THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *14/01/2021* **par :**

Guillaume DUPONT

Correct-by-Construction Design of Hybrid Systems Based on Refinement and Proof.

JURY

Dominique Méry	Professeur, LORIA, Université de Lorraine	Président du Jury
Erika Abraham	Professeur, RWTH Aachen University	Examinatrice
Richard Banach	Senior Lecturer, University of Manchester	Examinateur
Jean-Pierre Talpin	Directeur de Recherche, INRIA – Rennes	Rapporteur
Saddek Bensalem	Professeur, Université Grenobles Alpes	Rapporteur
César Muñoz	Directeur de Recherche, NASA Langley Research Center	Rapporteur

École doctorale et spécialité :

MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :

Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeur(s) de Thèse :

Yamine Aït-Ameur, *Marc Pantel* et *Neeraj K. Singh*

Rapporteurs :

Jean-Pierre Talpin, *Saddek Bensalem* et *César A. Muñoz*

Summary

Hybrid systems are a wide category of systems consisting of multiple computers, linked together by some means and interacting with physical devices. Usually cheap, adaptable and incredibly versatile, these systems rapidly found their place in our everyday life, and can be found in basic devices like smart light bulbs, or in more complex structures such as smart factories or autonomous vehicles. In particular, due to their numerous advantages, they can nowadays be found in safety-critical domains, including avionics, self-driving cars or energy distribution.

The complex and hybrid nature of hybrid systems make them quite hard to design in a safe way: while formal methods are mostly able to deal with their discrete parts and control theory with their continuous part, we lack some kind of formalism that would be able to deal with both of these aspects and their integration with one another. The formal modelling of safety-critical hybrid systems is thus a major challenge of the domain.

To overcome this challenge, we propose a generic formal framework, aimed at safely modelling hybrid systems in a correct-by-construction fashion, inherited from the Event-B method the framework is based on.

This framework takes the form of an extensive set of theories that expands Event-B with mathematical features necessary to model continuous behaviours (e.g. continuous functions and differential equations), a generic model that encodes a generic hybrid system, complete with its controller and continuous plant, and a series of patterns, based on refinement, that allow easing the design process.

In particular, we defined three formal patterns, inspired by general practice in hybrid system designs: approximation (substituting an equation system with an approximately equivalent one), centralised control with multiple plants (splitting a plant into a set of plant with a centralised controller) and distributed hybrid systems (system made of components that each consist of a controller and a plant, enforcing a global invariant).

This framework is designed to be extensible: adding a new component is done by describing it as a refinement of the generic model, and possibly to accompany it with relevant theories.

We used our framework on various problems taken from control theory, including computer-assisted cars, water tanks, robots and inverted pendulums.

Résumé

Les système hybrides représentent une large catégorie de systèmes, composés d'une multitude de calculateurs qui communiquent entre eux tout en interagissant avec des systèmes physiques. En général peu coûteux, adaptables et versatiles, ces systèmes ont rapidement su trouver leur place dans notre vie de tous les jours. Il peut s'agir d'objets simples comme des ampoules connectées, mais aussi de structures plus complexes, telles que des usines intelligentes ou des véhicules autonomes. En particulier, et grâce à leur nombreux avantages, on les croise de plus en plus dans des domaines critiques, par exemple dans l'avionique, les véhicules autonomes ou encore la distribution d'énergie.

La nature complexe et hybride de ces systèmes les rend assez difficile à concevoir de manière sûre : les méthodes formelles se sont penchées sur la partie discrète de ces systèmes, tandis que leur partie continue est étudiée par la théorie du contrôle ; mais dans l'ensemble, une approche formelle prenant en charge ces deux aspects simultanément, ainsi que leurs interactions réciproques nous fait défaut. La modélisation formelle de systèmes hybrides critiques est, de ce fait, un défi important du domaine des méthodes formelles.

Afin de répondre à ce défi, nous proposons un cadre générique et formel, avec pour but la modélisation de certains types de systèmes hybrides. Ce cadre, hérité de la méthode Event-B, permet la conception sûre et correcte par construction de ce type de systèmes. Il prend la forme d'un ensemble de théories, qui étendent la méthode Event-B avec les éléments mathématiques indispensables à la modélisation de comportements continus (par ex. : fonctions continues, équations différentielles), un modèle générique qui encode et abstrait les systèmes hybrides, intégrant simultanément leurs contrôleurs et leurs processus continus au même niveau, et une panoplie de patrons, construits sur le raffinement, qui permettent de faciliter la conception de ces systèmes.

En particulier, nous définissons trois patrons formels, naturellement inspirés par les pratiques communes du domaine de la conception de systèmes hybrides : l'approximation (remplacer un système d'équations par un système approximativement équivalant), le contrôle centralisé de plusieurs processus continus (décomposer un processus en plusieurs composantes avec un contrôle centralisé de ces composantes), et la gestion distribuée de systèmes hybrides (un système composé de plusieurs composants, qui sont chacun constitué d'un contrôleur et d'un processus continu, et qui ensemble maintiennent un invariant global).

Ce cadre de conception a été conçu pour être extensible : pour ajouter un nouveau patron, il suffit de le décrire sous la forme d'un raffinement du modèle générique, en l'accompagnant éventuellement de diverses théories, si nécessaire.

Ce cadre formel a été utilisé sur divers exemples empruntés à la théorie du contrôle ou à l'industrie, et notamment dans le domaine de la conduite assistée par ordinateur, des cuves hydrauliques, des robots ou des pendules inversés.

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.

I would like to thank Dominique Méry, for having accepted to preside the jury for this defence, and for his adequate moderation of the debates. I also wish to express my deepest thanks to the examiners, Erika Abraham and Richard Banach, for having accepted to attend this thesis presentation, and for the exceptional discussions their questions sparked during the defence. Also, I am immensely grateful to the referees for this thesis, Jean-Pierre Talpin, Saddek Bensalem and César A. Muñoz, for their thorough reading of this manuscript and for their most positive reviews. Their comments helped polish important aspects of my contributions, and sparked very interesting discussions in many directions around my work.

I would like to express my sincere gratitude towards my advisers, Yamine Aït-Ameur, Neeraj Kumar Singh and Marc Pantel, for their boundless support and availability, but also and above all for their valuable knowledge, hindsight and experience, both in their domain and in the world of research in general. If anything, I wish someday to become a researcher as esteemed and competent as them.

I would like to sincerely thank my colleagues for their support, both moral and scientific. My work greatly gained from their diverse points of view, their questions and ideas, and of course from our numerous passionate and enthusiastic discussions.

My thanks go to Alexander Romanovsky, Paulius Stankaitis and Alexei Illiasov of Newcastle University, and to Fuyuki Ishikawa, Tsutomu Kobayashi and Ichiro Hasuo of the National Institute of Informatics of Tokyo and of the ERATO MMSD Project, for welcoming me. Our meetings sparked interesting collaborations, and contributed to promoting my work on an international scale, and strengthening its reach and validity.

More generally, I wish to give my thanks in general to the formal methods community, and in particular to the members of the DISCONT ANR project for their great help in polishing my work and publishing it. Additionally, special thanks goes to Jean-Raymond Abrial, inventor of the Event-B method among other things, whom I had the great pleasure of meeting on several occasions, and from which my work greatly benefited through our discussions.

Moreover, I would like to deeply thank Richard Banach, most esteemed and brilliant inventor of Hybrid Event-B and retrenchment among other things, whose ideas and reflections were at the very origin of this thesis work, and whose work influenced my progression through the challenges of hybrid systems.

Last, I would like to thank the Institut National Polytechnique of Toulouse and the Institute of Research in Informatics of Toulouse for welcoming me and allowing me to pursue my work, both as a researcher and as a teacher, in more-than-optimal conditions. The experience I got from teaching at ENSEEIHT is priceless, and this work could not have been complete without the material and financial comfort provided by the school and the laboratory.

Special thanks goes to the administrative staff of Toulouse INP, of IRIT and of the École Doctorale MITT for their availability and swiftness when helping me setting up the defence.

On a more personal note, I would like to deeply thank my family and friends for their unwavering support and interest in my activity. I frankly cannot wish to have better moral support in my everyday life.

Contents

List of Figures

List of Listings

Introduction

The idea of delegating a manual task to a device that would be (fairly) automatic has probably emerged at the same time mankind discovered the first complex tools. Traces of documented automatic systems can be found in the work of Hero of Alexandria (c. 10 AD – c. 70 AD), and already feature a complex arrangement of user commands and automatically controlled physical phenomena.

The academic study of such controlled systems emerged in the 19th century with the advent of the industrial revolution. The famous paper by James C. Maxwell, *On Governors* [Max68], is often taken as the first academic resource on control theory. Since then, automatic control with physical means – that is, controlling a physical phenomenon using another physical phenomenon, typically mechanical or electrical – has greatly evolved, always pushing the capabilities of automatic machines further.

Less than a century later, when computers emerged in the scientific landscape, replacing imprecise and costly physical phenomena by digital controllers appeared to be a logical next step. In 1966, the first version of the Apollo Guidance Computer was introduced, a 16-bit, 2048 words RAM digital computer aimed at controlling the Apollo command module that would one day take mankind on the moon.

Nowadays, more than 50 years later, computers have dramatically shrunk in size and cost, and have greatly increased their computation power, allowing them to find a place everywhere, from simple smart light-bulbs to complex avionic systems, from cars to smart factories, from cameras to medical equipment.

Given the omnipresence of such systems and in particular their presence in critical setups (e.g. avionics, autonomous vehicles), it appears absolutely essential to provide a way to reliably establish properties on them, and typically *correctness* (i.e. is the system fulfilling its duty?), or more precisely *safety* (is the system harmful to others?).

Hybrid System This term is used to denote systems that consist of a physical phenomenon that is being controlled by a discrete system, in general a computer [Alu+95]. The term *hybrid* here is of particular importance: such systems integrate, on the same level, features that belong to fundamentally different worlds, with fundamentally different languages, techniques and issues.

On the one hand, controllers are discrete systems, machines that operate step-wise and in a finite or countable setting. They are better described using state-based methods (automata, abstract state machines) and algebra, and logic systems can be used to express and establish properties on them.

Meanwhile on the other hand, the physical phenomena under control are continuous objects: dense, with an ever-evolving state and in an uncountable setting. They are better described using continuous functions and differential equations, and properties may be established using topology and real/complex analysis.

As a result, dealing with these two worlds on the same level and in a rigorous way is a major challenge on the path to certifying hybrid systems.

Scope of the Thesis In this thesis, we are interested in the integration of continuous features in the Event-B formal method in order to be able to formally design hybrid systems with it. The choice of Event-B is quite relevant: first, controllers are typically event-based/reactive systems, for which this method is well-adapted; second, the low mathematical and semantic level of Event-B is such that it can be easily extended, with new constructs and new concepts.

Contributions We propose a generic framework together with a methodology for designing hybrid systems. The framework is based on Event-B. It consists of several tools used to model hybrid systems, and it has been designed to be easily extended. Our main contributions are overviewed below.

Event-B Extension Using Theories The concept of *theories* and the theory plug-in are used to incorporate a number of important concepts related to real numbers, continuous functions and differential equations. This allows the proper modelling of continuous behaviours within Event-B models.

Generic Framework and Methodology An overall framework for designing hybrid systems is proposed, based on an extensible set of components, using Event-B models. The framework is associated with a general methodology, aimed at easing the formal design and verification of hybrid systems [Dup+20a].

Generic Model The foundations of the framework are provided in the form of a generic hybrid system model, to be refined. The various patterns proposed are derived from this generic model, making the framework highly extensible. A number of proofs that would otherwise be recurrent are carried out at this level, and thus do not have to be conducted again [Dup+18b; Dup+18a].

Architecture Patterns The framework provides the possibility to shape a given hybrid system using multiple architectures, often set up in hybrid system design. These are: one controller with one plant (single-to-single), one controller with multiple plants (single-to-many, centralised control) and multiple controllers with multiple plants (many-to-many, distributed control) [Dup+19; Dup+20c].

Approximation Pattern The framework also provides a pattern that formalises approximation, as a refinement step. It allows the support of common operations of control theory and hybrid systems design, such as simplification and in particular linearisation [Dup+20b; Dup+20a].

Organisation of the Manuscript This thesis is organised as follows. Chapter 1 presents a survey of the state of the art, exploring the different types of system modelling and the way they can be verified, using model checking or proof-based methods.

Chapter 2 gives a number of core definitions used throughout the remainder of the thesis. In particular, the method on which our work is based, Event-B is presented: its modelling language, its proof system and its associated semantics. Its extension mechanism (Event-B theories) is also presented in details. Finally, we discuss the advantages of the method as well as its limitations, and use this to identify the key elements enriching the method in order to address the challenge of designing hybrid system with it.

Our contributions are presented in the next chapters. First of all, the proposed framework is laid out in Chapter 3. The motivation for this framework is discussed, and the requirements that led to its design are highlighted, in particular with regard to Event-B and its associated methodology. The chapter also gives an overview of the patterns defined in this framework, and presented more thoroughly in the remainder of the thesis.

Chapter 4, gives the details of the features required by the framework to model hybrid system, and to define the various patterns of the framework. A number of continuous concepts are introduced, which are used to handle continuous behaviours within Event-B, together with various properties to handle these concepts in proofs. The formalisation of these features in Event-B is also discussed.

Then, Chapter 5 describes thoroughly the *generic* model as the foundation of the framework, including the motivation and overall design ideas it relies on. Two well-known case studies are given to illustrate its use, both on a methodological and technical point of view.

Chapter 6 presents a category of formal patterns for modelling different hybrid system *architectures*, and in particular systems consisting of one controller and one plant (*single-to-single*), one centralised controller with multiple plants (*single-to-many*) and multiple controllers with multiple plants (*many-to-many*). A general case study is given as illustration for each of these architectures.

Next, Chapter 7 elaborates on the *approximation* pattern. Approximation is formalised as an operation of the framework, using Event-B's refinement. It allows substituting a system with another which behaviours is approximately equal, while retaining its properties. This pattern is applied to two case studies.

Finally, the conclusion of this manuscript gives a summary of our contributions, and discusses possible outcomes and future work related to our work.

In the context of our work, and to illustrate our contributions, many models have been defined, in Event-B, using Rodin and the theory plug-in. These models are presented throughout this manuscript. Additionally, they are freely available, in their entirety, in Appendices A and B, as well as at the address:
`https://irit.fr/~Guillaume.Dupont/models.php`

Part I

State of the Art

Chapter 1

Designing Safe Hybrid Systems

Hybrid systems are a particular class of systems that incorporate complex, heterogeneous elements related to each others. For this reason, the formal modelling, verification and overall design of hybrid systems are an important challenge that has drawn the attention of many areas of formal methods and computer science in general.

The issues raised by hybrid systems are split in three categories.

First, such systems require specific modelling features, and in particular the possibility to model both discrete and continuous behaviours at the same level in the model.

Second, these particular continuous features need to be handled in the associated verification system: it must thus be able to deal with continuous concepts (functions, differential equations, linear constraints) as well as with discrete features, and must be able to exploit the integration of both world to assert useful properties (reachability, safety, etc.).

Last, hybrid systems come with a number of specific techniques and methods (e.g. composition, approximation, linearisation, discretisation) that need to be addressed as well, both at the model's level and at the proof's level.

In this chapter, we present a survey of the literature addressing formal design and verification of hybrid systems. Formal modelling of hybrid systems is presented in Section 1.1, while Section 1.2 gives an overview of verification techniques for such systems. Section 1.3 investigates various development operations and general methodology for designing hybrid systems.

Last, in Section 1.4, we draw a general idea of what a formal method for designing hybrid system should layout, by outlining a set of requirements, inspired by this survey.

1.1 Modelling Hybrid Systems

The first major challenge in dealing with hybrid systems is the capability to formally model such systems. Indeed, proving or verifying the different properties of hybrid systems require means to model them, including all of their specificity.

The problem is to be able to describe both the discrete and the continuous parts of such systems at the same level, and in a way that allows extracting and verifying useful properties.

To this extent, a multitude of approaches exist. They often rely on a discrete framework capable of supporting the description of continuous behaviours, using continuous functions or differential equations, or a discretised version of these behaviours together with additional assumptions.

1.1.1 Hybrid Automata

Hybrid automata [Alu+95; Hen00] extends the notion of guarded automata with the possibility to handle continuous behaviours. Concretely, a hybrid automaton is a guarded automaton associated with real-valued variables. Every location of the automaton is given a set of *activities* that describe how these real-valued variables evolve, and an *invariant*, that enforces additional properties on the evolving variables (e.g. evolution domain).

The semantics associated with a hybrid automaton can be expressed as a transition system consisting of *discrete* steps (mode changes) and *time* steps (continuous variables changes without mode change).

1.1.1.1 Formal Definition

Formally, a hybrid automaton H is a 6-uplet $H = \langle Loc, Var, Lab, Edg, Act, Inv \rangle$ where:

- Loc is a (finite) set of *locations* or *modes*.
- Var is a (finite) set of real-valued *variables*. A valuation $\nu : Var \to \mathbb{R}$ is a function that associates a real value to a given variable. We denote V the set of valuations.
  A *state* $\sigma = (l, \nu) \in Loc \times V$ is a pair consisting of a location and a valuation. We note Σ the set of states.
- Lab is a (finite) set of *synchronisation labels*, including the special *stutter label*, denoted τ. In the following, we identify synchronisation labels using the colon symbol (e.g. `:synchro`).
- Edg is a (finite) set of *transitions* or edges. Each transition $e = (l, a, \mu, l')$ consists of a *source* location $l \in Loc$, a *target* location $l' \in Loc$, a synchronisation label $a \in Lab$ and a *transition relation* $\mu \in V^2$. This set always contains at least the *stutter transition* $e_{l,\tau} = (l, \tau, Id, l)$, where $Id = \{(\nu, \nu) \mid \nu \in V\}$.
  A transition $e = (l, a, \mu, l')$ is *enabled* if and only if, for a given state (l, ν), there exists some valuation ν' such that $(\nu, \nu') \in \mu$. The state (l', ν') is then called a *transition successor* of state (l, ν).
  Note that, given this definition, for any state (l, ν), the stuttering transition $e_{l,\tau}$ is always enabled, and its transition successor can only be (l, ν) (meaning the stuttering transition does not modify the variables and does not change the current location).
- Act is a labelling function that associates, to each location $l \in Loc$ a set of *activities*. An activity is a function $f \in \mathbb{R}^+ \to V$ that takes the time as input and yields a valuation of the real variables of the automaton. Activities are required to be *time-independent*: if $f \in Act(l)$ is an activity of location l, then $(f + t)$ the function such that $(f + t)(t') = f(t + t')$ (i.e. the function that outputs the same value as f but offsets its argument by t) is also an activity of l, that is $(f + t) \in Act(l)$.
  For any location $l \in Loc$, any activity $f \in Act(l)$ and any variable $x \in Var$, we note $f^x \in \mathbb{R}^+ \to \mathbb{R}$ the function of time that yields the valuation of variable x at any given time; that is, for any $t \in \mathbb{R}^+$, $f^x(t) = f(t)(x)$.
- Inv is a labelling function that assigns, to every location $l \in Loc$ a predicate on the valuation of the variables called *invariant*; that is, $Inv(l) \subseteq V$.

