{=}$
REFINES Ctrl
ANY $value$
WHERE
grd1 : $exec = ctrl$
grd2 : $value \in RReal$
grd3 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies$
$(prop\_safe(x))(plantV(t) \mapsto value) = TRUE)$
grd4 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies$
$(prop\_safeEpsilon(x))(plantV(t) \mapsto value) = TRUE)$
THEN
act1 : $ctrlV := value$
act2 : $exec := prg$
END

END

D.10 CONTEXT Heater_Ctx

CONTEXT Heater_ctx
EXTENDS Desolve_Ctx

CONSTANTS $p1, p2, prop\_val, T\_max, T\_min, T0, temp$
AXIOMS
axm1: $T0 \in RRealPlus \land T0 = plantVInit$
axm2: $T\_max \in RReal \land T\_min \in RReal \land T\_max \mapsto T\_min \in gt \land$
$T\_min \mapsto Rzero \in gt \land T0 \mapsto T\_max \in leq \land T0 \mapsto T\_min \in geq$
axm3: $prop\_val \in PROP \to P(RReal \times BOOL)$
axm4: $PROP = \{p1, p2\}$
axm5: $prop\_val = \{p1 \mapsto (\lambda\, T \cdot T \in RReal \mid bool(T\_min \mapsto T \in leq)),$
$p2 \mapsto (\lambda\, T \cdot T \in RReal \mid bool(T \mapsto T\_max \in leq))\}$
axm6: $prop = (\lambda\, T \cdot T \in RReal \mid bool((prop\_val(p1))(t) = TRUE \land$
$(prop\_val(p2))(t) = TRUE))$
axm7: $prop\_safe = \{$
$p1 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in RReal \land ctrlV \in RReal \mid bool(T \mapsto T\_max \in leq)),$
$p2 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in RReal \land ctrlV \in RReal \mid bool(T \mapsto T\_min \in geq))\}$
axm8: $prop\_safeEpsilon = \{$
$p1 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in RReal \land ctrlV \in RReal \mid$
$bool(plus(T \mapsto times(ctrlV \mapsto epsilon)) \mapsto T\_max \in leq)),$
$p2 \mapsto (\lambda\, T \mapsto ctrlV \cdot T \in RReal \land ctrlV \in RReal \mid$
$bool(plus(T \mapsto times(ctrlV \mapsto epsilon)) \mapsto T\_min \in geq))\}$
axm9: $prop\_evt\_trig = \{$
$p1 \mapsto (\lambda\, T \mapsto t1 \mapsto ctrlV \cdot T \in RReal \land t1 \in TIME \land ctrlV \in RReal \mid$
$bool(plus(T \mapsto times(ctrlV \mapsto t1)) \mapsto T\_max \in leq)),$
$p2 \mapsto (\lambda\, T \mapsto t1 \mapsto ctrlV \cdot T \in RReal \land t1 \in TIME \land ctrlV \in RReal \mid$
$bool(plus(T \mapsto times(ctrlV \mapsto t1)) \mapsto T\_min \in geq))\}$
axm10: $temp \in RReal \land temp \mapsto Rzero \in gt$
axm11: $prop\_evade\_values = \{p1 \mapsto \{uminus(temp)\}, p2 \mapsto \{temp\}\}$
END

D.11 Machine Heater_M

MACHINE Heater_M
REFINES TimeTriggered_desolve_M
SEES Heater_ctx, Theorems
VARIABLES t, T, $ctrlV$, $exec$
INVARIANTS
inv1: $T = plantV \land ran(T) \subseteq RReal$
inv2: $ctrlV \in \{temp, uminus(temp)\}$
inv3: $exec \neq plant \implies dom(T) = Closed2Closed(Rzero, t)$
inv4: $exec = plant \implies t \notin dom(T)$
inv5: $\forall x \cdot x \in dom(T) \implies prop(T(x)) = TRUE$
inv6: $\exists\, t1 \cdot t1 \in TIME \land dom(T) = Closed2Closed(Rzero, t1) \land$
$minus(t \mapsto t1) \mapsto epsilon \in leq \land (exec \neq plant \implies t1 = t) \land (exec = plant \implies$
$t \mapsto t1 \in gt) \land (\forall\, x \cdot x \in PROP \land ctrlV \notin prop\_evade\_values(x) \land exec = plant \implies$
$(prop\_safeEpsilon(x))(T(t1) \mapsto ctrlV) = TRUE)$
inv7: $\forall\, x \cdot x \in PROP \land ctrlV \notin prop\_evade\_values(x) \land exec = prg \implies$

$(prop\_safeEpsilon(x))(T(t) \mapsto ctrlV) = TRUE$

inv8: $\forall t1, t2 \cdot t1 \in TIME \wedge t2 \in TIME \wedge$
$dom(T) = Closed2Closed(Rzero, t1) \wedge dom(T) = Closed2Closed(Rzero, t2) \implies t1 = t2$

EVENTS

INITIALISATION $\widehat{=}$

THEN

act1: $t := Rzero$

act2: $T := \{Rzero \mapsto T0\}$

act3: $ctrlV := temp$

act4: $exec := ctrl$

END

Progress_time $\widehat{=}$

REFINES Progress_time

ANY $t1$

WHERE

grd1: $exec = prg$

grd2: $t1 \in TIME \wedge t \mapsto t1 \in lt \wedge minus(t1 \mapsto t) \mapsto sigma \in geq$
$\wedge\ minus(t1 \mapsto t) \mapsto epsilon \in leq$

THEN

act1: $t := t1$

act2: $exec := plant$

END

Thermostat_plant $\widehat{=}$

REFINES Plant_time_desolve

ANY $plant1$, $lastTime$

WHERE

grd1: $exec = plant$

grd2: $lastTime \in TIME \wedge dom(T) = Closed2Closed(Rzero, lastTime)$

grd3: $plant1 = B\_desolve(1 \mapsto ctrlV \mapsto T \mapsto t \mapsto (lastTime \mapsto T(lastTime)))$

grd4: $B\_desolve(1 \mapsto ctrlV \mapsto T \mapsto t \mapsto (lastTime \mapsto T(lastTime))) =$
$(\lambda t1 \cdot t1 \in TIME \wedge t1 \mapsto lastTime \in gt \wedge t1 \mapsto t \in leq |$
$plus(plus(times(ctrlV \mapsto uminus(lastTime)) \mapsto times(ctrlV \mapsto t1)) \mapsto T(lastTime)))$

grd4: $ode(f\_evol\_plantV(ctrlV), plant1(t), t) \in DE(RReal)$

grd5: $Solvable(Closed2Closed(Rzero, t) \backslash dom(T),$
$ode(f\_evol\_plantV(ctrlV), plant1(t), t))$

grd6: $AppendSolutionBAP(ode(f\_evol\_plantV(ctrlV), plant1(t), t),$
$Closed2Closed(Rzero, t) \backslash dom(T), Closed2Closed(Rzero, t) \backslash dom(T), plant1)$