Semantics At any given time, the state of a hybrid automaton is given by a location l and a valuation of its variables ν. Two types of state transitions are defined:

- with a *discrete* step that changes both the current location and the valuation of the variables; this step is instantaneous.
- with a *continuous* time delay that changes only the valuation of the variables, according to the activities of the current location.

The system remains in the same location as long as this location's invariant ($\mathit{Inv}(l)$) holds. In other words, a discrete transition *must* be triggered before the invariant is falsified.

A *run* of the hybrid automaton H is defined as a sequence (finite or not)

$$\rho \ : \ \sigma_0 \mapsto^{t_0}_{f_0} \sigma_1 \mapsto^{t_1}_{f_1} \sigma_2 \mapsto^{t_2}_{f_2} \dots$$

of states $\sigma_i = (l_i, \nu_i) \in \Sigma$, time steps $t_i \in \mathbb{R}^+$, and activities $f_i \in \mathit{Act}(l_i)$, such that, for all $i \geq 0$:

1. the current activity starts with the current valuation as initial condition: $f_i(0) = \nu_i$.
2. the location's invariant remains true: $\forall t, 0 \leq t \leq t_i, f_i(t) \in \mathit{Inv}(l_i)$.
3. the state σ_{i+1} is a transition successor of $\sigma'_i = (l_i, f_i(t_i))$.

The state σ'_i is called a *time successor* of σ_i. The state σ_{i+1} is a *successor* of σ_i. We write $[H]$ the set of runs for the hybrid automaton H.

Based on these previous remarks, for any given hybrid automaton H, a transition system is defined $\mathcal{T}_H = \langle \mathcal{T}, \Sigma \cup \mathbb{R}^+, \rightarrow \rangle$ where the step relation $\rightarrow$ consists of the *transition-step* relations $\rightarrow^a$ (when the system changes location) and the *time-step* relations $\rightarrow^t$ (when time progresses), with $t \geq 0$:

$$\frac{(l, a, \mu, l') \in \mathit{Edg} \qquad (\nu, \nu') \in \mu \qquad \nu \in \mathit{Inv}(l) \qquad \nu' \in \mathit{Inv}(l')}{(l, \nu) \rightarrow^a (l', \nu')}$$

$$\frac{f \in \mathit{Act}(l) \qquad f(0) = \nu \qquad \forall t', 0 \leq t' \leq t, f(t') \in \mathit{Inv}(l)}{(l, \nu) \rightarrow^t (l, f(t))}$$

Note that the stuttering transition ensures $\mathcal{T}_H$ is *reflexive* ($\sigma \rightarrow \sigma$ for any σ).

There is a natural correspondence between the runs of the hybrid automaton H and the paths through the associated transition system $\mathcal{T}_H$: for any states $\sigma, \sigma' \in \Sigma$ with $\sigma = (l, \nu)$ and for any $t \in \mathbb{R}^+$:

$$\exists f \in \mathit{Act}(l), \sigma \mapsto^t_f \sigma' \ \Leftrightarrow \ \exists \sigma'' \in \Sigma, a \in \mathit{Lab}, \sigma \rightarrow^t \sigma'' \rightarrow^a \sigma'$$

Time-Deterministic Hybrid Automata A hybrid automaton H is *time-deterministic* if, for every location $l \in \mathit{Loc}$ and every valuation $\nu \in V$, there is at most one activity $f \in \mathit{Act}(l)$ with $f(0) = \nu$. We then denote this activity $\varphi_l[\nu]$.

For time-deterministic hybrid automata, the time-step relation is simplified: time always progresses by a given amount $t \in \mathbb{R}^+$ from state (l, ν) as long as this is permitted by the location's invariant:

$$\frac{\forall t', 0 \leq t' \leq t, \varphi_l[\nu](t') \in \mathit{Inv}(l)}{(l, \nu) \rightarrow^t (l, \varphi_l[\nu](t))}$$

Linear Hybrid Automata A hybrid automaton H is *linear* if:

1. the predicates for its invariants and for its edge's guard are *convex linear*, i.e. conjunctions of inequations that only involve a linear combination of the system's variable;
2. the activities of each place is defined only by referencing the *derivative* of the system's variables;

As discussed later, linear hybrid automata play an important role in model-checking, as the reachability problem for linear hybrid automata is decidable [Hen+98].

1.1.1.2 Example

As a way to illustrate the definitions given, we present here a simple example taken from [Alu+95].

The objective is to control the temperature of a given room. The room presents a thermometer, able to access the room's temperature, and is hooked up to a heater that can increase the room's temperature. It is assumed that, when the heater is off, the room's temperature decreases naturally.

The goal of the controller is to keep the room's temperature between two constants, denoted m and M ($m \leq M$). We note $\theta \in \mathbb{R}^+ \rightarrow \mathbb{R}$ the room's temperature. The room is characterised by a thermal constant K that represents the speed at which it changes its temperature, and the heater is associated with a heating power, denoted h.

The rules of thermodynamics provide two differential equations for describing the temperature's behaviour:

- when the heater is off, the room's temperature decreases: $\dot{\theta} = -K\theta$
- when the heater is on, the room heats up: $\dot{\theta} = K(h - \theta)$

We model this system (controller + continuous behaviour of the temperature) using a hybrid automaton. This automaton consists of two locations for the two modes of the system (heater on and heater off), and switching to one another depends on the value of the temperature compared to the given bound m and M.

The resulting hybrid automaton is presented in Figure 1.1.

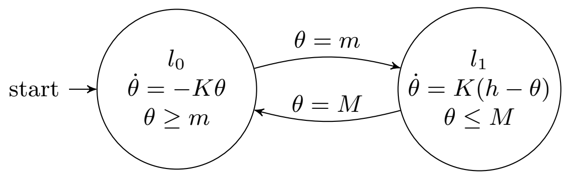


Figure 1.1: Hybrid Automaton for the Thermostat Example

1.1.2 Hybrid Programs

Hybrid programs [Pla08] is a language that allows modelling simultaneously the discrete and continuous features of a hybrid system in the form of a program, inspired by process algebra. A hybrid program expresses discrete features such as choices and repetition, and models continuous behaviours at the same level, using differential equations.

It is associated with a powerful logic system, *differential dynamic logic* to express relevant properties on hybrid programs such as safety or reachability.

1.1.2.1 Formal Definition

Hybrid programs are defined syntactically using a recursive grammar. A hybrid program can consist of:

- a *sequence* $\alpha; \beta$ of two hybrid programs α and β;
- a *repetition* α^* for an arbitrary number of time of the hybrid program α;
- a *non-deterministic choice* $\alpha \cup \beta$, where the hybrid program α or β is executed;
- a *test* $?\chi$, i.e. an instruction that fails if predicate χ is false;
- a *discrete assignment* $x := \theta$ where a given variable (x) is assigned to a real-arithmetic *term* (θ); non-deterministic assignment of a value to a variable is done with the notation $x := *$;
- the *continuous evolution* $x'_1 = f_1(x_1, \ldots, x_n), x'_2 = f_2(x_1, \ldots, x_n), \ldots \; \& \; H$, which set the continuous variables $x_1, \ldots, x_n$ to evolve following the ordinary differential equation system $(f_1, \ldots, f_n)$ while these variables are forced to remain within evolution domain H, characterised by a predicate;

Note that the predicates used in a hybrid program rely on basic predicate logic, extended with real arithmetic (basic real operations plus equality and inequality).

A hybrid system consists of two parts: the controller and the plant. Thus, hybrid programs are written following the pattern

$$(\mathit{ctrl}; \mathit{plant})^*$$

In other words, the hybrid program enacts the controller and then the plant, and repeat this cycle indefinitely (i.e. interleaves the controller and the plant).

Semantics Similar to hybrid automata, the semantics of a hybrid program can be described using transition systems. For a given hybrid program, we denote V the set of its variables. A state of the hybrid program is a map $\nu \in V \to \mathbb{R}$ that associates a value to each variable of the system. We denote Σ the set of states.

Given a term θ (i.e. a real formula possibly involving variables), we denote $[\![\theta]\!]_\nu$ the value of θ at the given state ν.

Given a hybrid program α, the transition relation $\rho(\alpha)$ that specifies which state ω is reachable from state ν through an execution of α is defined as follows:

- $(\nu, \omega) \in \rho(x := \theta)$ iff state ω is identical to state ν except that $\omega(x) = [\![\theta]\!]_\nu$;
- $(\nu, \omega) \in \rho(\alpha; \beta)$ iff there exists a state μ such that $(\nu, \mu) \in \rho(\alpha)$ and $(\mu, \omega) \in \rho(\beta)$;
- $(\nu, \omega) \in \rho(\alpha^*)$ iff there exists a $n \in \mathbb{N}$ and a finite sequence $(\mu_0, \mu_1, \ldots, \mu_n)$ such that $\mu_0 = \nu$, $\mu_n = \omega$, and for all i, $0 \leq i < n$, $(\mu_i, \mu_{i+1}) \in \rho(\alpha)$;
- $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$;
- $(\nu, \omega) \in \rho(?\chi)$ iff χ is true in state ν (i.e. $\nu \models \chi$), and if ω is identical to ν;
- $(\nu, \omega) \in \rho(x'_1 = \theta_1, x'_2 = \theta_2, \ldots, x'_n = \theta_n \; \& \; H)$ iff for some $r \geq 0$, there exist a function $\varphi \in [0, r] \to \Sigma$ such that:

– $\varphi(0) = \nu$ and $\varphi(r) = \omega$,

– at any given time $\tau \in [0, r]$ and for any i, $0 < i \leq n$:

$$\frac{\mathrm{d}[\![x_i]\!]_{\varphi(t)}}{\mathrm{d}t}(\tau) = [\![\theta_i]\!]_{\varphi(\tau)}$$

(i.e. the differential equation for x_i holds),

– at any given time $\tau \in [0, t]$ and for any variable $y \notin \{x_1, \ldots, x_n\}$:

$$[\![y]\!]_{\varphi(\tau)} = [\![y]\!]_{\varphi(0)}$$

(i.e. the other variables are left unchanged),

– at any time $\tau \in [0, r]$, $\varphi(\tau) \models H$
(i.e. the invariant H remains true[1] at any time)

A *run* of the hybrid program α is any sequence of states $(\nu_i)_{i\in\mathbb{N}}$ such that $(\nu_i, \nu_{i+1}) \in \rho(\alpha)$. Such a sequence is also called a *trace*.

Although hybrid programs tend to be monolithic, the recent work of [LP16] introduces a notion of *refinement* for hybrid programs. It allows factorising development and proofs. The work of [Lun+19] also extends the language and the proving process, allowing the definition of modules and their composition, bringing a high level of modularity to the method.

1.1.2.2 Example

We illustrate the use of hybrid programs with an example borrowed from [Que+16]. Note that the following system is also used as a case study for our work, and addressed in Chapter 5.

The problem is the following: a car, characterised by its position p, its speed v and its acceleration a, needs to stop before the given point SP. Our goal is to devise a controller that allows the car to move freely until it brakes to stop.

$$\begin{array}{rcl}
\mathit{sys} & \equiv & (\mathit{ctrl}; \mathit{plant})^* \\
\mathit{safe} & \equiv & x + \frac{v^2}{2B} < SP \\
\mathit{ctrl} & \equiv & (?\mathit{safe}; a := A) \\
 & \cup & (?v = 0; a := 0) \\
 & \cup & (a := -B) \\
\mathit{plant} & \equiv & (x' = v, v' = a \\
 & & \quad \& v \geq 0 \land x + \frac{v^2}{2B} \leq SP) \\
 & \cup & (x' = v, v' = a \\
 & & \quad \& v \geq 0 \land x + \frac{v^2}{2B} \geq SP)
\end{array}$$

Figure 1.2: Hybrid Program for the Stopping-Point Problem

Figure 1.2 proposes a hybrid program to address the case study. The system is encompassed in the *sys* expression. Note that it takes the form discussed in the previous section.

[1]Here $\models$ represent the standard first-order logic interpretation, with the addition of real predicates ($=$, $\leq$, etc.)

The controller is a non deterministic choice where two branches are guarded: if it is safe to do so, the controller causes the car to accelerate (a is assigned to a positive value). Otherwise, if the car's speed is 0, then its acceleration is also 0; this is simply because the car cannot go backward (negative speed) just by braking. Finally, if it is unsafe and the car's speed is not 0, or if none of the other branch has been taken, then the controller issues a braking command to the car (a is assigned to a negative value).

The plant is modelled by the differential equation $\dot{p} = v, \dot{v} = a$ that governs the car's dynamics. It is associated to an evolution domain that characterises the area before the car in which it is still safe to brake (based on the braking distance $\frac{v^2}{2B}$).

To check the correctness of this system, it is required to check that the predicate $p < SP$ always holds, either using model-checking or *differential dynamic logic*, as discussed later.

1.1.3 Event-Based Modelling

Event-based modelling takes the concept of hybrid automata further. It models hybrid systems with a general state, modified by a set of events. In this way of modelling, the state consists of both discrete and continuous variables. The former are handled by the controller, while the latter represents the continuous behaviour of the plant of the hybrid system.

This event-based form of modelling is quite interesting and relevant, as it approaches the way controllers are often programmed. They also focus on hybrid systems from the point of view of the controller, rather than the system as a whole.

1.1.3.1 Continuous Action Systems

A good representative of an event-based (discrete) modelling framework is Back's *Action Systems* [BK89]. As it is already suited for real-time systems, Back extended it to *Continuous Action Systems* [BPP00].

This approach adds to Action Systems a (finite) set of time-dependent variables, associated with a set of possible actions. These time-dependant variables are in essence functions, defined in a piece-wise fashion (note that, for instance "discrete" variables are in fact piece-wise constant). The semantics of Continuous Action Systems is defined in a similar way as "traditional" Action Systems, so discrete and continuous variables are treated in an uniform way.

Mapping of continuous action systems to action systems is not trivial, and has been addressed in [MH06] by assigning *stream*-based semantics to the models – contrasting with the usual *trace*-based semantics – together with data refinement rules, making it more complete and general.

This approach has numerous interests theoretically; in particular, it raises interesting questions on continuous functions and their assignment as well as on refinement of continuous system in general.

1.1.3.2 Event-B

Another typical modelling framework for event-based modelling is Event-B. This method has been used directly to model hybrid systems, by incorporating continuous behaviours straight into an Event-B model.

In [SAZ14], the authors expand on the idea developed for Continuous Actions Systems and adapt it to the Event-B method. Discrete events (i.e. the controller part) are modelled and then, in a refinement, continuous features are added, considering that a discrete system is an abstraction of a continuous one. The model, in particular, uses a variable *now* to model time explicitly, and a *click* event that makes time progress.

Note, however, that these continuous features are only described using real arithmetic, not with differential equations. To work around this limitation though, [SA14] proposes to study the analytical solutions of these differential equations, obtained by the complementary use of Matlab/Simulink, and integrate these solutions into the model.

In [BAB16], the authors use the theory plug-in [BM13] to incorporate new concepts into Event-B, namely reals, continuous and monotonic functions. In this approach, the continuous behaviour is modelled first; modes and then control strategy are later added. Similar to [SAZ14], time is modelled using a variable *now* and an event *click*, and all the difficulty is to ensure "nothing bad" happens in between two executions of *click*, which is why monotonic continuous functions are used. The approach has been used on the case of a set of wind turbines.

This approach lays down the first steps of actually modelling hybrid systems, and shows the problems that occur in doing so. It is sound and usable, but is also very limited.

Typically, time is handled in a discrete manner, like the clock of a controller. While this is closer to the way controllers work in practice, clocks are in fact a low-level concept that does not belong to such a high level of abstraction; different clocks, control and scheduling strategies are to be decided on later refinements, rather than embedded from the first model.

Another problem of this approach is that it requires analytical explicit functions; whereas, in practice, continuous behaviours are defined using differential equations.

1.1.3.3 Hybrid Event-B

The latter approach deals with continuity in a discrete way (time advances step by step). This means the approach is tied to a somewhat low level of abstraction, where the system has been studied and is close to implementation already. This hinders the interest of Event-B, but is necessary given the specificity of this method. To overcome this issue, another way to proceed is to modify slightly Event-B, its language and its semantics, in order to incorporate any required features in the method.

This is the path taken by [Ban+15] with Hybrid Event-B: Event-B is extended by the possibility of defining continuous variables and so-called *pliant* events, which are not instantaneous events, and allows the description of continuous behaviours with continuous functions and differential equations [Ban13].

Time In Hybrid Event-B, time is dense and represented as an interval $\mathcal{T} \subseteq \mathbb{R}^+$, partitioned in a sequence of left-closed right-open smaller intervals ($\mathcal{T} = [t_0, t_1[\cup [t_1, t_2[\cup \ldots$) such that discrete events always occur exactly at t_i, $i \in \mathbb{N}$.

Concretely, time is modelled by a unique read-only variable simply denoted t (defined in the `TIME` clause). Hybrid Event-B also allows the definition of *clocks* (defined in the `CLOCK` clause), which are pliant variables that follow time (i.e. with the same slope, or with a derivative equal to 1). Unlike time, clocks can be discretely set to an arbitrary value (e.g. reset).

Variables Variables can be of two distinct natures: *mode* variables, which are essentially discrete variables and are treated as regular Event-B variables (and so are defined in the standard `VARIABLES` section of the machine); and *pliant* variables, which can evolve both continuously or with discrete

```
MACHINE M_1
TIME t
CLOCK clk
PLIANT x , y
VARIABLES u
INVARIANTS
   inv1 : x ∈ ℝ , y ∈ ℝ
   inv2 : u ∈ ℕ
EVENTS
   INITIALISATION STATUS ordinary
   WHEN
      grd1 : t = 0
   THEN
      act1 : clk := 1
      act2 : x, y := x_0, y_0
      act3 ; u := u_0
   END
```

```
ModeEvent STATUS ordinary
ANY p
WHERE
   grd1 : 𝒢(u, x, y, p, t, clk)
THEN
   act1 : x, y, u, clk :|
      BAP(x, y, u, clk, p, x', y', u', clk')
END

PliantEvent STATUS pliant
INIT ℐ(x, y, u, t, clk)
ANY p
WHERE
   grd1 : 𝒢(u, p)
COMPLY 𝒫(x, y, u, p, t, clk)
SOLVE Dx = Φ(x, y, u, p, t, clk)
         y := E(x, u, p, t, clk)
END
```

Listings 1.1: Hybrid Event-B Syntax Demonstration

steps. They are considered to be continuous from the right and admit a limit from the left at any point.