THEN

act1: $T := T \leftarrow plant1$

act2: $exec := ctrl$

END

Ctrl $\widehat{=}$

REFINES Ctrl

ANY $value$
WHERE
grd1 : $exec = ctrl$
grd2 : $value \in \{temp, uminus(temp)\}$
grd3 : $\forall x \cdot x \in PROP \implies (value \notin prop\_evade\_values(x) \implies (prop\_safeEpsilon(x))(T(t) \mapsto value) = TRUE)$
THEN
act1 : $ctrlV := value$
act2 : $exec := prg$
END

END

Appendices

Appendix E

User Manual for the Plugin SAGEMATH

- To use the SageMath plugin, the user must launch the Eclipse platform and then open the workspace containing our plugin project called "fr.upec.sageplugin" (see Figure E.1). This project contains various classes that we used to define the functionality provided by the plugin (see Figure E.2).

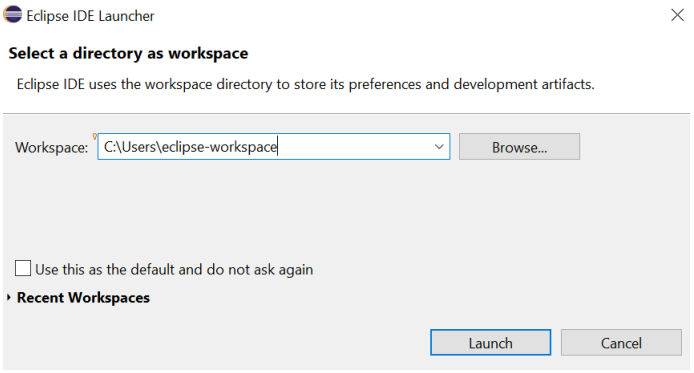

Figure E.1: Using plugin SageMath: Step 1.

- Then the user should launch Rodin from within eclipse, using Rodin as the target platform. For this purpose, first open the "Run/Run configurations" menu (see Figure E.3) and double click on "Eclipse Application" on the left to create a new configuration and rename the configuration to "Rodin 3.5". On the "Main" tab, select "Location" for the run-time workspace and select "org.rodin.platform.product" for the "Run a product" option (see Figure E.4). In the "Plug-ins", for "Launch with" option, choose "plug-ins selected below only". In the Plugins list, disable all test plugins for the Target platform (see Figure E.5).

- The user can then run a version of Rodin with the SageMath plugin integrated and upload the project containing the differential equations to be solved with SageMath, remembering to open the project named "SimpleDEq" containing the theory needed to prove our Event-B models (see Figure E.6).

- The user can finally open the proof obligations that contains the terms $B\_desolve$ in order to call SageMath. This is done by clicking in the goal tab on the left-hand side to get a button called SageMath (see Figure E.7).

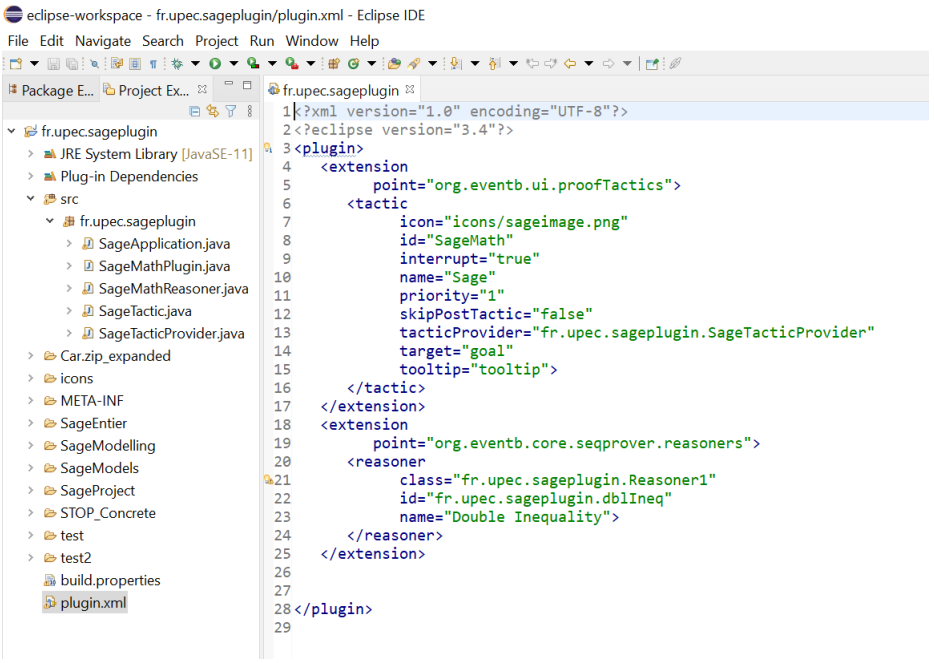

Figure E.2: Using plugin SageMath: Step 2.