Their evolution is given, on any time interval $[t_i, t_{i+1}[$, by a differential equation (`SOLVE` clause) with initial condition ($\mathcal{I}$ in the `INIT` clause of the event):

$$Dx = \Phi(x, t)$$

where x is a vector of pliant variables and D is the general time derivative operator.

This equation is well-defined if for any x, $\Phi(x, \cdot)$ is Lipshictz-continuous on $[t_i, t_{i+1}[$ and measurable in t, so that the equation admits a unique solution that is continuous on the interval.

A pliant variable may also be assigned directly (e.g. $x := E$), which allows handling any discontinuity in the variables' evolution, ensuring that they are piece-wise continuous on each interval.

As in Event-B, it is possible to define invariants on the system's variables (`INVARIANTS` section).

Transitions As there is a difference between mode variables and pliant variables, Hybrid Event-B differentiates between *mode* transitions or events, and *pliant* transitions.

In a mode transition (represented by a set of mode events, i.e. with `STATUS` *ordinary*, *convergent* or *anticipated*), a set of variables (mode or pliant) is discretely assigned using a before-after predicate, like it would be in a "normal" Event-B event. Such transitions occur at every t_i and are *timeless*, in the sense that they do not make time progress.

In a pliant transition (represented by a set of pliant events, with `STATUS` *pliant*), pliant variables are given a specific evolution using a differential equation (`SOLVE` clause) with initial conditions ($\mathcal{I}$ in the `INIT` clause of the event). A variable can be given a direct definition, in which case we must ensure the function is (absolutely) continuous, so that it does not interfere with the system's semantics. Unlike mode transitions, pliant transitions are set to occur on every time interval $[t_i, t_{i+1}[$.

Note that the guard of pliant events cannot relate to pliant variables or time/clocks, as it could create discontinuities.

Pliant transitions may define a *complying* predicate ($\mathcal{P}$ in the `COMPLY` clause) or "before-during-after-predicate" that is expected to remain true during the whole transition's duration.

Interleaving The partitioning of time ensures that mode and pliant transitions alternate. More precisely, there is always a mode transition enabled, and this transition enables a pliant transition, which ends up enabling a mode transition again at some point, and so on. Note that this also means that there can never be more than one mode transition occurring "at the same time".

When a pliant transition is running, it is expected to be interrupted by a mode transition (when one becomes feasible) in which case the transition is *preemptive*; but it can also put the system in a state that is incompatible with the current pliant transition (that violates the complying predicate) without any mode transition being enabled, in which case the system terminates. Also, a pliant transition may run indefinitely without any mode transition occurrence.

Semantics Similar to Event-B, Hybrid Event-B models are given a trace-based semantics. From a machine, a collection of traces is derived, which are a sequence (finite or not) of time points $(t_i)_{i \in \mathbb{N}}$ such that a mode transition happens at every t_i, and such that a pliant event is running on every time interval $[t_i, t_{i+1}[$.

The semantics associated with a machine is the set of traces for which invariants (and complying predicates) remain true.

Zeno Property The interval $[t_i, t_{i+1}[$ on which pliant events occur may be arbitrarily small, leading to so-called *Zeno behaviours*, where the system performs an infinite number of jumps in a finite amount of time. To avoid this, there shall exist a constant, δ_{Zeno} such that, for all i, $t_{i+1} - t_i \geq \delta_{Zeno}$.

The approach has been successfully applied to model real-world systems such as pacemakers [Ban+14] or fuel pumps [Ban16].

Note that a similar approach has been studied, by the same author, with abstract state machines [Ban+11; Ban+12].

The major advantage of this method is its high level of expressiveness. Thanks to the way the language is defined, it is possible to describe complex systems, and to prove them using the associated proof system.

However, the approach is not currently tool-supported.

1.1.4 Hybrid CSP, Hybrid Hoare Logic

The work of [Jif94; CJR95] proposes an extension of the Calculus of Sequential Processes (CSP) [Hoa85] to handle hybrid system modelling. It can be used to model real-time and continuous behaviours with message-based communication, thanks to the support of continuous variables and differential equations, times constructs and interruptions.

Moreover, the recent work of [Liu+10b] defines *Hybrid Hoare logic* (HHL), extending the notions of Hoare logic [Hoa69] to be able to express pre-/post-conditions and duration calculus on continuous behaviours. At this level, the notion of *differential invariant*, issued from differential dynamic logic

[Pla15], is introduced to extend the proof system. This way Hybrid CSP is associated to a complete proof system that allows the proving safety properties on models using this formalism.

This approach has had good results on real-world systems, and has been successfully used to model the level 3 of the Chinese Train Control System (CTCS-3) [Liu+10a].

Hybrid CSP inherits all the interesting features of CSP, and in particular refinement. As it involves message-passing, it is also well-suited for studying cyber-physical systems in general.

However, the formalism lies at a mathematically low level and at a high level of abstraction, making it difficult to use and to prove.

1.1.5 Synchronous Languages

When it comes to *reactive systems* (i.e. systems that respond to their environment), synchronous languages allow programming controllers with important timing constraints and bounded resources.

A good example of synchronous language is LUSTRE [Hal+91]: it allows the definition of reactive systems using data-flows, i.e. discrete sequences of values that are calculated on a per-step basis. LUSTRE is well-known in the domain of aeronautics and nuclear energy.

However, it is a fairly low-level language, and is used directly at the (discrete) controller level, meaning it cannot be used to design a hybrid system at higher level.

To overcome this, [BP13] proposed to extend LUSTRE and base it on *non-standard analysis*. The resulting tool, Zelus, is able to express hybrid system directly with their continuous dynamics. Like LUSTRE, it is associated with strong checkers and a very rich typing system to perform verification on it.

1.1.6 Controller Annotation

In the real world, a hybrid system is implemented using an embedded computer running a program, usually written in a low-level language such as C or Ada. From this observation, it appears logical that the way to deal with such a system is to look at the code of its controller and prove its properties, just like any other program.

C programs are analysed and proved using tools such as Frama-C [Cuo+12] and ACSL. The program is *annotated* with properties (preconditions, postconditions, invariants, etc.), and Frama-C is used to extract these annotations and to project them in Why [FM07] or Jessie [Mar07] as to create intermediate models, and ultimately to generate proof obligations.

In [Bol+14], this approach is extended to allow the annotation of C *controller* programs. The plant part of such programs is annotated with continuous elements such as differential equations and continuous constraints, while their discrete part is annotated using discrete features. The resulting information is then extracted with Frama-C and used to generate specific proof obligations.

The work of [Her+12] takes a similar approach using ACSL (ANSI/C Specification Language) annotations, that are extracted to generate proof obligations in the *Prototype Verification System* (PVS) proof assistant [ORS92].

Although this approach gives promising results and is methodologically sound, it is in practice quite hard to deploy. In particular, this method is used *on the already-defined system*, the last step of the design process; and writing annotation on the actual controller, after discretisation and optimisation, which can be cumbersome.

Moreover, the proofs are hard to carry out in general, partly due to the low level of abstraction that this method places itself at.

1.2 Verification of Hybrid Systems

As for any system, the verification of hybrid systems with regard to some requirements is carried by several approaches. Verification is carried out either using model-checking techniques (when possible), or by a set of proofs, both supported by formal verification frameworks.

Compared to other, purely discrete systems, hybrid systems have specific properties that are relevant to study. In particular, *reachability* properties are an important part of hybrid system behaviour: can the hybrid system reach a given state in a set amount of time?

Reachability is at the core of many other properties. For instance, ensuring a system cannot step outside of a given area is in fact equivalent to checking this area is *unreachable*. Similarly, establishing that the given system will eventually visit the given target is also a reachability problem.

1.2.1 Model Checking-Based Techniques

In state-based modelling techniques (such as hybrid automata), hybrid systems are associated with sets of licit traces, where transitions model either discrete changes in the controller or passing of time. Consequently, a way to verify hybrid systems is to explore the traces associated with it, using model checking [CGP99].

Standard model checkers are unable to handle continuous features, making them unfit to check hybrid systems properly. Specialised tools, called *hybrid model checkers* are used to perform this task. In general, hybrid model checkers rely on hybrid automata as their basic models, just like model-checkers for dynamic systems often rely on discrete automata or transition systems.

The capabilities of these hybrid model checkers vary widely depending on the type of hybrid automaton to be handled, its dynamics and the properties of its guards and invariants. Their main goal is to establish the reachability properties of the system (either to show its liveness or its safety if the property is negated).

1.2.1.1 Linear Hybrid Automata

In the case of linear hybrid automata (see Section 1.1.1), establishing reachability is *decidable*.

Tools such as HyTech [AHH96; HHW97] exploit this particularity: it proposes to model hybrid systems with a collection of hybrid automata and to perform symbolic model checking using a specification written in temporal logic. It is able to generate a counter-example trajectory when the specification is violated. This approach is useful to establish safety and reachability. This tool has been successfully applied on systems such as thermostat control and railroad gate controllers in [AHH96; HHW97].

It is possible to relax the linearity constraint a bit and handle *affine hybrid automata*, where the system's dynamics consist of linear formulas on the variables and their derivative. PHAVer [Fre08] is a tool able to exploit this particularity: it allows performing reachability analysis on affine hybrid automata with additional piece-wise constant bounds on the variables' derivative.

1.2.1.2 Approximated Model Checking

The limit between decidable and undecidable is easily reached for hybrid automata [Hen+98]: as soon as the hybrid automaton has non-convex linear constraints, or if its dynamics cannot be bounded by piece-wise linear envelopes, the reachability decision procedure is undecidable. These constraints are very limiting, as it excludes a wide variety of hybrid systems with richer dynamics.

To overcome this limitation, it is possible to relax the type of dynamics used in the hybrid automaton, and to use *approximation* in the decision procedure to maintain its convergence. Of course, the downside of such a technique is that the result is approximated, meaning it comes with a slight error. Fortunately, depending on the system's dynamics and constraints, it is possible to bound this error.

Linear Dynamics For systems that rely on linear ODEs and convex-linear constraints, there are powerful flowpipe calculation algorithms that are both efficient and generates an error that can be easily bounded. Tools such as d/dt [ADM02] and SpaceEx [Fre+11] exploit these kind of algorithms.

Bounded Horizon In order to accept even richer dynamics (i.e. non-linear ODEs), it is possible to make additional assumptions on the system's variables. In Flow* [CÁS13] for example, by giving an explicit bound for the variables and a definite time horizon (i.e. a specific time interval on which to perform the calculation), it is possible to use similar over-approximation techniques, and thus to give out an approximated reachability analysis.

Satisfiability Checking Tools such as iSAT [Frä+07] and iSAT-ODE [Egg+11] take a different approach, based on satisfiability checking and automatic theorem proving. These tools are able to process a boolean combination of non-linear arithmetic constraints involving transcendental functions.

1.2.2 Proof-Based Techniques

Hybrid model-checking suffers from the usual flaws of model-checking in general such as state explosion. It is also limited in terms of the types of dynamics that can be used in the system, and often requires a fair amount of work upstream, to determine the system's constraint or even transform it into a model that can be processed by the model checker.

For complex systems, it can yield valuable counter-examples with regard to reachability, but hardly any definitive positive result, such as invariant non-violation.

For this reason, verifying hybrid system often ends up relying on formal proofs.

1.2.3 Hybrid Programs and Differential Dynamic Logic

The hybrid program formalism (see Section 1.1.2) is associated with a powerful logic system, *differential dynamic logic* [Pla15] (abbreviated d$\mathcal{L}$), to reason on hybrid programs and to establish properties on these programs.

Syntactically, d$\mathcal{L}$boils down to first-order logic with real arithmetic, extended by two modalities ($[\cdot]$ and $\langle\cdot\rangle$) that represent a quantification on the states reachable by the hybrid program:

- $[\alpha]\,\varphi$ means: *for every state reachable by α, φ holds*
- $\langle\alpha\rangle\,\varphi$ means: *there exists a state reachable by α such that φ holds*

The first modality is useful to capture safety requirements, while the second allows encoding reachability properties.

For example, a safety property φ for a hybrid program α with initial conditions I can be established by proving the following sequent:

$$\vdash I \Rightarrow [\alpha]\,\varphi$$

For example, if we take the case study of the stopping car as given in Section 1.1.2.2, we can establish the safety property of the system ($p < SP$) by proving the following sequent:

$$\vdash p < SP \land v = 0 \Rightarrow [sys]\,(p < SP)$$

Note that the initial conditions here are no speed and a position located before SP.

In Platzer's approach, such a sequent is input in the KeYmaera X theorem prover [Ful+15; Que+16]. It is then possible to prove it in a natural deduction way, using standard logic rules in addition with some extra rules relating to hybrid programs and to d$\mathcal{L}$'s modalities.

In particular, d$\mathcal{L}$ is associated to a form of induction on continuous dynamics [PC09a] that allows the inductive proof of so-called *differential invariants*. Formally, given a d$\mathcal{L}$ formula $\psi \rightarrow [\mathcal{D}\,\&\,H]\varphi$ where ψ is a pre-condition, φ is a post-condition, $\mathcal{D}$ is a differential equation with associated evolution domain H, predicate F is differential invariant if:

1. it holds for the pre-condition: $\psi \land H \rightarrow F$
2. if it holds, then it holds after any run of the differential equation: $F \rightarrow [\mathcal{D}\,\&\,H]F$

Also, note that KeYmaera X is able to perform calls to external tools, and in particular to Mathematica [Ful+15], to obtain explicit solutions of differential equations, when available.

The approach has been successfully applied to various examples [Aré+12; Que+16] including autonomous vehicles, train systems [PQ09], flight collision avoidance systems [PC09b], as well as on mobile and surgical robots [Kou+13].

1.2.4 Event-B and Hybrid Event-B

The effort for modelling hybrid systems in Event-B (see Section 1.1.3.2) allows the generation of a number of interesting proof obligations, while benefiting from the inherent specificity of Event-B, and namely induction.

In the case of [BAB16], the authors are able to use the theory plug-in in the proof process to establish several important properties of the system, in the particular the absence of zero-crossing (i.e. nothing bad happens between two time points). This relies on the assumptions that the functions handled are continuous and monotonic between two time points.

For Hybrid Event-B [Ban13; Ban+15], a set of proof obligations pattern is given in [Ban+15] in a way similar to classical Event-B that allows deriving a number of proof obligations from any given Hybrid Event-B model; they then need to be discharged. However, there is no tool to generate these POs, and it involves continuous aspects that the standard Event-B prover cannot handle.

1.2.5 Isabelle/HOL and HHL

The work of [Imm18; IT19] proposes the formal verification of ODE solvers using the Isabelle/HOL [NWP02] theorem prover. The idea is to propose a complete formalisation of the mathematics needed to reason on ODE properties and their resolution, while remaining at the right abstract level. This allows the formalisation and verification of Runge-Kutta-based ODE solvers applicable to a wide variety of ODEs, but also the verification of flow derivative and Poincaré maps computation.

The resulting concepts, proofs and certified algorithms are embedded in the HOL-ODE-Numerics tool. It has been applied to different cases studies [Imm+18; Imm+19].

In addition, Isabelle/HOL has been used to design a proof system of Hybrid Hoare Logic [Zou+14; WZZ15] as presented in Section 1.1.4, which allows the tool to be used to prove Hybrid CSP models [Jif94; CJR95].

1.2.6 Coq, Coquelicot and Annotated Controllers

It is possible, thanks to the work of [Bol+14], to annotate C controller programs with continuous features (see Section 1.1.6). The author goes further by proposing a set of techniques to exploit these annotations. Once extracted and projected in Jessie [Mar07] or Why [FM07], proof obligations are generated, and discharged using Coq [BC04] and/or Gappa [BFM09] to handle issues with floating point arithmetic.

These techniques are supported by an extensive theory of reals, functions and topology, formalised in Coq in a way similar to Isabelle (see Section 1.2.5). This led to the creation of the *Coquelicot* library [BLM15].

1.2.7 Prototype Verification System (PVS) and Annotated Controllers

In the same vein as the Coq-based approach, the work of [Her+12] proposes the verification of Lyapunov-based stability on C-code annotations expressed with Hoare triples using ANSI/C Specification Language (ACSL) annotations. These annotations are mapped to linear algebra concepts formalised in PVS [ORS92]. This translation relies on a linear algebra library, developed in PVS, that allows discharging control-based properties in PVS as proof obligations derived from the Hoare-triples-based annotations of the C code using the theory interpretation mechanism [OS01].

More recently, in the same area, the work of Slagel et al. [SWD21] extended PVS by formalising multivariate polynomials, semi-algebraic sets and real analytic functions for checking real analytic solutions of differential equations, in particular for defining environmental constraints and control properties for developing hybrid systems.

Last, we mention that in the context of the formalisation of complex Unmanned Aircraft Systems (UAS), PVS has been used to develop the DAIDALUS (Detect & Avoid Alerting Logic for Unmanned Systems) [Muñ+15] analysis of UAS behaviours [Muñ+16]. The developed models make extensive use of theories based on real algebra to formalise vectors, coordinate systems (geodesic), distances, etc.

1.3 Development Methodology

In addition to system modelling and verification, one key point in the area of hybrid systems and more generally of cyber-physical systems (CPS) is the possibility to compose and transform systems, and thus being able to build a complex system from a simpler system or a set of simpler systems. This is especially important in CPS, which tend to be systems made of heterogeneous technologies communicating with each other [Lee08].

Several development operations exist at the system level, with the aim of facillitating development or carry out important steps (e.g. discretisation) in a formally verified manner.

1.3.1 Composition and Decomposition

One of the first important tasks of system design is the possibility of composing existing systems into a more complex one.

Composition can be done in a number of ways: merging two systems (variables, states and so on), creating a complex system with communicating subsystems, chaining components, etc. In any case, care shall be taken to ensure each component's behaviour is compatible with the other, and that the specific behaviour of each subsystem allows establishing a global invariant on the whole system.

Hybrid Programs To that extent, the work of [LBT17; Lun+19] extends the hybrid program formalism with two composition operators, making it possible to build a new system from two existing systems. The authors have also extended the proof system in order to be able to decompose the proof of the global system to (simpler) proofs for each component, plus some properties that ensures the operator is used in a correct manner.

Differential refinement logic [LP16] also enables d$\mathcal{L}$ to refine a system and decompose it, in order to ease the proving process and improve the reusability of its components.

Hybrid Event-B In the context of Hybrid Event-B, [Ban+17] proposes the possibility of defining a whole system based on a multitude of components, specified by *interfaces* and connected with one another. It gives a number of assumptions and requirements to determine the correctness of the system, which leads to proof obligations that ought to be discharged.

At the machine level, refinement can be used to decompose a system, which eases the proofs by decomposing them.

1.3.2 Approximation

In practice, real-life physical phenomena are described with differential equations that are generally extremely complicated, without analytical solutions, and on which it is very hard to establish important properties, such as stability or boundedness.

For this reason, it is very often needed to perform an operation called *approximation*. The goal is to find a differential equation that is *simpler*, but which solution is very close to the one of the complex differential equation (up to a maximum error). Establishing properties on the approximated system allows deducing properties on the original one.

This technique is fundamental in control theory, since a lot of systems actually involve equations for which no control can be (easily) found. In the domain of hybrid systems and formal methods, it has also been studied.

In Hybrid Model-Checking For instance, in the work of [GP07; GJP08], the authors use the concept of *approximated (bi-)simulation* [FGP07] and propose a method to model-check non-linear hybrid systems. The principle is to find a linear hybrid automaton that is in approximate (bi-)simulation with the system under study and to model check the former, taking advantage of the convenient properties of linear hybrid automata (i.e. decidability).

Approximate (bi-)simulation is obtained by demonstrating a particular form of approximation between the differential equations of both systems, and by "revising" the guards and invariants of the approximating hybrid automaton to make them stricter. This way, the authors establish that the reachability results obtained on the approximating automaton apply to the original system.

1.3.3 Simulation and Bi-simulation Relation

Section 1.1 outlined that many formalisms for hybrid systems have trace-based semantics. For this reason, it appears natural to investigate the use of simulation and bi-simulation [Mil89] relations, as they are essential in this context.

Informally, a system simulates another if we observe that it "can do the same thing". This operation allows putting in relation systems that are equivalent to observation or specification, meaning they can be substituted while retaining the system's properties.

Hybrid Automata A common operation on hybrid automata is to substitute a system with a simpler one that behaves in the same way, and to perform model-checking on this system. Note that this is similar to approximation, except that the variables of each systems are equal (and not approximately equal).

Techniques have been proposed by [HHW97] to build linear hybrid automata from non-linear ones (when it is possible). For instance, a differential equation with initial conditions of the form $\dot{x} = g(x), x(t_0) = x_0$ where g is integrable, can be simplified by adding a *clock* (i.e. t such that $\dot{t} = 1$). The value of $x(t + \Delta\, t)$ can then be calculated from $x(t)$.

The advantage of these techniques is to provide hybrid automata for which model checking is more efficient, or even decidable in the case of linear hybrid automata.

1.3.4 Refinement

In terms of design, refinement is a powerful operation that allows moving from an abstract system (e.g. a specification) to a more concrete one, while retaining all of its properties. Verification is performed at each step, making it possible to design a system in a *correct-by-construction* way. The advantage of refinement is that proofs are divided and factorised among several models, which makes them usually simpler.

In the context of hybrid systems, refinement allows extending the capabilities of the controller while keeping the exact same controlled continuous phenomena and retaining the system's property. Additionally, when refinement is applied to continuous behaviours, it can be used to alter the physical phenomenon under control: precise it, lift non-determinism, or even approximate it with other behaviours.

Event-B Refinement is at the core of the Event-B method. It allows moving from an abstract system to a more concrete one by adding events or constraining existing features. When using Event-B to handle hybrid systems, continuous and discrete features are modelled at the same level, and refinement affects both these features: it allows precising the behaviour of the discrete controller, but can also constraint the evolution domain or invariant of a continuous variable.

Event-B allows the definition of generic parameters for a model that can be *instanciated* (e.g. substituted) later during refinement. In this setting, refinement serves as the basis of other development operations, and in particular instantiation.

Hybrid Event-B The work of [Ban+15] (relating to Hybrid Event-B) extends Event-B refinement to allow the refinement of continuous behaviours (i.e. *pliant* events). The idea is that the definition of new pliant and mode events partition only one time interval (i.e. only occur on any $[t_i, t_{i+1}[$). This means, in particular, that mode events occurring at t_i and t_{i+1} in the abstract machine also occur at the same time points in the concrete machine.

Hybrid Programs In [LP16], the authors extend differential logic by adding a new operators: a relation on programs that essentially symbolises refinement. More formally, given two hybrid programs α and β, β is said to *refine* α from state ν, and we note $\nu \vdash \beta \leq \alpha$ if and only if, any state reachable by a transition of β starting from ν can also be reached by some transition of α, starting from ν.

This powerful new operator is accompanied by a number of proof rules for composing it with elements of hybrid programs and differential dynamic logic. It allows the decomposition of the proof for a hybrid program, in particular by isolating subsystems, or simplifying parts of the program.

Note that the refinement operator appears at the *proof* level. It can be seen as a general "proving technique" for establishing properties on a given system, with the help of other systems, but it is not really a structural operation applied on hybrid programs.

1.3.5 Numerical Methods, Simulation and Co-simulation

One of the core problems of hybrid systems is to deal with the continuous part, often modelled using complex differential equations. When a high level of certification is not needed, hybrid systems are simply *simulated*, taking advantage of efficient numerical methods for dealing with such differential equations (e.g. Runge-Kutta), even though these methods may introduce a substantial error.

Nonetheless, the possibility to simulate hybrid systems, in addition to verify them, is crucial as to evaluate their behaviour during a run (the same way a discrete system should be tested). For cyber-physical systems made of a set of heterogeneous components, *co-simulation* is especially relevant: each component is simulated *at its level*, and simulations are aggregated and treated together to give a global result.

In addition, the use of numerical or analytical solvers can ease the proof process. [SA14] proposes to integrate the result obtained through Matlab/Simulink in the Event-B model, and to make the proof with these results rather than to handle complex mathematical objects.

Similarly, KeYmaera X proposes a special rule when proving differential logic sequent refers to Mathematica [Ful+15] in order to produce an analytical solution (when possible), i.e. an explicit function, which is then handled in the proof process.

1.3.6 Discretisation

Hybrid system models feature physical phenomena modelled as continuous behaviours that are convenient and easier to handle (from a modelling point of view); but in practice, controllers are computers, i.e. fundamentally discrete systems, that do not support continuous behaviours.

More specifically, when it is implemented, the continuous aspects of the controller are *discretised*, that is, sampled over time and turned into a sequence of events that the controller will be able to provide. This operation is referred to as *discretisation*, and it is a key part of the design process for hybrid systems.

Discretesation raises several problems. In a purely continuous setup, any event is handled "right on time"; but once discretised, if the event occurs between two discrete time points, it can end up being detected slightly to late. This situation is referred to as the *zero-crossing* problem: upon discretisation, the designer shall ensure that the sensing happens at the right time (either slightly earlier or slightly later depending on the situation) as to preserve the system's properties.

This operation often consists in proposing a *clock* (a variable whose evolution follows that of time, i.e. $\dot{t} = 1$), together with a step length ϵ. Once the clock reaches ϵ ($t > \epsilon$), it resets ($t \leftarrow 0$), and this corresponds to one discrete step. The idea is then to demonstrate that "nothing bad" happens on $[0, \epsilon]$.

In such a system, any event is actually bound to the clock; we say the system is *time-triggered* (in contrast with their usual *event-triggered* nature).

This technique is used in hybrid automata, for example to linearise a non-linear hybrid automaton [HHW97]. It is also used in hybrid programs to model discretised systems [Aré+12; Que+16].

In [BAB16] and [SAZ14], the Event-B models proposed by the authors are in fact time-triggered; but no link is made with a possible event-triggered system that would have been developed earlier.

1.4 Discussion

Model checking and its hybrid extensions is a well-known category of formal modelling approaches dealing with hybrid systems; it has the advantage of handling the "hard part" (i.e. the continuous features) in an automated way. However, its main drawbacks are the impossibility to operate on any kind of continuous behaviour, or to sacrifice termination and exhaustive search.

Linear differential equations can be checked in a decidable way, but represent only a mere proportion of real-world systems. However, linearisation may be applied to the system, although this process is far from trivial, and introduces a deviation in the system's behaviour, meaning that results on the linearised system may not be true for the original one.

Proof-based approaches give a higher level of confidence, but are also far more difficult to put into practice, as they require actual proofs on the continuous aspects of the system, which is rarely easy to carry out. They also deal with features that are not included in model-checking, for instance how to model time and continuous behaviour in a usable way.

Overall, our study of the state of the art allowed us to express a number of *requirements* associated with formal methods for designing hybrid system, for handling both modelling and proof parts. These requirements are summarised in the following.

Discrete – Continuous Integration A hybrid system encompasses both discrete and continuous behaviours at the same level. A method for designing such systems *must* be able to handle both at the same level, and to address their integration, i.e. being able to express both behaviours in the model, but also being able to manipulate these behaviours in the verification process.

Extensibility and Theory Import Continuous behaviours come with their theories, languages and properties. They bear a large variety of differences, specific theorems depending on the type of their behaviours, and so on. It is not realistic (or at least not relevant) to build a unique formal theory that encompasses each possible type of continuous behaviours. In particular, this means that 1) the theories behind a formal method for designing hybrid systems must be extensible, so that it can model other needed new types of behaviours, and 2) theories formalising continuous behaviours rely on mathematically sound results, modelled in the formal method and accompanied by well-definedness requirements.

Verification and Scalability Verifying continuous behaviours requires specific techniques that shall be incorporated in the formal verification process. In particular, when the number of different behaviours to handle in the model increases, the verification effort shall remain reasonable, either through the use of specific simplification or trimming techniques (approximation, (bi-)simulation, etc.) or by allowing the designer to decompose the hybrid system (incremental design, architectures, etc.).

Composition and Modularity Hybrid systems and more generally cyber-physical systems are often made of multiple components, interacting with each other with different types of communication/interaction means (e.g. networks, direct connection, etc.). This pushes the idea that a formal method for handling hybrid systems should allow some level of modularity and/or composition. Indeed, it is possible to design several separate systems, establish properties on them and then be able to integrate them together, using their properties to deduce global invariants on the whole system.

Reusability In the same vein as modularity, hybrid systems rely on common types of components and development operations (e.g. approximation, specific architectures, etc.). A formal method may take advantage of the availability of such concepts, and offer the possibility to model parts of the system as reusable components, and propose generic patterns, available for instantiation, to formalise development operations.

Domain specific knowledge is also usually common to entire categories of systems. A formal method should be able to formalise this knowledge and share it as an entity that can be (re)used in several independent models that require this knowledge.

Animation In parallel to modelling and verification, it is useful to propose animation capabilities for hybrid system models. Animation consists of (co-)simulating the various parts of the system together (generally in an interactive way) as to evaluate its behaviour. As consequence, the system can be "tested", to understand its bugs or to explore the limitations of its requirements for example. This is particularly important for hybrid systems, for which only the controller is simulated, and not the higher-level model with continuous features.

Methodology Hybrid systems are associated with specific high-level development operations and overall techniques (approximation, discretisation, etc.) that together constitute a *methodology*. Such a methodology shall be supported by a formal method for hybrid system design, in the form of specific composable formal patterns that support high-level development operations on hybrid systems.

In the context of our work, we aim to propose a formal, generic, reusable and extensible framework, based on the Event-B formal method, for designing and verifying hybrid systems. The purpose of this framework is to address these requirements in a satisfactory and usable way.

Chapter 2

The Event-B Method

Our work relies heavily on the Event-B method [Abr10], a state-based, correct-by-construction formal method for the design of complex systems. This method originated as an evolution of the B method [Abr96], the aim of which is to ensure correctness between the system and its formal specifications.

Event-B is a method: it offers a modelling and mathematical language, as well as a number of principles and rules that allow establishing the conditions under which a given model is coherent and safe with regard to a given invariant (i.e. proof obligations), as well as a mean to prove them.

The method is supported by the Rodin tool[1] [Abr+10], an integrated development environment that enables users to define and handle Event-B models (or *components*), generate the associated proof obligations and allow proving them using automatic and/or interactive provers.

In the context of our work, the choice of Event-B is motivated by the following points:

- it is adapted to the design of *systems* in general; this makes it a solid foundation for the design of hybrid systems;
- it is *state-based* (relying on *events*) that are appropriate for modelling hybrid systems (hybrid automata is another example of a similar modelling approach);
- it revolves around the idea of *incremental design*: a system is built by progressively adding features and performing verification on each step rather than on the entire system, distributing and factorising the proof effort;
- it is *correct by construction*: a system is proved a priori, at the same time as its design, rather than once it has been fully implemented;
- it is *mathematically low level*, relying solely on first-order logic and set theory; this means, in particular, that it can be used to express any concept needed for modelling hybrid systems (e.g. continuous mathematics);
- it is *extensible*, offering the capability to define and refer to customised types, constructors, operators and proof rules;

[1] `http://www.event-b.org/install.html`

In this chapter, we present the Event-B method, as it is the basis of our work. First, Section 2.1 gives an overview of the core principles of the method, while Section 2.2 enumerates the various components it handles. Then, Section 2.3 presents the proof system associated to Event-B and how it is used to establish model correctness. Finally, Section 2.4 details some of the operators of Event-B, used throughout this manuscript, and Section 2.5 presents Event-B's extension mechanism, the *theories*. Finally, Section 2.6 gives a conclusion to the chapter.

2.1 Principles of the Method

Event-B is a formal method that proposes a way to represent systems with specific *models*. It defines what a model is made of, how it can be interpreted (*semantics*) and how to ensure a number of properties on it (e.g. consistency, safety, etc.) using a specific proof system.

In this section, we give a general overview of Event-B as a method, the objects it can handle and the way to handle them.

2.1.1 Modelling

Formally, an Event-B model consists of a *state*, described as a set of variables valued in some arbitrary set. This state is modified by a set of *events* that models the transformations that can be applied to it.

An event may be *guarded* by a predicate on the state. During an "execution" (or *run*) of the model, an event is *enabled* if and only if its guards are true with regard to the current state. Enabled events can be *fired*, in which case the transformation they model is performed on the state.

Events define the transformation applied to the state under the form of a predicate that links the value of the state *before* the event is fired, and its value *after* it has been fired. For this reason, it is called a *before-after predicate* (BAP).

If we denote v the current state, we usually use v' to denote the state *after the event was fired*. The transformation is then written:

$$BAP(v, v')$$

This can be read as: *the state v becomes such that predicate BAP is true.*

An Event-B model always contains a special event, usually called *skip*, that has no guard (i.e. is always enabled) and does nothing (i.e. its BAP is $\top$). This event is important in term of semantics as well as for the refinement operation presented later.

Additionally, every model (provided it has a non-empty state) must provide a special *initialisation* event that informally corresponds to the "first" event ever to be fired. It serves to initialise each variables, and is thus crucial for the global correctness of the method, as it represents the base case in the *induction principle* behind Event-B's proof system (see Section 2.1.4).

Note that the before-after predicate of an initialisation *must* change every variables, and *cannot* reference the previous value of those variables. For this reason, the BAP of an initialisation event is simply written:

$$BAP(v')$$

A model may define an *invariant*, which is essentially a predicate on the state. Informally, the invariant must hold for any model execution; that is, it must hold after the initialisation, and must still hold after the firing of any (enabled) event, and so on. The invariant must be proven (see Section 2.1.4).

2.1.2 Semantics

As often for state-based methods, Event-B is given a *trace-based* semantics [STW14]. Formally, a model is associated to a *transition system*, i.e. generates a set of *traces* (sequences of states) where each state is linked to the next by the firing of an enabled event. The semantics of a model is then defined as the set of traces it generates, where the model's invariant is true at every step.

In this setup, events are considered to be *atomic*: when an event is fired, it is executed *as a whole*, and it corresponds to exactly one transition. Additionally, there can never be more than one event fired at the same time: when multiple events are enabled simultaneously, one is chosen non-deterministically. This is known as *interleaving*.

Note that, in practice, traces associated with a model are infinite, and potentially contains an arbitrary number of *skip* event firing, which are essentially τ-transitions.

2.1.3 Refinement

Refinement is at the core of the Event-B method. The idea of refinement is to provide a formal operation that allows transforming a model into a more precise one by adding details or restraining existing behaviours, while retaining the properties of the original model. It can be seen as a designing step that allows extending an existing machine, either by adding new behaviours or restraining the existing ones, while retaining a number of properties that are true for this machine.

Formally, refinement relies on a *simulation* relation between the transition systems of the two models involved in the refinement relation: given two Event-B models A and B, we say that B refines A if B can do everything A can do (modulo some additional observation mapping).

Adding and Replacing Variables Refinement supports the addition of new variables to a given model, or the replacement of some or all of the existing variables by other variables. For example, it makes it possible to replace the direct access to some measurement (e.g. volume) by a calculation and the access to a derived value (e.g. height).

When replacing abstract variables with new variables, a special invariant, which binds these two sets of variables, need to be given, the *gluing invariant*. This invariant is fundamental as 1) proving its correctness ensures the variables are always correctly linked together and 2) it will appear as hypothesis in refinement-related proof obligations that greatly need it.

Constraining Events An abstract event may be refined with a concrete event, as long as it is stricter (with regard to the abstract variables). In particular:

- if evt^A is refined by evt^C, then evt^C cannot be triggered at a moment evt^A would not be (or, in some way, it cannot be triggered "more often" that evt^A); this corresponds to *guard strengthening*;
- if evt^A modifies variable v, then evt^C has to modify it in a way that is allowed by evt^A; this corresponds to *simulation*;
- if evt^A does not modify variable v, then evt^C cannot modify it (this does not apply to variables introduced by the refining machine); this correspond to *equality*;

Those three properties entail the simulation relation at the core of refinement: establishing them demonstrates simulation (with no further proof needed).

Adding Events As discussed in Section 2.1.1, any machine has an implicit *skip* event that does nothing and is always enabled. It is always possible to refine this event while observing the constraints given in the previous paragraph, and thus to write a *new* event, as long as it does not modify any abstract variable (property of equality since *skip* does not modify any such variable).

2.1.4 Verification

For any model, the method allows calculating *proof obligations* (PO) using substitution calculus, that are predicates that must be proven (or *discharged*). Once every PO is proven, the model is proved to be consistent and safe (with regard to its invariants), by virtue of the Event-B method.

2.1.4.1 Proof System

Event-B is associated with a proof system based on natural deduction to reason on first-order logic with set theory. It consists of several proof rules such as case disjunction, implication elimination (*modus ponens*), cut rules, as well as several rules for quantifier eliminations and for rewriting set-theoretic formulas.

This system allows proving generated proof obligations and thus the consistency of the model, through the discharging of its associated proof obligations.