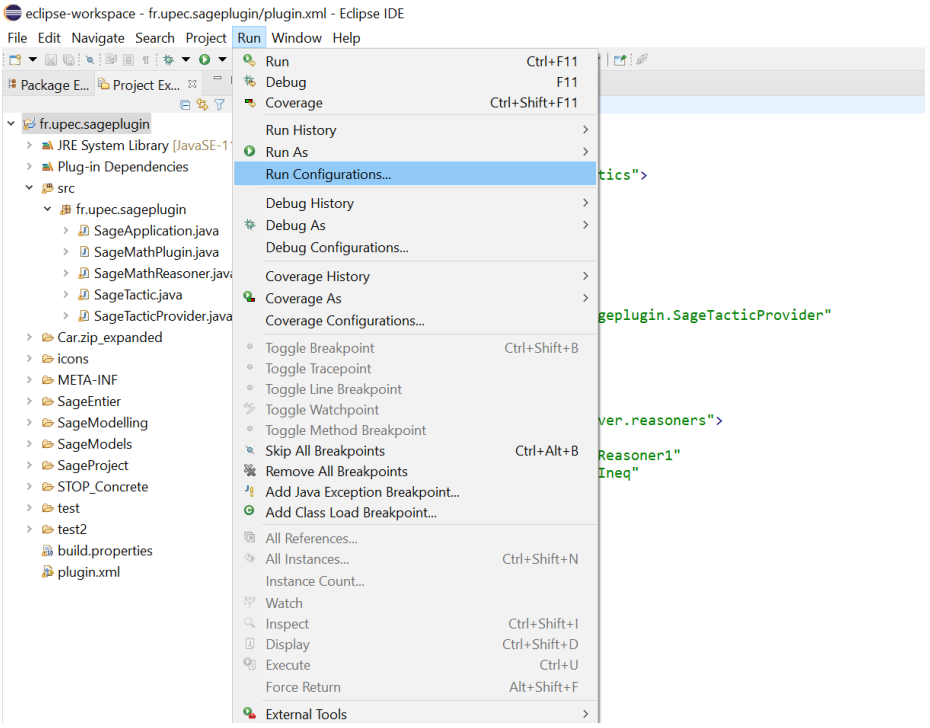

Figure E.3: Using plugin SageMath: Step 3.

The user can eventually open the proof obligations containing the terms, *B_desolve* or *B_desolve_rk4*, and then invoke SageMath. To do this, click on the left side of the goal tab to get a button called SageMath (see Figure E.7).

- Finally, the user calls SageMath and executes the command line load("script1.sage"). This allows the user to execute the statements defined in "script1.sage" script in order

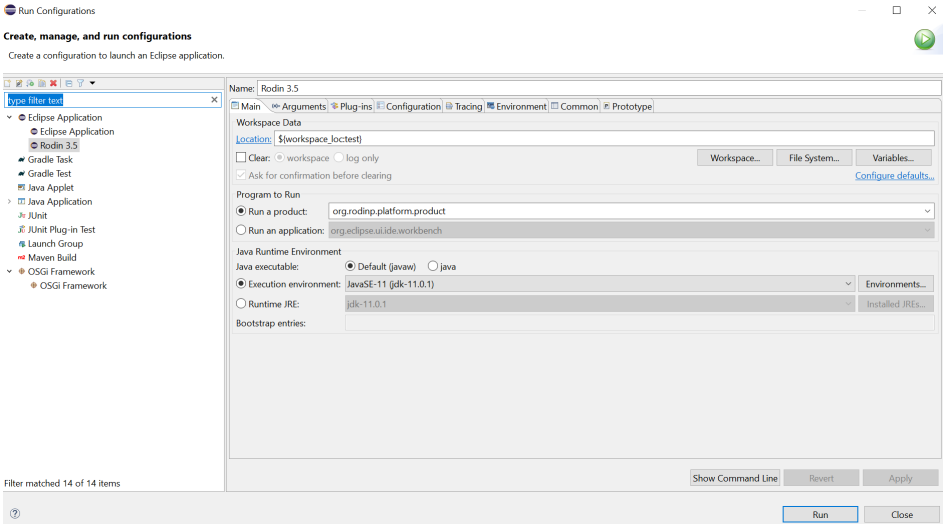

Figure E.4: Using plugin SageMath: Step 4.

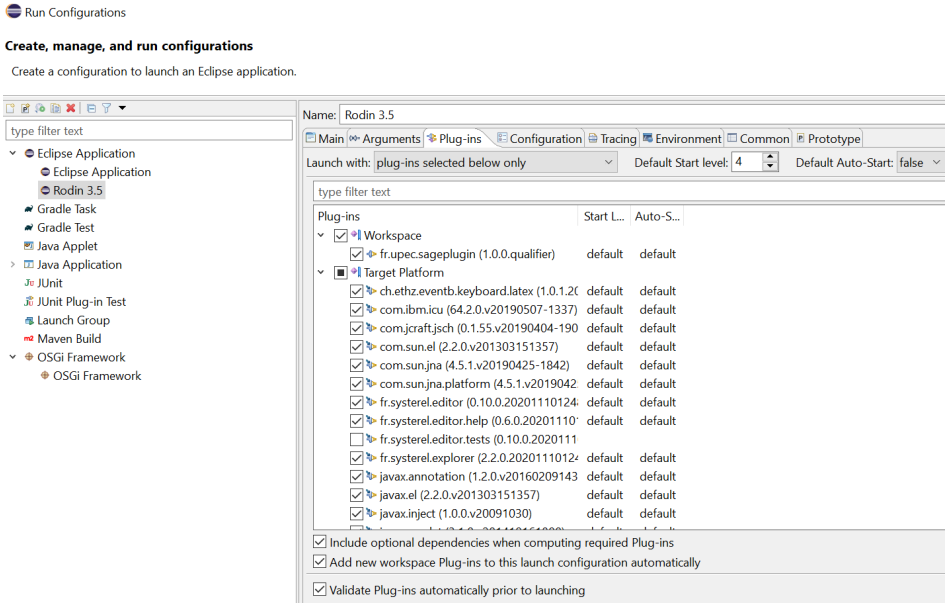

Figure E.5: Using plugin SageMath: Step 5.

to solve the ordinary differential equation determined by the function $B\_desolve$ in the current proof obligation (see Figure E.8).

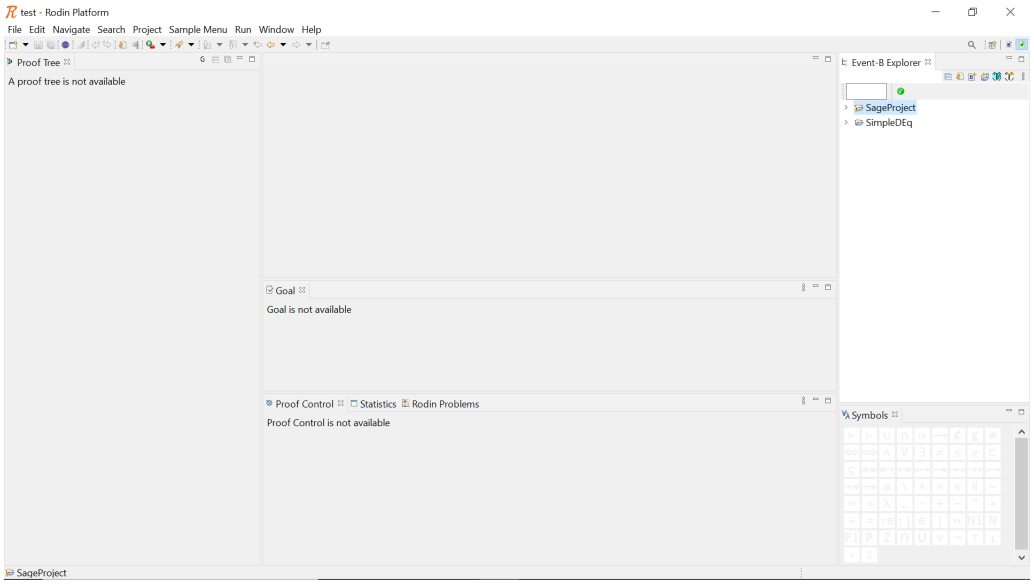

Figure E.6: Using plugin SageMath: Step 6.