In addition, the method supports, at the model level, proofs by *induction*. Formally, given a property $\mathcal{P}$, the induction principle states that, if:

1. $\mathcal{P}$ holds at **initialisation**

2. if $\mathcal{P}$ holds for some given state, then $\mathcal{P}$ is true after triggering any event

then $\mathcal{P}$ holds for the entire model, i.e. all states.

Model-level induction makes it possible greatly decompose the verification process: instead of having to be proved on the entire system, the property is decomposed, and a proof obligation is generated for each event. Event-B's proof system is used to discharge these proof obligations and, by virtue of the induction principle, they entail that the property holds on the entire system.

Moreover, refinement allows building a concrete system by adding features one by one, starting with an abstract system. This design operation also allows proof decomposition: once the abstract system is proven, it is sufficient to prove that the added feature does not compromise its consistency. In other word, proofs is done for new features only, and not for the entire system again.

2.1.4.2 Types of Properties

We give here an overview of the various types of POs and other properties handled by Event-B. Section 2.3 gives a more thorough presentation of the proof obligation system (and in particular how they are generated).

Well-Definedness An Event-B model presents different mathematical expressions, written in Event-B's specific expression language. This language is based on the syntax of first-order logic with set theory. Like for many languages, a valid expression does not always makes sense; for this reason, Event-B makes sure any expression written, in addition to being syntactically correct, is *well defined*. A typical example of well-definedness is the arithmetic division a/b: this formula is syntactically correct, but is well defined if, and only if $b \neq 0$.

Consistency In order to be correct, a model must be proven to be *consistent*, i.e. it cannot behave badly or in an unexpected way. In particular, a model must enforce its invariants (i.e. must not violate them). Additionally, since every event is expressed using a before-after predicate, it is important to establish that *any enabled event as a feasible action* or, in other words, that if an event is enabled, there must exist a new state (v') such that its associated before-after predicate ($BAP(v, v')$) is true.

Refinement Correctness As explained in Section 2.1.3, when refining an Event-B model, one must ensure guard strengthening, simulation and equality. Note that, thanks to the induction principle, these three properties must be established for each event rather than for the whole system, and the fact that they are true for each event demonstrates the correctness of refinement.

Convergence It is possible to demonstrate that some events of the model are executed a finite number of times, thanks to the use of a *variant* (similarly to how a while loop is handled in imperative programming). A variant is a variable valued in a lower-bounded set (typically $\mathbb{N}$ or $\mathbb{P}(S)$ for any S) that must decrease (strictly) in convergent events. Using a fix-point property, the variant allows establishing that convergent events occur a finite number of times (until the variant is re-initialised).

Note here that induction is used to demonstrate that the variant is decreasing in every convergent event, and thus that it is globally decreasing in the model.

Additional Properties Invariants can be used to encode additional properties on a machine. For instance, we can explicitly write the disjunction of every guard to model the fact that the system is always progressing (i.e. deadlock-free). One can also write the two-by-two not-and of each guard to ensure the system is *deterministic* as per its events, i.e. at any given time, at most one event is enabled.

2.2 Modelling

We presented, in Section 2.1, an overview of the Event-B method at a fairly abstract level. We saw that Event-B allowed the definition and handling of *models*, that encode the behaviour of systems with variables and events.

In this section, we elaborate on the way models are written, by giving details on Event-B's structural language.

In practice, an Event-B model consists in several *components*: contexts and machines, linked together using various special relations.

2.2.1 Contexts

Contexts (see Listing 2.1) are the first type of Event-B component that encompasses a number of fixed definitions used throughout the models. They allow encapsulating the constant elements and hypotheses (axioms) of the systems, and gathering them in one specific element. They are also used as a form of model *parameterisation*.

```
CONTEXT Ctx
EXTENDS Ctx', ...
SETS S, ...
CONSTANTS c, ...
AXIOMS A, ...
THEOREMS T, ...
END
```

Listings 2.1: Event-B Context

Extension Relation A context may *extends* one or more other contexts. As the name suggests, this basically means the content of the extended contexts are directly accessible from the given context, exactly as if the definitions were simply copied.

This relation can be used to delay some definitions: a context C_1 may define a constant $a \in \mathbb{N}$, and an extending context C_2 may append an axiom like $a = 2$; this is where contexts are especially powerful, effectively allowing model parameterisation.

Carrier Sets Contexts may define abstract sets, also called *carrier sets*. These are basically arbitrary sets of unknown size or structure, that serves as types in Event-B formula. It is to be noted that carrier sets are “top-level” sets or types: a carrier set cannot be a subset of another set.

Constants *Constant* is the second type of element that may be defined in a context. They are essentially literals that can be used throughout the models. They are required to be given a type (using an axiom).

Axioms A context may gather a number of *axioms*, that are essentially predicates considered to be true. Axioms may represent mathematics truths, types, or more generally the models' hypotheses.

Note that Event-B does not check the consistence of axioms, meaning it is possible to define an incoherent context. Model-checking can be used however to try and find out if a context is coherent.

Theorems A *theorem* is essentially like an axiom, but is required to be proved within Event-B. Ideally, a model should have a limited number of axioms together with any number of derived theorems, in order to prevent inconsistency.

2.2.2 Machines

While contexts are the static parts of models, machines represent the actual dynamic part. A machine is defined by a *state* that can be modified by *events*.

A machine is divided into two major parts: the machine's header (Listing 2.2), which defines its state and properties, and the events (Listing 2.3) that define its behaviour.

2.2.2.1 Header

```
MACHINE Mach
REFINES Mach'
SEES Ctx,  ...
VARIABLES v,  ...
INVARIANTS I(v)
THEOREMS T_mach,  ...
VARIANT V(v)
EVENTS ...
END
```

Listings 2.2: Event-B Machine (Header)

Relation to Other Components A given machine can relate to other Event-B components. First it can *see* one or more contexts in order to access the elements defined in them and be able to use their axioms in proofs. Second, a machine may *refine* at most one other machine. Refinement is explored more thoroughly in Section 2.1.3.

Variables As said before, a machine consists of a state that is modified by events. This state is expressed as a set of *variables*. Similar to how constants are defined, variables must be given a type, using an invariant. A variable defined in a machine can be read everywhere; it can be *assigned* in actions (contrary to constants, by definition).

Invariants A machine can define multiple properties that constitute its *invariant*. An invariant is a predicate, generally involving the state (hence the notation $I(v)$), that must remain true in every situation. Basic invariants simply give a type to the given variables, but a machine can define much more complex properties, such as a specific domain for a variable or a particular equation linking some variables together.

When an invariant is written, it needs to be proven. This is explained in depth in Section 2.3.

Theorems *Theorems* are special invariant that derive their correctness from the one of other invariants. In other word, theorems are properties deduced solely on other invariants (and axioms) and do not need to be proven inductively like general invariants do.

Variant A *variant* is a special quantity that allows establishing the termination of a given system with regard to some events. Concretely, a variant is a function of the state that must be strictly decreasing (for a given (partial) order, typically $\leq$ or $\subseteq$) and lower-bounded (e.g. by 0 or $\emptyset$).

Termination proofs are briefly covered in Section 2.3.

2.2.2.2 Events

Events are at the core of Event-B machines. They model the actual behaviour of the system, under the form of transformations (i.e. assignments).

INITIALISATION
REFINES **INITIALISATION**
WITH $w :\ W(v, w)\ ,\ \ \ldots$
THEN $v :\mid BAP(v')$

Evt
REFINES **Evt'**
ANY $p\ ,\ \ \ldots$
WHERE $G(v, p)\ ,\ \ \ldots$
THEOREMS $T_{evt}(v, p)\ ,\ \ \ldots$
WITH $w :\ W(v, w, p, q)\ ,\ \ \ldots$
THEN $v :\mid BAP(v, v', p)$

Listings 2.3: Event-B Machine Event and Initialisation

Event Parameters An event may propose one or several *event parameters*, introduced with the `ANY` clause. Informally, they can be seen as local variables or parameters that are filled in, one way or another. An event parameter has to be given a type (just like constants and variables) using a guard; further constraints allow guaranteeing those parameters to have the correct properties when they are used.

In general, an event parameter is an element that is implicitly defined, and is provided by some external means, which allows the simulation of the behaviour of a parameter passed on to a function call, or of a "return value" taken back from that function.

Formally, they represent an *universal quantification* on events: an event with the clause `ANY p, q, ...` actually model a set of events, for all value of p and q.

Guards We saw in Section 2.1.1 that an Event-B model consisted of guarded events. Guards are syntactically introduced using the `WHERE` clause, followed by a predicate on variables and event parameters ($G(v, p)$).

Actions As discussed in Section 2.1.1, the transformation associated to an event (also called *action* of the event) is given using a before-after predicate. Actions are introduced using the `THEN` clause.

An event may define multiple actions, but a same variable cannot be changed in multiple action simultaneously.

Note that, as a matter of simplification, Event-B provides two additional assignment operators, similar to declaration and assignment operators found in programming languages:

- Point-wise assignment: set the value of variable v to be equal to term a:

  $$v := a \ \equiv\ v :\mid v' = a$$

  This is the standard assignment found in several programming languages (C, Ada, etc.).

- Non-deterministic set membership: set the value of variable v to be some value taken in set S:

  $$v :\in S \ \equiv\ v :\mid v' \in S$$

  This is especially useful when we do not know the exact way a value is assigned, but are sure about its type or subtype (e.g. variable declaration without initialisation in languages such as C).

Theorems Similarly to contexts and machines, an event can also define *theorems*. A theorem is simply a guard that can be deduced from other guards (like a machine theorem is an invariant that can be deduced from other invariants).

Initialisation As discussed in Section 2.1.1, an Event-B machine must define a special event called *initialisation*. This special event sets the variables of the machine in their initial state. It gives the machine's initial state, and serves as the foundation of the inductive reasoning associated with Event-B (see Section 2.3).

Refinement An event may refine another event from a refined machine, tying the behaviour of this event to that of the refined one. Refinement is covered more in depth in Section 2.1.3.

Witnesses Witnesses are predicate that make the link between the variables and parameters of refined machines and the one of the concrete one. Formally, when a variable or a parameter *disappears* (i.e. present in the abstract model but no longer present in the concrete model), a predicate may be defined, to make explicit the relation between those disappearing variables/parameters and the one available in the concrete machine. Note that this can be seen as a "local gluing invariant".

Witnesses are especially useful when establishing *simulation*, as they allow linking the states of the abstract and the concrete machines.

They can also be seen as a way to *instantiate* variables and parameters: an abstract parameter p may be refined by a direct value (e.g. $p = 2$). From this point of view, witnesses allow instantiating *existential proof obligations*.

2.2.3 Example

We illustrate the use of the various elements presented in this section using a practical example.

The goal is to design a controller for a logistic automaton, that is able to grab unsorted, labelled boxes and route them to their target loading bay. The automaton has a dedicated *input* from which it retrieves the boxes, and various *outputs* in which it puts them. The system must 1) process every boxes eventually (convergence) and 2) not lose any box (correctness).

First, we will abstract away the outputs of the automaton and consider there is only one global target. A refinement later introduces a more advanced routing system.

2.2.3.1 Context

```
CONTEXT Automaton_ctx_0
SETS BOXES
AXIOMS
   axm1 : finite(BOXES)
END
```

```
CONTEXT Automaton_ctx_1
SEES Automaton_ctx_0
SETS
   BAYS
END
```

Listings 2.4: Logistic Automaton Contexts

Listing 2.4 gives the contexts for this development. The system handles boxes; we model them using an abstract carrier set named *BOXES*. Note that this set must be finite so that we can establish convergence (`axm1`).

Another set is defined, in an other context extending the first one, that represents *docking bays* (denoted *BAYS*). This set is used in a later refinement.

2.2.3.2 Abstract Machine

```
MACHINE Automaton_0
SEES Automaton_ctx_0
VARIABLES Input , Output , Transit
INVARIANTS
    inv1 : $Input \subseteq BOXES$
    inv2 : $Output \subseteq BOXES$
    inv3 : $Transit \subseteq BOXES$
    inv4 : $Input \cup Transit \cup Output = BOXES$
    inv5 : $Input \cap Output = \emptyset \land Input \cap Transit = \emptyset$
               $\land Transit \cap Output = \emptyset$
    thm1 : $Input = \emptyset \land Transit = \emptyset$
               $\Rightarrow Output = BOXES$
VARIANT $2 \times \mathrm{card}(Input) + \mathrm{card}(Transit)$
END
    INITIALISATION
    THEN
        act1 : $Input := BOXES$
        act2 : $Output, Transit := \emptyset, \emptyset$
    END
```

```
Take convergent
ANY b
WHERE
    grd1 : $b \in Input$
THEN
    act1 : $Input := Input \setminus \{b\}$
    act2 : $Transit := Transit \cup \{b\}$
END

Put convergent
ANY b
WHERE
    grd1 : $b \in Transit$
THEN
    act1 : $Transit := Transit \setminus \{b\}$
    act2 : $Output := Output \cup \{b\}$
END
```

Listings 2.5: Logistic Automaton Abstract Machine

In order to establish convergence and correctness on the system, we first model it in a fairly abstract way. The automaton just has an input and an output; it can *take* a box in the input, and *put* a box in the output, provided it has taken it. We model the input and the output using two simple variables that are subsets of *BOXES*, and we add a third variable that models the “inside” of the automaton. The capability of the machine to take and put boxes from the input and in the output are modelled as events. Note that initially, and for convenience, we consider the input to be filled with the whole set of boxes to be processed (meaning the automaton and the output do not contain any box).

In addition to the variables’ “type”, we also write additional invariants in order to establish the machine’s correctness. In the end, the machine must have processed every boxes, i.e. every elements of *BOXES*. This means that, initially, *Input* will contain the entirety of *BOXES* and, in the end, *Output* will contain it. In the meantime, boxes can be in the input, in the output, or in transit inside the machine, but must follow the rules that 1) there cannot be a box that is at multiple places at once, and 2) a box must be in one of the three locations at any time. These rules are expressed as invariants in the machine (`inv4` and `inv5` respectively).

We add a convenient theorem here, saying that, if the input and the automaton are both empty, then it must be that the output contains every boxes (`thm1`). In conjunction with the convergence property of the machine, this entails that the system does eventually process every box.

Last, to establish convergence, we need to provide a variant for the machine. The variant must decrease strictly for both events, so that we can prove those two events are triggered a finite amount of time (until the variant is reset, but this will not happen in this setting). Since *BOXES* is finite, so are all the variables, so we can use their respective cardinality.

A fairly trivial variant for the system is: $v = 2 \times \text{card}(Input) + \text{card}(Transit)$, that indeed decreases in both *take* and *put*, and is a natural number.

The complete machine is given in Listing 2.5; note that it is correct (and the proofs are fairly trivial).

2.2.3.3 Refinement

```
MACHINE Automaton_1 REFINES
      Automaton_0
SEES Automaton_ctx_1
VARIABLES Input , Bay , Transit
INVARIANTS
   inv6 :  Bay ∈ BOXES ⇸ BAYS
   inv7 :  Output = dom(Bay)
END
   INITIALISATION REFINES
        INITIALISATION
   THEN
       act1 :  Input := BOXES
       act2 :  Transit := ∅
       act3 :  Bay := ∅
   END
```

```
Take convergent REFINES Take
ANY b
WHERE
    grd1 :  b ∈ Input
THEN
    act1 :  Input := Input \ {b}
    act2 :  Transit := Transit ∪ {b}
END

Put convergent REFINES Put
ANY b , o
WHERE
    grd1 :  b ∈ Transit
    grd2 :  o ∈ BAYS
THEN
    act1 :  Transit := Transit \ {b}
    act2 :  Bay := Bay ∪ {b ↦ o}
END
```

Listings 2.6: Logistic Automaton Abstract Machine

Now that we have a correct abstract machine modelling our system, we propose a refinement aimed at introducing some kind of routing system for the boxes (Listing 2.6). The principle of this invariant is to replace *Output* with a fairly more complicated structure, for example a set of *bays*. A bay is a particular place that holds boxes. A box can only be in one bay at a time. At this level, we are not interested in how the boxes are actually routed; it is sufficient to say that they are, one way or the other (using event parameters).

In this setting, the *Output* variable disappears and is replaced by *Bay*, which is a function that associate a bay to each box. The replacement is formalised using a gluing invariant (`inv7`); here, the set of boxes in the output is equal to the set of boxes handled by the *Bay* function, or in other words its *domain*.

Note that, since the abstract machine is correct, if we establish the correctness of refinement, then we can deduce that the machine is correct as well, with no additional proof.

2.3 Verification of Event-B Models

As indicated in Section 2.1.4, Event-B allows calculating *proof obligations* for any model, that must be proven in order to establish that model's consistency and correctness (with regard to a specification).

In this section, we present more thoroughly the proof obligation system: how they can be calculated, what they represent, as well as how to model additional advanced properties.

In the following, we use the following notation (derived from the one used in Section 1.1):

- A denotes the conjunction of every *axioms* (and *theorems*) defined in the current context or the contexts seen by the machine;
- I denotes the machine's *invariant* (and *theorems*), or more generally the conjunction of all of its invariants;
- v denotes the machine's *variables*;
- p denotes the event's *parameters*;
- G denotes the event's *guard(s)*;
- W denotes the event's *witness(es)*; in particular, for any x (variable or parameter), $W(x)$ is the witness restricted to x;
- BAP denotes the event's *before-after predicate*;
- when applicable, we use the superscript A or C to denote a particular object from the abstract (resp. concrete) machine; for instance: I^C is the concrete machine's invariant (note that we do not use superscript for *witnesses* as they only appear in *concrete* machines);

2.3.1 General Proof Obligations

The first category of proof obligations presented relates to the general use of Event-B. In particular, they are here to ensure that what is written is correct: well-formed expression and correct theorems.

2.3.1.1 Well-Definedness

One of the most recurrent POs to be found in models relates to well-definedness (or WD). Event-B always makes sure the expressions and predicates written in the model *makes sense*, or, in other words, are well-defined.

Basically, any place where a predicate can be written may issue a WD-type proof obligation. Such POs are generally of the form:

$$\Gamma \Rightarrow \mathcal{L}(a)$$

Where Γ is a conjunction of hypothesis taken from the model (axioms, invariants, guards, and so on) depending on where the expression is written, and $\mathcal{L}(a)$ is a predicate that models the well-definedness of expression a.

2.3.1.2 Theorems

Another common type of PO arises from the use of *theorems*. Theorems can appear in four places (namely theories, contexts, machine header and events), and they generate a proof obligation that consists basically in proving them. For instance, if we have a set of invariants I_1, I_2, etc. and a theorem T, the associated theorem (or THM) PO will be of the form:

$$A \wedge I_1 \wedge I_2 \wedge \ldots \Rightarrow T$$

And similarly with axioms ($A_1 \wedge A_2 \wedge \ldots$) and guards ($G_1 \wedge G_2 \wedge \ldots$).

Note that theorem proofs are not done inductively, unlike most other proofs. A theorem is deduced directly from other properties (invariants, guards, axioms, etc.).

2.3.2 Machine Consistency

When a machine is written, it is required to prove that it is consistent. A machine is consistent if 1) the actions of the enabled events are *feasible* and 2) if the invariants are never violated by any run.

In other words, the transition system associated with the machine is consistent, and each state reachable abides by the machine's invariant.

2.3.2.1 Feasibility

As mentioned in Section 2.2.2.2, an event contains a set of actions, expressed under the form of a before-after predicate. Since it is a predicate, however, it may not always be satisfiable; in the case where it is unsatisfiable, and if the event is triggered anyway, it results in an unspecified behaviour, and is not allowed in Event-B.

The property that a given before-after predicate is satisfiable, under the assumption that the event to which it is associated is enabled, is called *feasibility* (FIS). It is of the general form:

$$A \wedge I \wedge G \wedge W \Rightarrow \mathcal{F}(a)$$

Where $\mathcal{F}(a)$ denote, the feasibility of expression a.

In the general case of the before after predicate, feasibility $\mathcal{F}(a)$ consists in proving the existence of a value that satisfies the predicate, in other words:

$$A \wedge I \wedge G \wedge W \Rightarrow \exists v' \cdot BAP(v, v', p)$$

In the particular case where the before-after predicate is a point-wise assignment, the feasibility is easily automatically proved. In the case where it is a non-deterministic set membership assignment $v :\in S$, it is sufficient to prove that S is not empty.

Feasibility also appears when writing witnesses: since a witness is a predicate that links abstract and concrete variable/parameters, it is required that they are satisfiable; otherwise the concrete event would be feasible while the abstract one is not. This is known as *witness feasibility* (WFIS), and the general PO has the following form:

$$A \wedge I \wedge G \wedge BAP(v, v', p) \Rightarrow \exists x \cdot W(x)$$

2.3.2.2 Invariants

Invariant proving is the core of establishing a machine's consistency. Invariants are used in a model to denote the system's requirements, for instance safety, correctness and so on.

An invariant proof is of the following form:

$$A \wedge I \wedge G \wedge W \wedge BAP(v, v', p) \Rightarrow I\,[v'/v]$$

In other words, we want to establish that I remains true once it has been substituted ($[\ldots/v]$) with the new value of the variables (v') obtained using the BAP. This basically represents the *inductive step* of an inductive proof.

In the particular case of the initialisation, variables does not have a previous value, and the invariant cannot be considered true already; this is called invariant establishment (or *base case* of the inductive proof) and the PO is of the form:

$$A \wedge W \wedge BAP(v') \Rightarrow I\,[v'/v]$$

Note that the BAP cannot reference the previous value of v, since it does not exist.

2.3.3 Refinement Correctness

Refinement is associated to correctness proof obligations, that need to be proven. These POs follow the discussion of Section 2.1.3.

2.3.3.1 Guard Strengthening

An event evt^C with guard G^C refining an event evt^A with guard G^A must not be enabled when the latter is not enabled. This means that an event that is being refined cannot occur "more often" in the new machine (however it can happen *less* often).

In practice, this basically means that, if G^C is true (i.e. evt^C is enabled) then it must be the case that G^A is true as well (i.e. evt^A is enabled). Formally, this yield POs of the form:

$$A \wedge I^A \wedge I^C \wedge W \wedge G^C \Rightarrow G^A$$

2.3.3.2 Action Simulation

If two events evt^C and evt^A with evt^C refining evt^A both modify the same variable v, then the modification in evt^C must be allowed by evt^A, or in other word the transformation in evt^C is not outside of the range of behaviours described initially by evt^A.

This is called *action simulation* (SIM), and we see how it is closely tied to the general concept of simulation. Its general form is the following:

$$A \wedge I^A \wedge I^C \wedge G^C \wedge W \wedge BAP^C(v, v', p^C) \Rightarrow BAP^A(v, v', p^A)$$

We use v without superscript to represent the fact that these variables are common to both abstract and concrete machines.

In the case where a new event is defined in the concrete machine, this event in fact refines *skip* (see Section 2.1.1). This particular event has before-after predicate $\top$ (i.e. it does not affect the machine's variables).

When refining *skip*, the right-hand side of the implication in the SIM PO is equal to $\top$, meaning simulation is always true for new events.

2.3.3.3 Preserved Variables

In addition to action simulation, variables that are not modified by an abstract event must not be modified by any refining (concrete) event. Concretely: if abstract event evt^A does not modify variable v, and if concrete event evt^C refines evt^A, then it must not modify variable v as well.

Note that, in most cases, v is a variable that disappears during refinement, or that is simply not referenced in the event. However, this PO arises when an event is added that modifies abstract variables; indeed, such an event would refine *skip* (see Section 2.1.1) which, by definition, does not modify any variable.

Other than this specific case, the *preserved variable* proof obligation (EQL) is often proven automatically. Its general form is the following:

$$A \wedge I^A \wedge I^C \wedge W \wedge BAP^C(v, v', p^C) \Rightarrow v' = v$$

2.3.4 Convergence and Variant

Event-B allows the formalisation of specific properties of machines at the execution level, such as the fact that some events are executed a finite number of times. To this extent, it is possible to define a *variant* in the Event-B model, which then allows establishing the *convergence* of certain events. The principle is as follows: a variant is defined for the machine, which is an expression (in general a natural number or a finite set) derived from the machine's state. A set of events is denoted as being *convergent*, which means that they cause the variant to decrease (strictly).

The variant is lower-bounded by construction (by 0 if it is a natural number, or by $\emptyset$ if it is a set), and given it is strictly decreasing in some events, it follows that it will reach this bound at some point (modulo interleaving with fairness hypothesis, irrelevant here).

The variant acts as for "counting" the number of times that convergent events are being executed; once it reaches its fixed point, by definition, it no longer moves, meaning that no more triggers of those events will happen (unless it is reinitialised by another, non-convergent event).

2.3.4.1 Variant Consistency

In the case where the variant is a set (with its associated partial order $\subseteq$), we need to prove that it is *finite*. If it is not, then it can be indefinitely decreasing, meeting its lower bound after an infinite number of steps, which is incompatible with the idea of termination.

The proof obligation (denoted FIN) is the following (with V the considered variant):

$$A \wedge I \Rightarrow \text{finite}(V)$$

Note that finite is a special predicate on sets that is true if and only if its parameter is a finite set.

In the particular case where the variant is a number, it has to be proven that it is a *natural* number, i.e. an integer greater than or equal to 0. If it can be negative, we find ourselves in the same situation as for set variants: we can decrease indefinitely ($\mathbb{Z}$ having no bound) and cannot use the fixed point theorem.

This proof obligation (denoted NAT) has the form:

$$A \wedge I \Rightarrow V \in \mathbb{N}$$

It may also be the case we need to prove this PO for every convergent event, since they may use arithmetic that could render the variant negative. Typically, we hence need to prove:

$$A \wedge I \wedge G \Rightarrow V \in \mathbb{N}$$

This PO means that, *if the event is enabled, then the variant cannot become negative.*

2.3.4.2 Variant Decreasing Monotony

The other main particularity of a variant is that it is strictly decreasing in the particular event (tagged as *convergent*). We have two cases here depending on the type of variant used. For sets, we use the inclusion partial order ($\subseteq$); for natural numbers, we use the classical arithmetic order ($\leq$).

The PO for a set variant is of the form:

$$A \wedge I \wedge G \wedge BAP(v, v', p) \Rightarrow V\,[v'/v] \subset V$$

Informally, this can be understood as: *the value of the variant when replacing variable v by its new value is a strict subset of its former value.*

Likewise, we can give the PO for a natural number variant:

$$A \wedge I \wedge G \wedge BAP(v, v', p) \Rightarrow V\,[v'/v] < V$$

This means that *the value of the variant when replacing variable v by its new value is strictly lower than its former value.*

2.4 Important Operators and Mathematical Concepts

Event-B features an expression language, used to build predicates and expressions used in the models. This language is based on set theory and first order logic, and in particular, as for the set-theoretic part, is built on 3 basic constructs: sets, power sets ($\mathbb{P}(\ldots)$) and Cartesian products ($\ldots \times \ldots$).

From these three constructs, together with additional predicates, it is possible to build higher-level structures needed in the models. To ease the use of such high-level concepts, Event-B proposes a set of particular operators, allowing the handling of relations and functions.

In this section, we present some of these operators, extensively used throughout our work.

2.4.1 Relations

The basic structures we can build in set theory are *relations*. A relation is a subset of a Cartesian product, interpreted as a *mapping* between some elements of two sets.

Definition 1 (Relation)**.** *Let E and F be two sets. A relation r between E and F is a subset of $E \times F$ ($r \subseteq E \times F$). We note $E \leftrightarrow F$ the set of every possible relations between E and F.*

Additionally, we define elements of a relation as *mappings*; they are essentially pairs of the underlying Cartesian product.

Definition 2 (Mapping)**.** *Let $r \in E \leftrightarrow F$ be a relation. Given two elements $x \in E$ and $y \in F$, we say x and y are* ***in relation by*** *r or that x* ***maps to*** *y by/in r if and only if $(x, y) \in r$.*
This can also be written $x\ r\ y$ or $x \mapsto y \in r$

A relation can be seen as a list of mappings between elements of E and F, or in other words as a list of pairs of the Cartesian product $E \times F$. It is possible to retrieve the elements mapped by r to the elements of a particular subset of E using the relational image operation.

Operator 1 (Relational Image)**.** *Let $r \in E \leftrightarrow F$ be a relation. Let $S \subseteq E$. The* ***relational image*** *of S by r is denoted $r[S]$ and defined as so:*

$$r[S] = \{y \in F \mid \exists x, x \in S \wedge (x, y) \in r\}$$

Operators are available to retrieve interesting information about relations, namely which elements are mapped by it in general:

Operator 2 (Domain and Range)**.** *Let $r \in E \leftrightarrow F$ be a relation. The* ***domain*** *of r, denoted $\mathrm{dom}(r)$, is the set of elements of E that are mapped by relation r:*

$$\mathrm{dom}(r) = \{x \in E \mid \exists y \in F, (x, y) \in r\}$$

Likewise, the ***range*** *or* ***image*** *of r, denoted $\mathrm{ran}(r)$, is the set of elements of F that are mapped by relation r:*

$$\mathrm{ran}(r) = \{y \in F \mid \exists x \in E, (x, y) \in r\}$$

Relations can also be processed by operators to elaborate other relations.

Operator 3 (Inverse)**.** *Let $r \in E \leftrightarrow F$ be a relation. The* ***inverse*** *of r, denoted r^{-1}, is a relation in $F \leftrightarrow E$ such that:*

$$r^{-1} = \{(y, x) \in F \times E \mid (x, y) \in r\}$$

Equivalently, we have $(y, x) \in r^{-1}$ if and only if $(x, y) \in r$.

Operator 4 (Backward Composition)**.** *Let $r \in E \leftrightarrow F$ and $s \in F \leftrightarrow G$ two relations. The* ***backward composition*** *of r and s, denoted $s \circ r$ is a relation of $E \leftrightarrow G$ such that:*

$$s \circ r = \{(x, z) \in E \times G \mid \exists y \in F, (x, y) \in r \wedge (y, z) \in s\}$$

Intuitively, backward composition consist in "applying" relation s to every values of relation r.

2.4.2 Functions

Functions are particular cases of relations, with the additional constraint that any value of the domain of a function is mapped to *one unique* value of the co-domain.

Definition 3 (Partial Function)**.** *Let E and F two sets. $f \subseteq E \times F$ is a* ***partial function*** *over E, valued in F, and we note $f \in E \nrightarrow F$ if and only if f is a relation and has the unique mapping property:*

$$\forall x \in E, \forall y_1, y_2 \in F, (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2$$

Functions defined that ways are said to be *partial*, because they are not required to provide a mapping for every point of their domain (meaning there may exist values for which the function is not defined). Following this remark, it is possible to define *total* functions.

Definition 4 (Total Function)**.** *Let $f \in E \rightarrow\!\!\!\!\!+ F$ a partial function and $S \subseteq E$ some set. f is said to be **total** on S, and we note $f \in S \rightarrow F$, if it maps every element of S, i.e.:*

$$\forall x \in S, \exists y \in F, (x, y) \in f$$

Total functions are closer to the mathematical idea of a function (whereas partial functions are in essence very fit to describe programming functions). This is a very strong and restrictive property a function can have, that brings a lot of very useful side effects (but also a little bit of additional proof effort). In our work, most continuous functions are total on a given interval, simply because intervals are compacts, which gives useful properties to the function.

Definition 5 (Functional Image)**.** *Let $f \in E \rightarrow\!\!\!\!\!+ F$ a partial function and $x \in \mathrm{dom}(f)$. The functional image of x by f, denoted $f(x)$, is the unique value mapped to x by f (i.e. $(x, f(x)) \in f$). This can also be seen as the only element of the set $f[\{x\}]$.*

In addition, functions may be composed using various useful operators. Since they are relations, any operator that can be applied to a relation can be applied to a function; but some operators actually do preserve its operands' properties.

Operator 5 (Direct Product)**.** *Let $r \in E \leftrightarrow F$ and $s \in E \leftrightarrow G$ two relations. The **direct product** of r and s, denoted $r \otimes s$ is a relation of $E \leftrightarrow (F \times G)$ such that:*

$$r \otimes s = \{(x, (y, z)) \in E \times (F \times G) \mid (x, y) \in r \land (x, z) \in s\}$$

The direct product is a way to "bind" two relations together and make one with it. When used on functions, it has the property to yield a function itself, which domain is the intersection of the two operands' domains. In the context of our work, this operation is used to build a single function from two separate functions, which allows using them in operators than accept only single functions (this effectively gives the possibility to build function *systems*, or kinds of *vectorial functions*).

Operator 6 (Domain Restriction)**.** *Let $r \in E \leftrightarrow F$ a relation and $S \subseteq E$ some set. The **domain restriction** of r on subset S, denoted $S \lhd r$, is a relation of $E \leftrightarrow F$ with the same mappings as r but restricted to pairs which left element is in S. Formally:*

$$S \lhd r = \{(x, y) \in E \times F \mid x \in S \land (x, y) \in r\}$$

The domain restriction operator is used to "focus" on a specific part of a relation (or a function). This operation preserves the nature of the relation, meaning if r is a (total) function, then $S \lhd r$ is a (total) function as well. This is fundamental in our work since it makes us able to effectively *splice* functions: we can take parts of functions and build a coherent function with them.

2.5 Theories

The expressions found in Event-B models are based on set theory and first order logic. This low mathematical level allows, in theory, building any required mathematical structure (provided it is compatible with ZFC axioms); in practice however, complex structures are difficult to handle and define, mostly because we can only handle sets, functions and relations, and that components offer no form of genericity whatsoever.

To overcome this limitation, an extension has been proposed to Event-B [Abr+09; BM13], under the form of a new type of component called *theories*. This extension has been implemented in the Rodin platform as a plug-in, usually called *theory plug-in*[2].

Severall Event-B theories have already been defined and made available, including theories for lists, sequences, binary trees, Peano arithmetic, reals, fix points, closures, and so on. They form the so-called *standard libraries* of Event-B theories[3].

2.5.1 Theory Description

Formally, a theory is a component that lets us define a set of types, operators and associated properties, either directly or algebraically (using axioms). A theory may define genericity parameters, effectively enabling its definition to be used on any Event-B type.

The global structure of a theory is given in Listing 2.7.

```
THEORY Th
IMPORT Th1, ...
TYPE PARAMETERS $E$, $F$, ...
DATATYPES
   Type1($E$, ...)
   constructors
      cstr1($p_1$: $T_1$, ...)
      ...
OPERATORS
   Op1 <nature> ($p_1$: $T_1$, ...)
      well−definedness $WD(p_1, ...)$
      direct definition $D_1$
   Op2 <nature> ($p'$: Type1(...), ...)
      recursive definition
         case $p'$
            cstr1(...) => $DC_1$
            cstr2(...) => $DC_2$
            ...
```

```
AXIOMATIC DEFINITIONS
   TYPES $A_1$, ...
   OPERATORS
     AOp2 <nature> ($p_1$: $T_1$, ...) : $T_r$
        well−definedness $WD(p_1, ...)$
   AXIOMS $A_1$, ...
THEOREMS $T_1$, ...
PROOF RULES
   Metavariables $m_1 : T_{m_1}$, ...
   Rewrite Rules
      $R_1$: $LHS_1(m_1, \ldots)$
         $PC_1 \Rightarrow RHS_1(m_1, \ldots)$
         $PC_2 \Rightarrow RHS_2(m_1, \ldots)$
      ...
   Inference Rules
      $I_1$: $H_1(m_1, \ldots), \ldots \vdash C_1(m_1, \ldots)$
      ...
END
```

Listings 2.7: Event-B Theory

Type Parameters A theory can define none, one or several *type parameters*, that are essentially genericity parameters. When writing the theory, this allows the access to an abstract arbitrary type, typically for writing operators and properties that are agnostic of the sets they handle.

Upon using the theory, these type parameters are to be substituted or *instantiated* with actual Event-B types (that is, a carrier set, a type from a theory, a power set of a type or a Cartesian product of two types).

Custom Types One of the core features of theories lies in the ability to define special *types*. Theories offer different ways of creating types, and notably *parameterised* types (using type parameters),

[2]http://wiki.event-b.org/index.php/Theory_Plug-in
[3]https://wiki.event-b.org/index.php/Theory_Plug-in#Standard_Library

that are then accompanied by a set of operators and axioms, and in some cases constructors and destructors. Theory types behave like any other Event-B type; in particular, they are non-empty and maximal, and can be composed (using Cartesian product and power set) to build complex Event-B types used in expressions.

Explicit Data Types The first kind of types that can be defined in a theory are *data types.* Such types are complex composite types (made of other types), defined in a way similar to algebraic data structures (in functionnal programming for instance), allowing the definition and handling of sum and product types. Data types may be polymorphic (as they can reference type parameters).

A data type must define one or several *constructors*, special operators that allow building an inhabitant of said type. Those constructors may have parameters making reference to other types (specific or parametric), and those parameters themselves define *destructors*, i.e. operators that can retrieve information from a value of the type.

As in type theory, data types may be used in a number of ways: the multiple constructors are used to define sum types (e.g. enumerations, options, inductive data-types) and the multiple parameters/destructors for each constructor allows for product types (e.g. records). Note that inductive data types must define a base case for well-foundation.