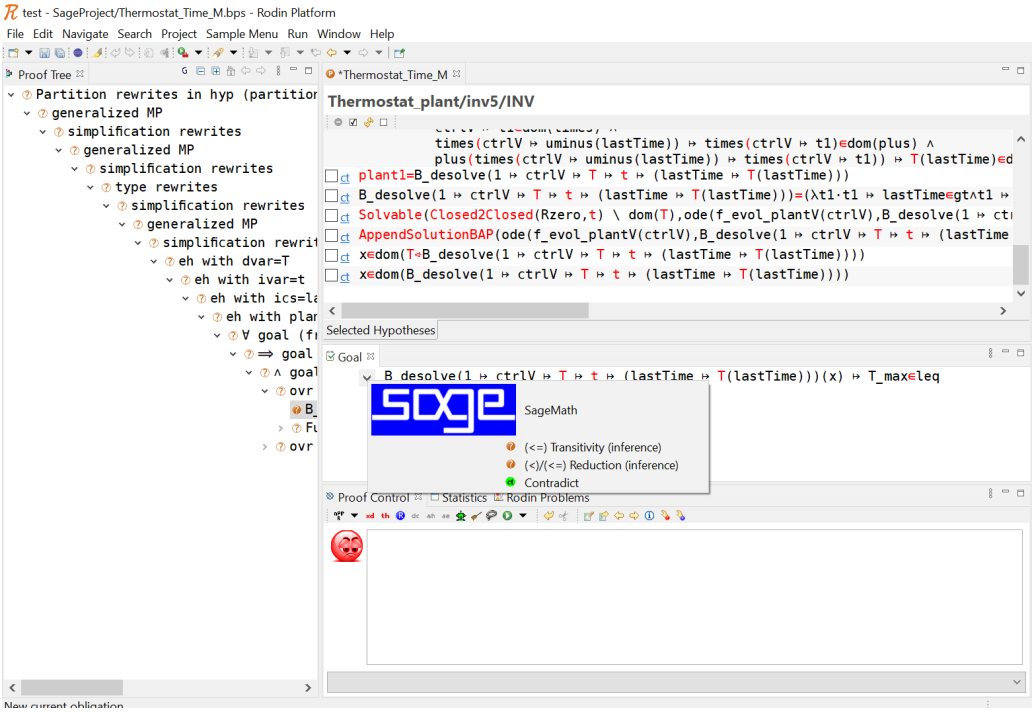

Figure E.7: Using plugin SageMath: Step 7.

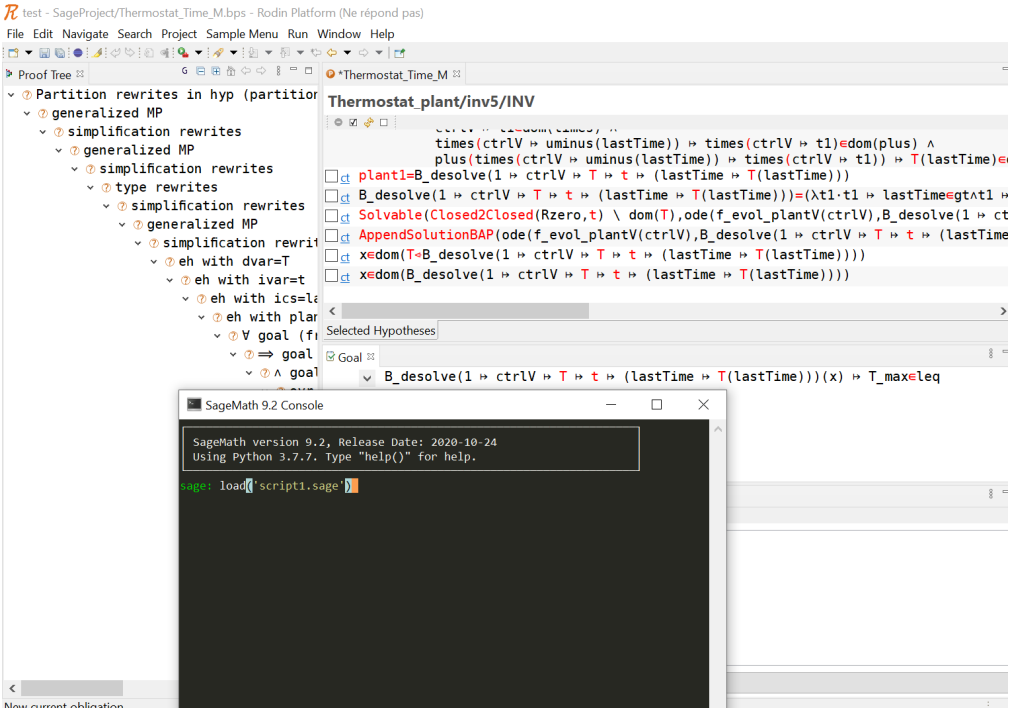

Figure E.8: Using plugin SageMath: Step 8.

Une Approche Correcte par Construction pour la Modélisation et la Vérification de systèmes cyber-physiques dans Event-B

AFENDI MERYEM

1 Introduction

Les progrès récents dans le secteur industriel ont permis le développement d'un nouveau modèle de production basé sur les architectures numériques en réseau ou "usines connectées". Ce nouveau modèle de production a donné naissance à "l'industrie 4.0" ou "industrie du futur". Les systèmes cyber-physiques (SCPs) [1] sont l'une des principales technologies de cette industrie et forment donc la base des technologies du futur. Le domaine de ces systèmes est devenu rapidement une source d'innovation avec des applications dans tous les secteurs : santé, transport, smart grid, etc. Ces systèmes connectent le monde virtuel discret et le monde physique continu via un réseau de capteurs et d'actionneurs.

Le modèle mathématique adapté aux SCPs est celui des systèmes hybrides qui combinent un comportement discret représenté par des machines à états (ou des automates finis) avec un comportement continu décrit par des équations différentielles. Dans les systèmes hybrides, les comportements continus sont mesurés par des capteurs. Idéalement, les capteurs ont un accès continu à ces mesures, ce qui correspond à un modèle abstrait de SCPs, appelé *Event-Triggered system* [2]. Cependant, la mise en œuvre de tels modèles est difficile en pratique. Il est donc préférable d'introduire un modèle plus concret, appelé *Time-Triggered system* [2], où les capteurs prennent des mesures périodiques. Platzer et al. [3, 4] utilisent les modèles *Event* et *Time-Triggered* pour concevoir et vérifier des systèmes hybrides. Ils ont prouvé qu'un modèle *Time-Triggered* est un raffinement d'un modèle *Event-Triggered*, en utilisant une extension de la logique dynamique différentielle (d$\mathcal{L}$), appelée logique de raffinement différentiel (dR$\mathcal{L}$).

Introduite par J. Raymond Abrial [5], Event-B est une méthode formelle qui permet de décrire des systèmes discrets en utilisant des événements. Un modèle Event-B se compose de plusieurs composants de type, *Context* et *Machine*. Un contexte peut définir des ensembles abstraits et énumérés, des constantes, des axiomes et des théorèmes. Une machine Event-B spécifie le comportement dynamique du système modélisé. Une machine modélise un système à l'aide de variables d'état et d'événements qui mettent à jour ces variables. Un modèle Event-B s'accompagne d'une série d'obligations de preuve (OPs) visant à vérifier les propriétés de sûreté du système modélisé. Le point fort d'Event-B consiste à utiliser des modèles abstraits pour représenter le comportement abstrait d'un système donné et le raffinement pour introduire des détails et démontrer la conformité entre le modèle abstrait et le modèle concret. Le raffinement d'un modèle formel permet d'enrichir ce modèle pas à pas.

2 Motivations

Le comportement continu des systèmes hybrides est souvent décrit par des équations différentielles ordinaires (ODEs) qui impliquent une fonction inconnue dépendant d'une seule variable d'état. Il existe deux types de méthodes pour résoudre les équations différentielles ordinaires : les méthodes analytiques (symboliques) et les méthodes numériques. Les méthodes analytiques utilisent un ensemble de théorèmes pour obtenir une solution exacte pour une équation différentielle donnée. Par exemple, l'outil SageMath (System for Algebra and Geometry Experimentation) [6] fournit une fonction prédéfinie qui utilise des méthodes analytiques pour trouver des solutions pour les ODEs. Cependant, la plupart des équations différentielles ne peuvent pas être résolues de manière exacte. Il faut donc s'appuyer sur des méthodes numériques pour obtenir des solutions approchées ou utiliser des techniques d'approximation pour obtenir une équation équivalente avec une solution exacte pour laquelle des méthodes analytiques deviennent applicables. Par exemple, des techniques de linéarisation peu-

vent être utilisées pour transformer une équation différentielle non linéaire en une équation différentielle linéaire.

L'interaction entre la partie discrète et la partie continue de systèmes cyber-physiques fait de la vérification des systèmes hybrides un défi. Le développement de techniques et d'outils pour vérifier des systèmes hybrides a attiré l'attention de nombreux chercheurs. Les approches traditionnelles sont basées sur des outils de simulation comme Matlab/Simulink [7] ou Stateflow [8] qui cependant produisent des résultats entachés d'incertitude. C'est pourquoi la conception et la vérification de systèmes hybrides avec des propriétés critiques de sûreté nécessitent l'utilisation de méthodes formelles. Pour cela, plusieurs approches formelles ont été proposées [9, 10, 11, 12, 4, 13, 14]. Ces approches peuvent être regroupées en deux catégories : les approches basées sur le model-checking et les approches basées sur les preuves formelles.

- Les approches basées sur le model-checking utilisent des automates hybrides pour modéliser des systèmes hybrides et des méthodes d'analyse algorithmique pour prouver leurs propriétés de sûreté. Ces approches sont basés sur le calcul de l'ensemble des états atteignables pour les automates hybrides. Ces approches souffrent des problèmes classiques liés à l'explosion de l'espace d'états.

- Les approches basées sur les preuves formelles utilisent la vérification déductive pour prouver les propriétés de sûreté des systèmes hybrides. L'un des points forts de ces approches est qu'elles restent applicables même pour des systèmes de grande taille et de n'importe quel type (linéaire ou non linéaire). Cependant, ces approches nécessitent des efforts importants et une grande expertise lors de la phase de preuve.

La définition d'approches génériques, comme celle présentée dans cette thèse utilisant la méthode formelle Event-B, pour la modélisation et la vérification de systèmes hybrides peut favoriser l'utilisation d'approches basées sur les preuves formelles pour le développement et la vérification de systèmes hybrides. L'utilisation de la méthode Event-B et de sa plateforme Rodin, outil de développement de projets Event-B, permet de vérifier l'exactitude des systèmes hybrides à l'aide des démonstrateurs automatiques et interactifs inclus dans la plateforme. De plus, l'intégration d'un système de calcul formel tel que SageMath avec un démonstrateur de théorème interactif permet de traiter la résolution d'équations différentielles ordinaires lors de la modélisation d'un système hybride.

3 Contribution

Notre objectif, dans le cadre du projet DISCONT [15], est de développer des approches formelles de modélisation et de vérification de systèmes hybrides. Dans ce contexte, nous avons développé une approche générique pour modéliser et prouver les systèmes *Event* et *Time-Triggered* en utilisant la méthode formelle Event-B. Le processus de raffinement de cette méthode permet de gérer la complexité des systèmes. Comme la méthode Event-B ne permet pas la résolution des équations différentielles ordinaires, nous proposons d'interfacer l'outil Rodin avec un solveur d'équations différentielles, SageMath dans notre cas, en utilisant la notion de plug-in. Les principaux apports de cette thèse sont les suivants :

- une approche générique formelle pour modéliser des systèmes cyber-physiques en considérant un nombre quelconque de propriétés de sûreté. Cette approche consiste à définir trois modèles génériques Event-B en commençant par un modèle abstrait de systèmes cyber-physiques puis en utilisant la stratégie de raffinement pour introduire des détails plus concrets. Ces modèles sont vérifiés sous Rodin à l'aide d'un ensemble de théories introduites dans [16]. Cette approche générique modélise et prouve la relation de raffinement entre les systèmes *Event* et *Time-Triggered* en Event-B.