Abstract Types Additionnally, a theory allows defining *abstract types*, that are types without any constructors. In essence, those types are similar to *carrier sets* in Event-B contexts, with the advantage that they can be accompanied by axioms, operators and theorems in the theory.

Operators A theory may define *operators* in two ways: direct (in the first `OPERATORS` section of theory) and algebraic (in the `OPERATORS` subsection of the `AXIOMATIC DEFINITIONS` section).

An operator is identified by a unique name, a *nature* and optional parameters. For algebraic expression operators, a type is required since it cannot be inferred, unlike for direct definition where an explicit expression is given.

Operators may define *well-definedness conditions*, predicates that are the preconditions under which the operators can be used. This is used by Event-B when establishing well-definedness requirements and proof obligations (see Section 2.3.1.1).

Directly defined operators are required to give a substitution to the operator (possibly depending on different cases of constructors if some parameters are data types).

Operator Nature An operator have one of two natures: *predicate* or *expression.* A predicate operator, when used, yields a predicate, that is an object that can be composed using boolean operators ($\wedge$, $\vee$, $\neg$, etc.) and used in axioms, invariants, guards, witnesses, etc. An expression operator, when used, yields a typed object that can be used in formulas, to build more complex expressions such as sets, functions and so on.

Axioms Theories can make available *axioms*, in exactly the same way as contexts. Those axioms generally specify the properties of axiomatic operators, or model particular properties that are true but are not to be proved. Those axioms can then be used in proofs. One of their major strength is that, thanks to type parameters, these predicates are *polymorphic*, meaning they can be used with any Event-B type upon deployment.

Note that the only thing that has to be proven on these axioms is their well-definedness; in particular, consistency of axioms is *not* checked.

Theorems Similarly to contexts, theories can define *theorems* (although they are in their own separate sections). Theorems are predicates that can be used later in proofs. Unlike axioms, they have to be proved in order for the theory to be considered “correct” (modulo consistency).

Proof Rules Finally, theories allows defining specific *proof rules*, or in other words to extend the prover’s capabilities by adding new deduction or transformation rules to it. Proof rules come in two fashions in the theory: *rewrite rules* allow substituting an expression with an equivalent one, and *inference rules* allow deducing a fact based on special hypotheses.

Like for theorems, proof rules must be proved.

2.5.1.1 Example

We show how a theory is defined in practice using a small toy example, that expands on the logistical automaton presented in Section 2.2.3. The goal is to formalise box labels, so that we can specify the routing mechanism introduced in the refinement of the automaton machine. The theory is shown in Listing 2.8.

```
THEORY BoxLabel
TYPE PARAMETERS T ,  S
DATATYPES
  BoxLabel(T ,S)
  constructors
    invalid
    label(labelId: ℕ,  target: T ,  hri: S)
OPERATORS
  isValid  predicate  (l: BoxLabel(T,S))
    recursive definition
      case l
        invalid => ⊥
        label(i,t,h) => ⊤
  SetHRI  expression  (l: BoxLabel(T,S),  newHri: S)
    well−definedness condition isValid(l)
    direct definition label(labelId(l),target(l),newHri)
AXIOMATIC DEFINITIONS
  TYPES BarCode
  OPERATORS
    encode  expression  (l: BoxLabel(T,S))  : BarCode
      well−definedness condition isValid(l)
    decode  expression  (b: BarCode)  : BoxLabel(T,S)
  AXIOMS
    axm1: ∀i,t,h · i ∈ ℕ ∧ t ∈ T ∧ h ∈ S ⇒ isValid(decode(encode(label(i,t,h))))
    axm2: ∀i,t,h · i ∈ ℕ ∧ t ∈ T ∧ h ∈ S ⇒ labelId(decode(encode(label(i,t,h)))) = i
    . . .
THEOREMS
  thm1: ∀i,t,h1,h2 · i ∈ ℕ ∧ t ∈ T ∧ h1 ∈ S ∧ h2 ∈ S ⇒ labelId(SetHRI(label(i,t,h1),h2)) = i
  . . .
```

Listings 2.8: Theory of Boxes Label

The theory defines a data type for handling box labels. Those labels are either *invalid* (unreadable, inexistant, etc.) or consist of an identifier that is a natural number, a target (address or bay) and *human-redable information* (HRI). Note that we do not know what the theory's user may use for targets and HRI, so we abstract their respective types using type parameters.

We define a simple predicate on labels that determine if they are valid or not. This predicate is then used throughout the theory as a *well-definedness condition* for other operators (e.g. for the **SetHRI** mutator, that is well-defined only on valid labels).

We also define an abstract type to model bar codes. We are not interested in the precise form of a bar code, simply that it can be obtained from a label (encoding) and yield a label when read (decoding). We also write important axioms characterising invariant properties of the encode/decode functions.

Note that we could also propose various proof rules, derived from the theorems, to ease the proof process when using the theory.

2.5.2 Theories Interactions

Once defined, a theory interacts with other components in three major ways: first, models (contexts and machines) may use the theories, that is, use the operators and types it defines. Second, a theory may *import* other theories and use their content, like any other component, effectively creating a kind of theory hierarchy. Finally, theories also appear during the proof process, adding new theorems and proof rules.

2.5.2.1 Use in Models

Once made available to Event-B models, the content of a theory (operators and types) can be used in the model, as though they have been "added" to Event-B's expression language, among the other predefined operators (such as finite, dom, partition and so on). If a construct from a theory reference type parameters, type inference will allow determining how those parameters are instantiated. Besides, if an operator has specific well-definedness conditions, the corresponding WD proof obligation will be generated as for any other operator (see Section 2.3.1.1).

Example Once defined, the theory presented in Section 2.5.1.1 can be used in models. For example, we could refine the machine obtained in Section 2.2.3 to provide a more precise model of the routing mechanism.

In this refinement of `Automaton_1`, we implement the routing mechanism using labels and bar codes, as defined in the theory. When it is taken from the input, each box is associated with a bar code, that can be decoded into a label. The automaton associate a label with each box, that allows it to route them appropriately.

As labels can either be valid or invalid, we introduce a new event that specialises `Put` into `PutInvalid` for invalid labels (that puts the corresponding boxes on a special separate bay), and we strengthen `Put` to only accept boxes with a valid label.

POs for this model are as expected (simulation, invariant, guard strengthening, etc.). Note that, as we use the target operator on a label (l) in event `Put`, a well-definedness PO is generated, and we need to ensure that we have **valid**(l). This is easily discharged since this property appears directly in the event's guard.

```
CONTEXT Automaton_ctx_2
SEES Automaton_ctx_1
SETS
   STR — Represent strings
CONSTANTS
   InvalidBay — For invalid boxes
AXIOMS
   axm1: InvalidBay ∈ BAYS
END
```

```
MACHINE Automaton_2 REFINES
     Automaton_1
SEES Automaton_ctx_2
VARIABLES Input, Bay, Labels
INVARIANTS
   inv8:
  Labels ∈ BOXES ⇸ BoxLabel(BAYS, STR)
   inv9: Transit = dom(Labels)
END
   INITIALISATION REFINES
        INITIALISATION
   THEN
      act1: Input := BOXES
      act2: Labels := ∅
      act3: Bay := ∅
   END
```

```
Take convergent
REFINES Take
ANY b, c
WHERE grd1: b ∈ Input
      grd2: c ∈ BarCode
THEN
   act1: Input := Input \ {b}
   act2: Labels := Labels ∪ {b ↦ decode(c)}
END

Put convergent
REFINES Put
ANY b, l
WHERE grd1: b ∈ dom(Labels)
      grd2: l = Labels(b)
      grd3: valid(l)
WITH o: o = target(l)
THEN
   act1: Labels := Labels \ {b ↦ l}
   act2: Bay := Bay ∪ {b ↦ target(l)}
END

PutInvalid convergent
REFINES Put
ANY b
WHERE grd1: b ∈ dom(Labels)
      grd2: ¬valid(Labels(b))
WITH o: o = InvalidBay
THEN
   act1: Labels := Labels \ {b ↦ l}
   act2: Bay := Bay ∪ {b ↦ InvalidBay}
END
```

Listings 2.9: Logistic Automaton Abstract Machine

2.5.2.2 Theory Hierarchy

Thanks to the `IMPORT` clause, theories may import other theories. The content of imported theories is available inside the theory as it would be in any other components (see Section 2.5.2.1). In particular, type inference is used to determine the types that instantiate the imported theories' type parameters; but in this particular context, those instantiating types may also be type parameters (which allows genericity to be preserved overall).

The import mechanism makes it possible to greatly factorise definitions: for instance, since *pre-orders* are very useful on a number of mathematical constructs, it is interesting to define a generic theory of pre-orders, and then import this theory in any theory that defines a construct with a pre-order structure. This is also very convenient for creating advanced theory hierarchies. Typically, in our setting, a theory of *orders* would rely on the theory of pre-orders, and would thus import it to expand on it.

Note that – as for the current state of the theory tool – theories may not define symbols with the same identifiers, even if they do not share the same signature.

2.5.2.3 Use in Proofs

Theories defined and referenced in Event-B models can be used during the proof process in several particular ways.

Operator Substitution Any operator that has a direct definition can be substituted with that definition during proof process. At this point, the operator's theory's type parameters are inferred based on the arguments provided to that operator.

Proof Rules Proof rules are made available in the provers at the same level as Event-B's proof rules. For instance, rewrite rules are made available whenever an expression matches one of their antecedent, and invoking them simply carries out the rewriting. Similarly for inference rules, it is possible to select particular hypotheses, and if they match a inference rule antecedent, said rule will be made available.

Like for operator substitution, type parameters are inferred, and arguments and meta-variables are automatically instantiated.

Axiom and Theorem Instantiation Axioms and theorems defined in a theory can be added as hypotheses of the current sequent using a mechanism called *instantiation*. During this process, an axiom or theorem is chosen from a specific theory, and the possible type parameters of that theory are provided explicitly. The instantiated predicate (with actual types) is then added to the hypotheses, and can then be used in the proof process as usual.

2.5.3 Theory Verification

As for any Event-B component, a theory must be verified to ensure its correctness and consistency (to some extent). The proof obligations generated for a theory are of two categories.

Well-Definedness Theories essentially contain Event-B formulas, that have to be well-defined. Like for other components, the use of operators with well-definedness conditions (either from Event-B or from theories) yields well-definedness proof obligations.

Theorem and Inference As mentioned in Section 2.5.1, theorems and proof rules of the theory must be proven, using the axioms defined in the theory, other theorems (defined before) or concepts defined in imported theories.

Axiom Compatibility It is to be noted that axiom compatibility is *not* checked: it is possible to define contradicting axioms, resulting in an inconsistent theory. This is also the case for context, although tools such as ProB allows model-checking those components' consistency, unlike for theories.

2.5.4 Overall Design Ideas

Event-B theories offer a very high level of expressiveness, due to the different types of objects to be defined. The fact that they can be based on each other also allows a good level of progressive construction, as well as a high level of reusability.

But this expressiveness is double-sided, because it also means that a theory can be written in a number of different ways, seemingly and theoretically equivalent, but far from being equal with regard to ergonomics, prover-frendliness and genericity.

2.5.4.1 Algebraic vs. Explicit Definitions

Because theories can simultaneously define axioms and constructs (i.e. objects that are built upon other objects rather than algebraically), an important decision to make when writing a theory is to establish what will be given as axioms and what will be built from pre-existing objects. Defining an operator axiomatically is sometimes easier, but writing relevant constraints on it might be difficult; operators built from other constructs benefit from the theorems and properties already written for these constructs, but a direct definition for the concepts we want to express can be complex and impractical to handle.

These conditions need to be considered when designing a theory. In our case, a good rule of thumb was to express the core building blocks of each theory in an algebraic manner (i.e. using axioms), and build as much operators as possible explicitly, using these core concepts.

Likewise, it is generally a good idea to have as few axioms as possible, and to try to prove theorems from existing axioms, simply because the more axioms a theory defines, the higher the probability is to have incoherent axioms. This is generally possible for simple theories; but when theories get complicated and handle high-level concepts (possibly simplified/approximated), this often becomes cumbersome.

Typically, theorems that involve complex mathematical constructs are better expressed as axioms. The difficulty here is to transcribe a theorem in Event-B; but if this transcription is correct and if the objects handled by the theorem are well-defined, then the theorem can be trusted. Note that the theorem could be formally proven in another tool (such as Coq, Isabelle, Dedukti [Sai13], etc.); only the result itself is relevant to our method.

2.5.4.2 Proof Activity and Prover Ergonomics

The early state of the theory plug-in – as of 2020 – makes it a bit cumbersome to use sometimes. A variety of bugs exist; and although they do not compromise the correctness of what is being written, they hinder its usability, especially in the prover.

The way Event-B and theories work makes it generate numerous of trivial subgoals for each proof obligation, which *should* be automatically discharged by the prover, but in practice they are not. The problem with this is that using a theorem or an axiom from a theory is cumbersome, and requires to manually instantiate it and its generic types – that is, theorems and axioms are not directly usable in the prover, like other hypothesis coming from the model.

A good way to work around this is to define *proof rules.* Indeed, the theory plug-in lets us define inference and rewrite rules, that may be used to handle objects defined is those theories in the prover.

In the current state of the theory plug-in, these rules cannot be applied automatically by the prover; but the mere fact that they can be applied manually without requiring explicit instantiation of the theory makes the theories much more usable.

In general, it is a good idea to create a rewriting or inference rule for any "simple" theorem or axiom defined in the theory (something with few variables and few antecedents). That being said, this technique should not be abused as it could hinder the prover's overall ergonomics.

In every case, it is generally a good idea to write a set of rules dedicated to *type* the custom operators defined in a theory; indeed, whenever the operator is used, it is difficult for the prover to determine the type of the parameters and the return value. Simply rules that clearly give the type of an operator can thus greatly improve the usability of the theory, and help overcome a lot of small subgoals, which are simple to discharge but hindering the workflow.

2.6 Conclusion

The Event-B method is a solid foundation for designing state-based systems. From a design point of view, it is a powerful method, in particular thanks to refinement. On the formal side, it has the advantages of using relatively low-level and abstract mathematics, and thus to be able to express more advanced concepts. It also has simple yet rich and expressive semantics, extensible thanks to the use of theories. Finally, it is tool-supported, allowing the method to be used on real-world systems.

However, core Event-B does not support the modelling of hybrid systems in an efficient and useful way. Indeed:

- it is essentially *discrete*: the systems handled by Event-B correspond to discrete traces, with no notion of continuity between states;
- it does not support *continuity*: this concept is absent from "native" Event-B, although it can be formalised using set theory and first-order logic; in particular, continuous behaviours cannot be expressed in core Event-B;
- it barely supports *reals*: a theory of reals has been written to handle reals in Event-B models, but it is somewhat summary, and does not go beyond basic operators (e.g. no intervals, no continuous functions, etc.);

To summarise, although Event-B perfectly fits formalising discrete behaviours (such as the discrete part of hybrid systems), it lacks a way to express continuous behaviours, together with their interactions with the discrete behaviours.

The objective of this thesis is to bridge the gap between Event-B and modelling hybrid systems, through the definition and use of new theories, as to create a generic *formal framework* for correct-by-construction hybrid system design, that is sound and tool-supported.

Part II

Contributions

Chapter 3

Framework for Hybrid System Design

The overall goal of the thesis is to provide a framework for designing safe hybrid systems. The proposed framework is generic and extensible, and takes the form of a method and a set of tools that engineers can apply and use when developing such systems. It relies on Event-B, extended to address the requirements of a formal method for designing hybrid systems as proposed in Section 1.4.

In this chapter, Section 3.1 briefly presents the extensions to Event-B that we proposed in order to lay the foundations of our framework. Section 3.2 presents the framework itself, and Section 3.3 describe its components, as well as the way to use it. Finally, Section 3.4 defines the methodology associated with the framework and its use, and Section 3.5 concludes the chapter with a discussion on the framework. Details of the methods and tools shown here are presented in the following chapters.

3.1 Requirements for Modelling Hybrid Systems in Event-B

Event-B, in its core, lacks several important features to be able to model hybrid systems. We propose to illustrate the lacking features in the form of a set of *requirements* that we address when proposing our framework.

Dense Time Being mainly oriented towards discrete system design, Event-B does not embed the concept of time, and instead deals with causality and precedence. To be able to formalise continuous behaviours and constraints, it is required to add the possibility to express and handle explicit *dense time*, on which physical systems are based.

Continuity Expanding on the idea of dense time, events in Event-B are instantaneous and timeless; this cannot be the case of physical behaviours that occur on dense *time interval*, and present a form of *continuity*. To model physical phenomena, it is required to incorporate continuous behaviours in Event-B, both on the modelling side and on the semantics and proof side.

Physical Phenomena The goal of hybrid systems is to control a particular physical phenomenon, called *plant*. Such phenomena are subject to constraints that differs from the ones found in the discrete world. In particular, plants are generally subject to physical environmental constraints (wind, rain, temperature, etc.) that modify their behaviour; this kind of alteration must be taken into account when designing the system, so that it may be able to correct it.

In other words, modelling hybrid systems requires to be able to express both the continuous behaviours of the plant under control, and the effect of the environment on this plant. Note that it is difficult to incorporate every possibility offered by the environment in the model and still be able to prove its correctness and consistency (closed world modelling hypothesis).

Separation Discrete and continuous behaviours occur in separate parts of the system (the controller and the plant respectively), and are thus modelled apart from each other. This separation is convenient as 1) a plant may be controlled by different controllers (that could be switched, during refinement for example) and 2) upon implementation, only the controller is considered (the plant is not directly "implemented").

Interaction and Interleaving Although separate, continuous and discrete behaviours are closely interleaved: the controller's behaviour depends on the plant's behaviour (which it can access, e.g. through sensors), and conversely the plant's behaviour is affected by the controller (e.g. via the use of actuators). In general, discrete behaviours are *timeless* and *instantaneous*; from an abstract point of view, discrete events always happen "exactly at the right time", meaning as soon as they are enabled. Continuous behaviours occur in between those discrete events, and have a duration (meaning they span on a whole time interval).

This discrete/continuous separation-interleaving is close to the way hybrid automata behave (with continuous behaviour happening in the automaton's places and discrete events being its transitions).

3.1.1 Expanding Event-B

In the context of our thesis, we had to address these requirements, in order to provide the foundation of our *framework* for formally designing hybrid systems. In the following, we present the choices we made to handle these requirements.

A Theory of Reals We extended Event-B's expression language with real numbers. The expression language as well as the possibility to define axioms in Event-B allows the formalisation of reals in a semi-algebraic way, rather than using Dedekind cuts or Cauchy sequences (which are too low level with regard to our purpose). This addition, formally implemented using Event-B's theory mechanism, allows defining real variables, and then handle them in expressions and predicates using custom operators and relations.