- un ensemble de règles d'instanciation définies pour construire systématiquement le modèle d'un système hybride spécifique. De plus, nous fournissons un ensemble d'invariants génériques qui ont été identifiés à partir des études de cas pour prouver les propriétés de sûreté. Il suffit de les instancier pour prouver un cas d'étude spécifique.

- une extension de l'approche générique pour interfacer Event-B avec le solveur SageMath. Pour cela, un nouveau niveau de raffinement est défini. Il raffine le modèle *Time-Triggered* en introduisant une fonction pour modéliser les appels au solveur. Un outil a été implémenté comme un nouveau plug-in Rodin. Ce plug-in permet d'appeler SageMath pendant la phase de preuve.

- un ensemble d'études de cas pour valider notre approche. Ils ont été choisis de manière à représenter différents types de SCPs : des systèmes hybrides à une ou plusieurs variables continues, une ou plusieurs propriétés de sûreté et un système hybride non linéaire.

3.1 Approche générique pour la modélisation de systèmes hybrides

Pour modéliser un système hybride, notre approche se compose de trois modèles comme illustré par la Figure 1. Le modèle *ContSystem* qui spécifie le comportement continu du système, le modèle *EventTriggered* qui spécifie les interactions entre la partie discrète et la partie continue du système, et le modèle *TimeTriggered* qui spécifie le comportement de la partie discrète du système.

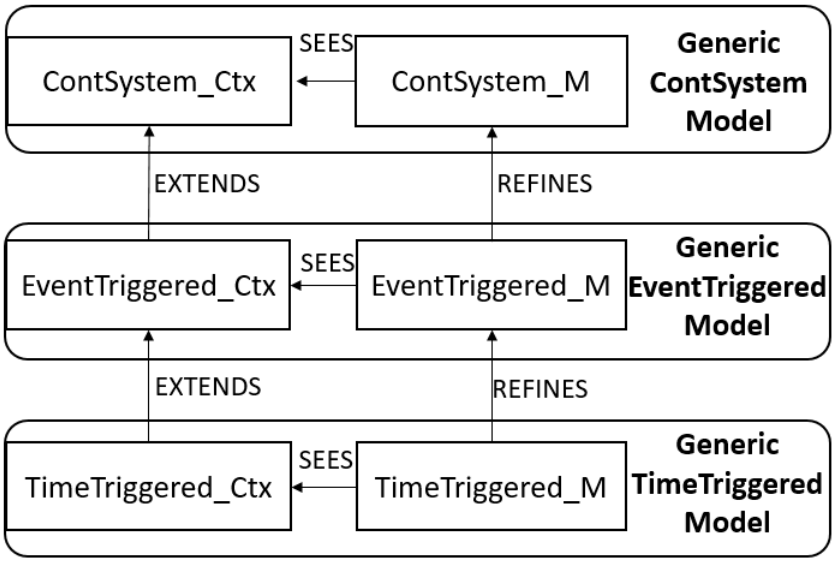

Figure 1: Structure de la spécification générique d'Event-B.

3.1.1 Modèle ContSystem

Le modèle *ContSystem* représente le modèle abstrait de l'approche. Il s'inspire du modèle abstrait de [16] qui vise à modéliser la partie continue des systèmes hybrides en Event-B. Le modèle *ContSystem* est composé du contexte *ContSystem_Ctx* et de la machine *ContSystem_M*. La machine *ContSystem_M* contient deux variables, t et $plantV$, et deux événements, $Progress$ et $Plant$. Les variables t et $plantV$ représentent réspectivement l'évolution continue du temps et des variables d'état. L'événement $Progress$ modélise la progression du temps. L'évolution de la partie continue est modélisée à l'aide de l'événement $Plant$ sur lequel des propriétés de sûreté sont vérifiés au niveau *EventTriggered*.

3.1.2 Modèle EventTriggered

Le modèle générique *EventTriggered* inclut deux composants:

- un contexte nommé *EventTriggered_Ctx* qui introduit l'enveloppe de sûreté du système, représentée par la formule *safe* et calculée à partir de l'exigence de sûreté que le système doit satisfaire.

- une machine nommée *EventTriggered_M* qui introduit le comportement discret du système représenté par la variable contrôlée $ctrlV$.

La sémantique de ce modèle est que la partie physique évolue en parallèle avec le temps et que les deux s'interrompent dès que l'enveloppe de sûreté devient fausse. À ce niveau, on exprime les propriétés de sûreté du système. Pour ce faire, *EventTriggered_Ctx* étend *ContSystem_Ctx* pour représenter ces propriétés. La machine *EventTriggered_M* raffine la machine *ContSystem_M* en introduisant deux nouvelles variables :

- *ctrlV* représente la variable contrôlée. La valeur actuelle de cette variable correspond à l'état actuel du contrôleur.
- *exec* est utilisée pour modéliser l'alternance entre le contrôleur et la partie physique. Par conséquent, *exec* peut prendre deux valeurs *ctrl* et *plant*. Dans Event-B, le temps doit être explicitement géré. Pour être sûr que ce temps explicite progressera entre *ctrl* et *plant*, on ajoute une troisième valeur à *exec*, *prg*, afin de permettre à l'événement *Progress* de s'exécuter. Par conséquent, notre modèle suit la structure suivante : $init; (ctrl; prg; plant)^*$, où *init* représente l'événement *INITIALISATION*.

Pour modéliser l'évolution de la partie physique, on raffine l'événement *Plant* de la machine *ContSystem_M* en remplaçant l'équation différentielle abstraite par celle définie pour une fonction notée *f_evol _plantV*. La fonction *f_evol_plantV* décrit l'évolution de la variable d'état *plantV* en fonction de l'état discret du système. Concernant l'évolution de la partie contrôle, deux nouveaux événements sont ajoutés:

- *Ctrl_normal* représente le mode *normal*. Il se déclenche lorsque c'est le tour du contrôleur à s'exécuter ($exec = ctrl$) et s'il existe une valeur pour laquelle la formule *safe* est vraie. À l'issue de cette exécution, la main est donnée à l'événement *Progress*.
- *Ctrl_evade* représente le mode *evade*. Lorsque l'enveloppe de sûreté $safe$ n'est plus satisfaite le système doit passer au mode evade. Dans ce cas, *Ctrl_evade* affecte une valeur *evade* à la variable de contrôle *ctrlV* et donne également la main à l'événement *Progress*. La valeur *evade* doit être choisie dans l'ensemble des valeurs *evades* du système. Ces valeurs *evades* garantissent que le système satisfera toujours ses propriétés de sûreté.

3.1.3 Modèle TimeTriggered

Le modèle *TimeTriggered* raffine le modèle précédent pour obtenir un système correspondant au modèle *TimeTriggered* de Kopetz. Comme mentionné précédemment, les capteurs d'un modèle *TimeTriggered* prennent des mesures périodiques des variables d'état physique et son contrôleur s'exécute à chaque mise à jour des capteurs. Le modèle *TimeTriggered* est composé du contexte *TimeTriggered_Ctx* et de la machine *TimeTriggered_M*. Le contexte *TimeTriggered_Ctx* étend le contexte *EventTriggered_Ctx* en ajoutant deux constantes :

- *epsilon* : désigne la plus longue durée entre deux mises à jour des capteurs du *TimeTriggered*.
- $safeEpsilon$: garantit que le système reste dans un état sûr pendant les prochaines epsilon unités de temps.

La principale différence entre les modèles *Event* et *TimeTriggered* réside dans la modélisation de la progression du temps. Le plus long laps de temps entre deux mises à jour des capteurs *TimeTriggered* est limité par la durée *epsilon*. Par conséquent, le contrôleur peut s'exécuter au moins chaque *epsilon* unités de temps. Pour cela, on raffine l'événement *Progress* en ajoutant le prédicat ($t' - t \leq epsilon$). Ce prédicat exprime que le temps ne peut pas progresser de plus de *epsilon* unités. Puisque le contrôleur d'un modèle *TimeTriggered* doit faire un choix qui sera sûr jusqu'à *epsilon* temps, nous définissons une nouvelle enveloppe de sûreté nommée *safeEpsilon* dans le contexte *TimeTriggered_Ctx*. Ensuite, dans l'événement *Ctrl_normal_time* qui raffine *Ctrl_normal*, nous ajoutons une contrainte pour s'assurer que *safeEpsilon* est vrai.

3.2 Interfacer Rodin avec SageMath

Pour traiter la résolution des ODEs linéaires dans Event-B, nous proposons d'interfacer l'outil Rodin avec SageMath. L'approche suit le schéma de développement décrit par la Figure 2. Il étend

l'approche générique en ajoutant, par raffinement, un nouveau modèle générique appelé *TimeTriggeredDesolve_M* qui introduit une fonction nommée *B_desolve* pour modéliser des solutions exactes d'équations différentielles ordinaires dans Event-B. Dans le cas de systèmes non linéaires, si l'ODE est linéarisable, on applique le même raffinement en utilisant la fonction *B_desolve* sur la forme linéaire de l'équation. Sinon, nous utilisons la fonction, *B_desolve_rk4()*, qui renvoie une solution approchée.

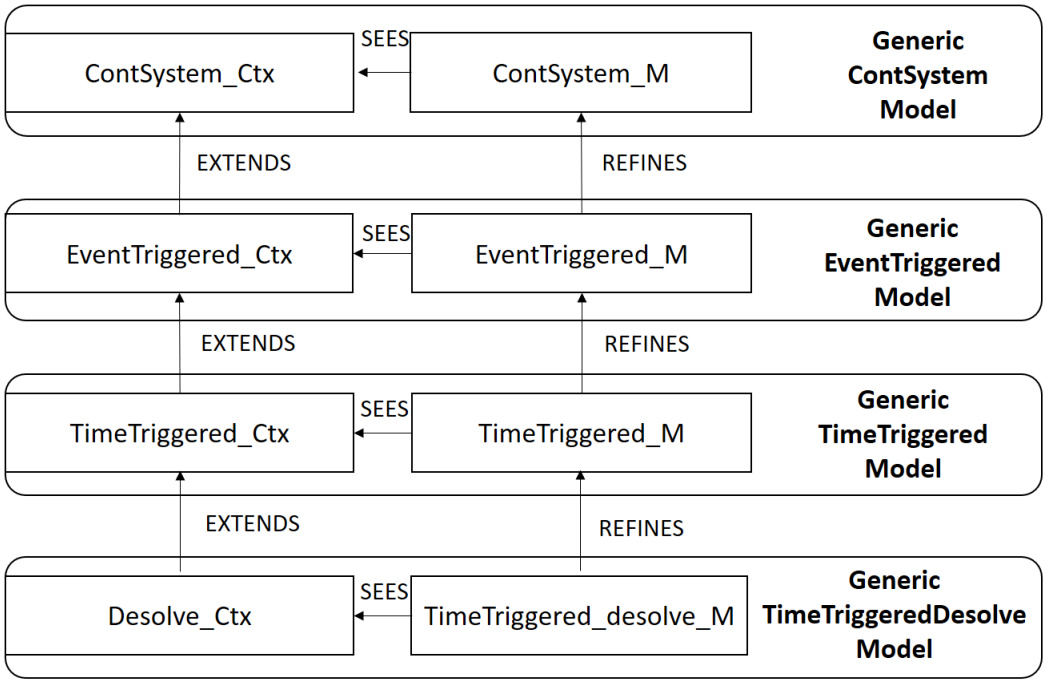

Figure 2: Spécification générique Event-B avec la fonction *B_desolve*.

Le contexte *Desolve_Ctx* étend le contexte *TimeTriggered_Ctx* en introduisant la fonction générique *B_desolve*. L'introduction de cette fonction permet d'établir le lien entre nos modèles Event-B et le solveur d'équations différentielles SageMath. La machine *TimeTriggered_desolve_M* raffine la machine *TimeTriggered_M* en utilisant la fonction *B_desolve* dans l'événement *Plant* pour spécifier la solution générique de l'équation. L'événement *Plant* du modèle *TimeTriggered* est raffiné pour calculer, à l'aide de la fonction *B_desolve*, les nouvelles valeurs de *plantV* depuis sa dernière mise à jour. Afin de mettre en œuvre notre approche, nous avons developé un nouveau plug-in Rodin, appelé *SageMath*, qui interface la plateforme Rodin avec SageMath pour calculer les solutions des ODEs.

3.2.1 Le processus général du plug-in *SageMath*

Pour établir une obligation de preuve contenant les termes *B_desolve* ou *B_ desolve_rk4()*, les étapes suivantes sont nécessaires : (1) appeler SageMath depuis Rodin, (2) résoudre l'équation différentielle et (3) utiliser le résultat renvoyé dans Rodin (voir Figure 3). Pour ce faire, un champ de saisie permettant d'appeler SageMath depuis Rodin apparaît lorsque l'obligation de preuve contient les termes *B_desolve* ou *B_desolve_rk4()*. La deuxième étape consiste à appeler un script prédéfini généré systématiquement à partir des fonctions *B_desolve(...)* ou *B_desolve_rk4()*. La dernière étape consiste à traduire le résultat de SageMath dans le langage spécifique Event-B. Ce résultat est ajouté comme hypothèse pour prouver l'obligation de preuve.

- *Appeler SageMath depuis Rodin (Étape 1)*: pour appeler SageMath depuis Rodin, un bouton appelé *sage* a été ajouté dans la fenêtre de preuve à l'aide d'un plug-in Eclipse. Pour développer un plug-in Rodin, Eclipse fournit un ensemble d'interfaces Java. Ces interfaces sont destinées à être implémentées en fonction de l'objectif du plug-in.

- *Résolution d'ODE dans SageMath (Étape 1' et Étape2)*: un script SageMath est systématiquement généré à partir des fonctions EventB *B_desolve* ou *B_desolve_rk4()*. Dans ce script, l'équation

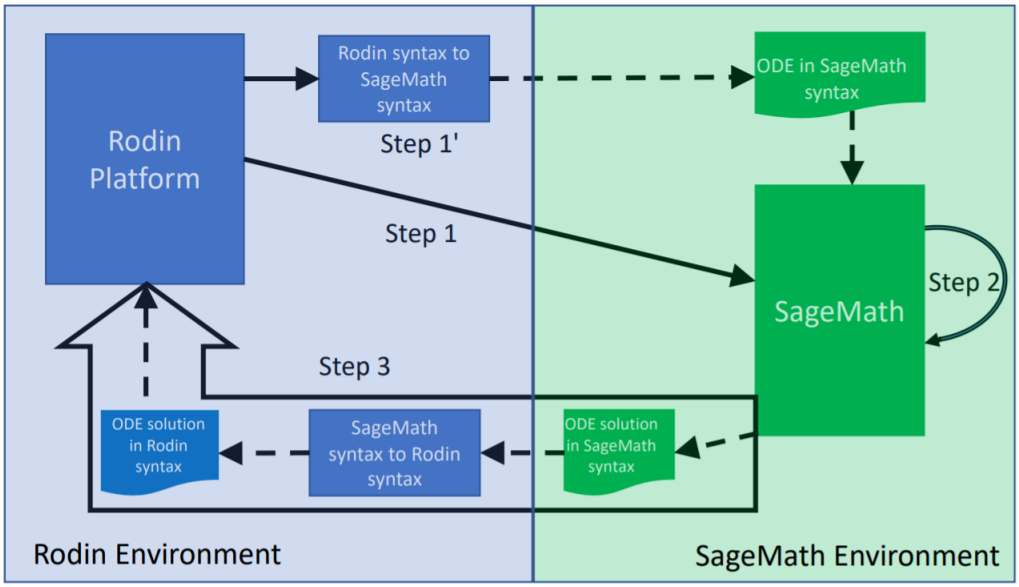

Figure 3: Processus Général.

différentielle doit être exprimée en fonction de la variable contrôlée *ctrlV* qui relie la partie continue et la partie discrète d'un système hybride donné.

- *Utilisation des résultats de SageMath dans Rodin (étape 3)*: à cette étape, la solution renvoyée par SageMath est intégrée comme hypothèse supplémentaire à l'obligation de preuve.

4 Conclusion

Dans cette thèse, nous avons présenté une approche formelle orientée preuve pour modéliser et vérifier des systèmes hybrides à l'aide de la méthode formelle Event-B. L'approche proposée est basée sur la modélisation et la vérification de la relation de raffinement entre les systèmes *Event* et *Time-Triggered* en utilisant Event-B. Le modèle générique *Event-Triggered* décrit l'interaction entre la partie physique et la partie discrète tant dis que le modèle générique *Time-Triggered* introduit la notion de période de contrôle pour représenter le comportement périodique du contrôleur. Nous avons également introduit un niveau plus abstrait, le modèle générique *ContSystem*, qui spécifie les aspects continus des systèmes hybrides.

Comme la méthode Event-B ne permet pas de résoudre les équations différentielles, nous proposons une approche qui permet d'intégrer, dans une spécification Event-B, des appels au solveur d'équations différentielles SageMath. Ceci est rendu possible en implémentant un plug-in à Rodin qui permet d'appeler SageMath.

Jusqu'à présent, nous avons validé notre approche sur des études de cas simples mais représentatives des systèmes hybrides. Dans nos futurs travaux, nous envisageons d'appliquer notre approche à des systèmes plus complexes dont le comportement continu serait décrit avec des équations non linéaires admettant des solutions approchées.

Le plugin SageMath est encore en phase de développement; certaines étapes sont encore manuelles et requièrent l'intervention de l'utilisateur. Par exemple, les scripts SageMath sont exécutés manuellement. Nous envisageons donc d'automatiser ces différentes étapes pour décharger l'utilisateur de toutes ces tâches et rendre l'interaction de Rodin avec SageMAth complètement automatique.

References

[1] Edward A Lee. Cyber-physical systems-are computing foundations adequate. In *Position paper for NSF workshop on cyber-physical systems: research motivation, techniques and roadmap*, volume 2, pages 1–9. Citeseer, 2006.

[2] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, pages 86–101. Springer, 1991.

[3] André Platzer. A Complete Uniform Substitution Calculus for Differential Dynamic Logic. *J. Autom. Reason.*, 59(2):219–265, 2017.

[4] Sarah M Loos and André Platzer. Differential refinement logic. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. IEEE, 2016.

[5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[6] Paul Zimmermann, Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, et al. *Computational mathematics with SageMath*. SIAM, 2018.

[7] Ricardo Sanfelice, David Copp, and Pablo Nanez. A toolbox for simulation of hybrid systems in matlab/simulink: Hybrid equations (hyeq) toolbox. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 101–106, 2013.

[8] Paolo Zuliani, André Platzer, and Edmund M Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 243–252, 2010.

[9] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[10] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *International workshop on hybrid systems: computation and control*, pages 258–273. Springer, 2005.

[11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer, 2011.

[12] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[13] Zhou Chaochen, Wang Ji, and Anders P Ravn. A formal description of hybrid systems. In *International Hybrid Systems Workshop*, pages 511–530. Springer, 1995.

[14] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing Hybrid Systems with Event-B and the Rodin Platform. *Science of Computer Programming*, 94:164–202, 2014.

[15] DISCONT ANR Project. https://discont.loria.fr.

[16] Guillaume Dupont, Yamine Aït-Ameur, Marc Pantel, and Neeraj Kumar Singh. Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 155–170, Cham, 2018. Springer International Publishing.