This allows us to define a state variable for time, in the dense field of reals.

Theories for Continuity and Differential Equations Expanding on the theory of reals, we have further extended Event-B's expression language to be able to define, characterise and handle *continuous functions*. These functions serve as the basis of continuous behaviours: a plant is modelled by *continuous state variables*, which are essentially piece-wise continuous functions of time.

Enriched Semantics To be able to properly characterise continuous behaviours at a method level, we extend Event-B's semantics and add a new category of events, continuous events. Unlike discrete events which are timeless, continuous events are considered to have a *duration*, i.e. last on a specific time interval. During such events, the plant's behaviour is updated, typically using differential equations, and time progresses densely along the time interval. A continuous event is "interrupted" whenever a discrete event is enabled; in that case, the interrupting discrete events are triggered, then execution follows with an other continuous event, until it is interrupted again, and so on. Discrete events are said to be *preemptive* on continuous events.

With this semantics, a model may define discrete and continuous events in a separate way (an event cannot be both), with a strong interleaving policy, ensuring the system can always be well-controlled.

These features have been implemented in the proposed method, in particular using theories. We give an in-depth description of them in Chapter 4.

3.2 Generic Approach

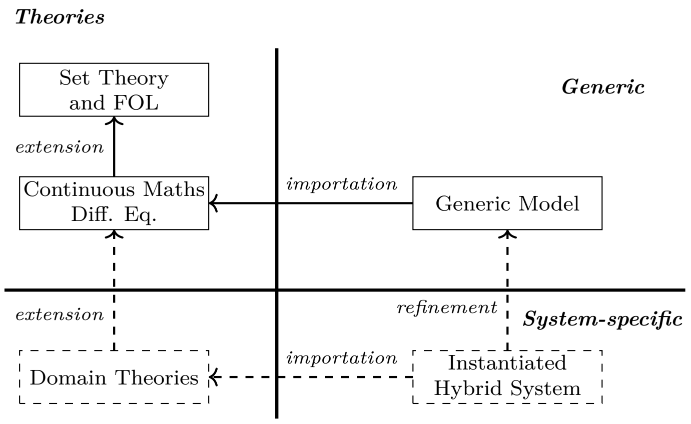


Figure 3.1: Base Approach

The diagram of Figure 3.1 summarises the foundations of our framework. The first important point on which it is based is a collection of *theories* that extends Event-B's core mathematical theory (i.e. set theory and first order logic) with fundamental concepts and constructs, such as continuity and differential equations, essential for expressing continuous dynamics. These theories are more thoroughly discussed in Chapter 4.

The second important point of the approach is the *generic model*. In essence, each hybrid system developed using this approach is a refinement of this generic model. It is an abstraction of controller-plant loop hybrid system, able to capture the behaviour of any hybrid system with this structure. Refinement is then used to narrow down these behaviours and gradually build the desired hybrid system(s). Note that, as expected, it relies on the theories of continuous mathematics and differential equations. We discuss this model and its specificity more extensively in Chapter 5.

In order to model a hybrid system using this framework, two steps need to be taken:

1. A theory needs to be defined to synthesise and gather the knowledge and concepts relative to the specific system to be developed; this theory is said to be a *domain* theory because it is specific to the domain (or "type") of the hybrid system being modelled. Theories on specific machinery (valves, cars, etc.) or type of motion (pendulum, etc.) are defined/used at this step.

2. The generic model needs to be refined in the desired hybrid system, possibly via multiple refinements, and also possibly using some of the components proposed in this framework. Of course, models derived this way generally reference the domain theories defined in the previous point.

3.3 A Framework for Supporting Hybrid System Design

In addition to providing theories and a generic model for safely designing hybrid systems, the other part of the framework makes available multiple formalised patterns, aimed at carrying out development operations or design choices useful for hybrid systems formal modelling.

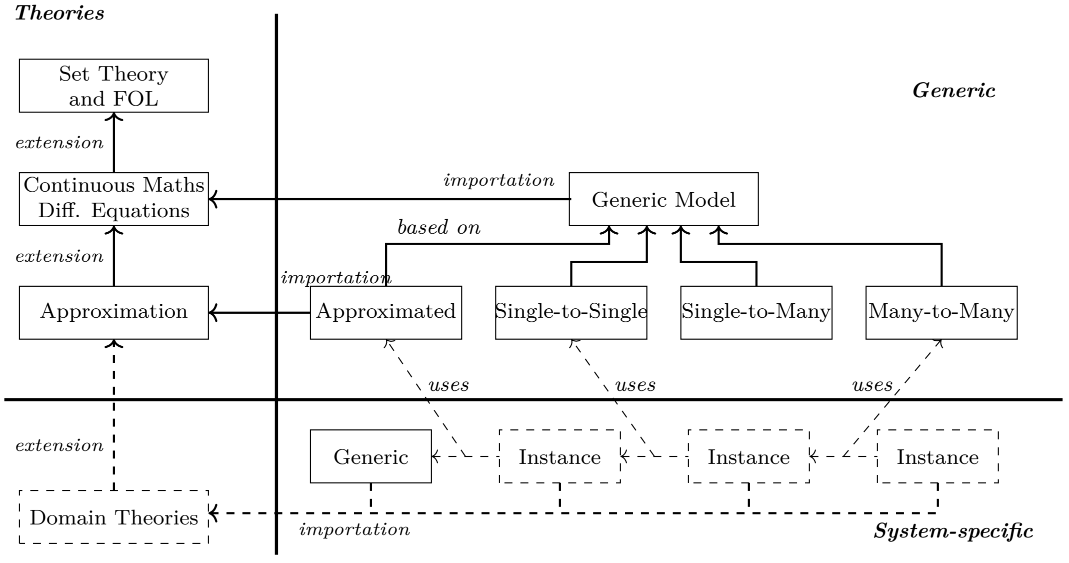


Figure 3.2: Framework Components

The diagram of Figure 3.2 summarises the formal models and patterns we have developed for this framework as well as an example of how to use them. All of these patterns take the form of refinement patterns and methods, accompanied by additional theories. Because these patterns are based on refinement, they may be combined in a number of different ways. Moreover, the framework may be enriched with other patterns and components, while retaining its high degree of composability.

The patterns presented in this thesis can be categorised into two families: structural and behavioural.

Structural patterns allow the projection of a hybrid system or hybrid functionalities onto a particular type of architecture. Hybrid systems typically consists of controllers and plants. Most hybrid system actually consists of *one* controller and *one* plant (something we call *single-controller-to-single-plant*, or *single-to-single* for short); but hybrid systems can feature multiple controllers and multiple plants, tied together in complex ways. Cyber-physical systems [LS14] are an example of this type of architecture: they often consist of multiple autonomous controllers linked together in some way (e.g. by some kind of network) and controlling multiple plants.

In this case, the system is expressed using a particular *architecture pattern*, which represents its overall structure. We identified 3 main architecture patterns:

- one controller with one plant (*single-to-single*), which is the "default" pattern modelled in the core generic model [Dup+18b; Dup+18a];
- one controller with many plants (*single-to-many*) [Dup+19];
- many controllers with many plants (*many-to-many*), which allows modelling autonomous and cyber-physical systems [Dup+20c];

Behavioural patterns are used to formalise an operation carried out on the system's behaviour (in contrast with its structure/architecture). The representative pattern studied in this thesis is approximation: we take a simple hybrid system on which we can prove invariants, then approximate its continuous behaviour and thus obtain a system closer to reality, while preserving the proofs made before [Dup+20b; Dup+20a]. Other examples include: use of a PID (Proportional Integration Derivative) controller to control a plant, discretising continuous behaviours, etc.

3.4 Methodology

The framework we propose serves as support for the design of hybrid systems. It is associated with a general *methodology* that aims at guiding hybrid systems developments, inspired from the Event-B method for designing systems. This methodology intertwines hybrid systems design with the use of our framework in a systematic way, easing the development and verification process.

In this section, we discuss the methodology associated with this framework. Note that this methodology is used throughout this thesis when addressing the various case studies that illustrate our other contributions.

Step 1: Preliminary Study

The goal of this framework is to model controller-plant loop hybrid systems. This structural constraint guides the study of systems to be designed: such systems feature a separate controller and plant, each of which has its own specificity. In practice when writing the actual model, these two parts are associated with their own variables, parameters, guards and invariants.

Plant Description. Hybrid systems revolve around the control of a plant; a logical first step is to study how this plant behaves, including what commands can be issued to it. This part of the study yields the continuous states as well as the various descriptions for its associated continuous behaviours (usually under the form of differential equations).

This also highlights the physical/environment-related properties of the plant, as well as the invariants it should observe.

Controller Description. Hybrid system controllers handle a *mode*, with an associated particular actuation (i.e. a particular behaviour for the plant). Studying these modes and how the system switches from one another lays out the system's events with their associated guards:

- internal calculation, user commands, timers, or other computations that do not relate directly to the plant forms the *transitions* of the system;
- any access to the plant's state is supported by *sensing*, where the actual "reading" of said state is generally expressed by a guard;
- every mode corresponds to one *actuation*, with associated continuous behaviours;

Requirements. The system's properties, both given and expected, are summarised as a list of requirements. In our studies, three types of requirements are identified:

FUN Functional requirements, what the system shall do, its overall goals;
ENV Environmental properties, how the system is constrained by physical means;
SAF Safety properties, what the system must guarantee;

It is worth noticing that **ENV**-type properties are of special interest for hybrid systems that evolve in a constraining environment. Such requirements may refer to gravity, inertia, heat, etc.

In the context of our work, we make the assumption that the model encompasses any *expected behaviour* (good or bad). In particular, parts of the environment are explicitly modelled, and serve to demonstrate how the system is able to maintain its control despite perturbations.

This is known as *close loop modelling*: a finite set of assumptions is made on the system and its environment that represent a *model of the environment.* As for any model, it may be incomplete or imprecise, and the system may be in a situation where these assumptions are not met; in this case, its model cannot be used to predict its behaviour, and ensure its safety.

Step 2: Domain Theories and Additional Components

In practice, hybrid systems are based on specific physical phenomena, with their own general constraints and features. These properties are not directly part of the model, but are used by it in order to establish its behaviour. They model *domain specific knowledge* that are exploited when writing the model and when verifying it. Note that they are generally shared between hybrid systems with similar types of plant: cars, tanks, robots, etc.

Domain specific knowledge is gathered and formalised in special components called *domain specific theories*, referenced in models and proofs.

Step 3: Generic Model Instantiation

The entry point for developing a specific hybrid system is the generic model. This model abstracts controller-plant loop hybrid systems and separates the controller from the plant.

Using the preliminary study, the generic features of this model (variables and parameters) are provided. The system's behaviour, categorised as *transition*, *sensing*, *actuation* and *environment* are introduced by refining the generic model's corresponding events. On the technical level, gluing invariants and witnesses are used to instantiate these features.

At this level, additional invariants are formulated that model the requirements established earlier.

Step 4: Use of Framework Components

The patterns provided in the framework are formalised as refinements: their entry point is a parameterised Event-B model, and its application is a refinement of this model, where parameters are provided using witnesses (instantiation). The main difficulty is to identify which part of the model is to be handled by the component.

Selecting the right component to use and in which order is specific to every system development. It is defined by a *refinement strategy.* General guidelines can be issued:

- architectural patterns are to be applied first, on a system that already has a constrained behaviour, i.e. a global invariant;
- behavioural pattern are to be applied on specific sub-components;
- to avoid interference and proof complexity, patterns shall be applied *one at a time*;

A development for a specific hybrid system is a chain of refinement, corresponding to pattern instantiations. Patterns may be applied to any component of the system, and at any point in the refinement chain.

Extending The Framework

The framework is based on a generic model and a set of patterns, together with general theories for incorporating missing features in Event-B.

This core is stable in the sense that its components do not change, and are proved once and for all; but it is possible to extend the framework, by enriching its core with new components.

For instance, it is possible to define other generic models for modelling different structures of hybrid systems. It is also possible to define new patterns, as a generic refinement of a generic model, possibly supported by new theories.

3.5 Discussion

The goal of this thesis is to provide a framework for designing hybrid systems, based on Event-B. In particular, this framework has the following import characteristics.

Generic and Reusable The proposed framework is generic thanks to the use of abstract Event-B models, parameterised using abstract variables and event parameters. Event-B refinement serves as a base for instantiating models and patterns, making it reusable, since the framework's *core* (upper part of Figure 3.2) is stable, and does not have to be developed again for every hybrid system development.

Extensible The framework's core consists of a set of components that can be extended with the addition of new models, theories and patterns, expressed as refinements of the generic model.

In the remainder of this manuscript, we explore the framework, its features and components, and use it to address multiple case studies.

Chapter 5 presents in detail the generic model, the idea behind its design. It is used to address two cases studies: a car with automatic braking and signalised left-turn assist systems.

Chapter 6 details the architecture patterns hinted in this chapter. We demonstrate the use of these patterns by addressing the case study of the control of the volume of liquid in a tank.

Finally, the approximation pattern is studied in depth in Chapter 7, where it is used to develop two hybrid systems, a robot that visits a given set of targets and an inverted pendulum.

Chapter 4

Fundamental Required Features

Chapter 3 outlined that the modelling of hybrid systems required several important and specific features, revolving around the general concept of continuity: real numbers, continuous functions, etc.

Event-B was developped with the idea of verifying computer systems, and is thus well adapted for handling *discrete* objects, such as integers and other countable sets. Although this is enough for dealing with programs, it is not sufficient for handling continuous features that are part of hybrid systems models, such as continuous functions and differential equations.

Fortunately, since Event-B is based on set theory and first-order logic, it is possible to express these continuous features within its language, using Event-B's extension mechanism (theories) to formalise and provide them in the form of reusable datatypes, operators and properties (Section 2.5).

In this chapter, we give an overview of the features needed to model hybrid systems. In particular, the concept of continuous variables and the operators for handling them are described in Section 4.1. Section 4.2 presents a formal theory of approximation, used in particular when defining the approximation pattern (Chapter 7). Implementations of these features using Event-B theories is discussed in Section 4.3.

Finally, Section 4.4 concludes the chapter with a discussion on the presented theoretical features and associated Event-B theories.

4.1 Basic Hybrid Modelling Features

Hybrid systems deal with notions such as time and continuity, which are absent from the discrete world of Event-B. For this reason, in order to model them, it is necessary to provide a way of handling these notions in the target modelling language. In this section, we present the formal modelling framework that we have devised to be able to handle the required hybrid features in Event-B, without introducing any change in its core modelling language.

4.1.1 Modelling of Time

Unlike purely discrete systems that obey their own "step-wise" causality induced by a scheduler of some kind, continuous phenomena follows physical time, which is usually considered as *dense*, meaning that it flows continuously, without any "jump" and without stopping.

Continuous phenomena then occur on time *intervals*, and we consider that any discrete event happens at the junction between these intervals. However, such discrete events do not interrupt the flow of time, as they have no duration.

For this reason, we represent time as a positive real number ($t \in \mathbb{R}^+$), and its evolution is given "interval-wise": continuous behaviours make the time progress on a specific interval, rather than discretely increasing. Note that, since we can build intervals that are arbitrarily small, it is always possible to make time progress exactly until the next discrete event.

4.1.2 Continuous State Variables

Although the discrete part of hybrid systems can be modelled with standard *point-wise* state variables, it is not the case for the continuous part. Indeed, in between two time points, a physical phenomenon evolves continuously, while a discrete variable remains essentially unchanged. As a way to capture this specificity, we model the plant's state variables using functions of time.

These functions are generally valued in *complete metric spaces* (which allows the definition of derivatives for them among other things), and in particular in sets of the form $S = \mathbb{R}^n$, $n \in \mathbb{N}^*$, which are real vector spaces.

4.1.2.1 Continuous Before-After Predicate

Since plants are modelled using continuous functions, changing their behaviours (to model an actuation, typically) in fact requires to modify the functions modelling this behaviour. One simple intuition that arises from this remark is to say that, whenever a change occurs in the evolution of the plant, the function modelling that evolution is updated as a whole, but this solution is actually quite flawed.

Indeed, when handling continuous behaviours, there are multiple aspects that need to be considered. If t denotes the current time at which the change occurs, we need to take into account

1. what happened before t, or in other words the *past* of the system: this cannot change when the behaviour of the system is modified, as it would mean that the model can revoke its own past, which is impossible.

2. what will happen after t, or in other words the *future* of the system: the system evolves from time t to time $t' > t$, and its behaviour on the interval $[t, t']$ is changed. Also, on this interval, it is expected that nothing else happen, and in particular that the controller will not trigger any actuation. Taking this problem in reverse, and exploiting the particularity of dense time discussed in Section 4.1.1, we always need to make sure t' is such that nothing happen on $[t, t']$.

3. what happens at t, or in other words the *present* of the system: physical phenomena are bound to be fairly regular, although they can present (a countable number of) *jumps*; they are thus usually modelled using piece-wise continuous functions, which always have a right-hand limit, and are always left-hand defined (i.e. functions defined on intervals of the form $[t, t'[$).

Taking this into consideration, it appears that the assignment of continuous state variables is of a very different nature compared to discrete state variables. Just like discrete assignment can be expressed using the before-after predicate :| (see Section 2.2.2.2), we introduce the *continuous before-after predicate* (CBAP), which goal is to embed the properties discussed above.

Operator 7 (Continuous Before-After Predicate)**.** *Let $t, t' \in \mathbb{R}^+$ two time points with $t' > t$. Let $x_p \in \mathbb{R}^+ \to S$ a continuous state variable. Finally, let $\mathcal{P} \subseteq (\mathbb{R}^+ \mapsto S) \times (\mathbb{R}^+ \mapsto S)$ a predicate on the before and after values of the state variable and $H \subseteq S$ an evolution domain constraining the evolution of x_p. The continuous before-after predicate modelling the change of x_p on time interval $[t, t']$ following predicate $\mathcal{P}$ and constrained by evolution domain H, denoted $x_p :|_{t \to t'} \mathcal{P} \,\&\, H$, is defined as so:*

$$\begin{aligned} x_p :|_{t\to t'} \mathcal{P}(x_p, x_p') \,\&\, H \equiv\ & [0,t[\lhd x_p' = [0,t[\lhd x_p && (PP) \\ & \land \mathcal{P}([0,t] \lhd x_p, [t,t'] \lhd x_p') && (PR) \\ & \land \forall t^* \in [t,t'], x_p(t^*) \in H && (LI) \end{aligned}$$

Using the domain restriction operator $\lhd$ (Operator 6), the continuous before-after predicate operator ensures that the past of the continuous variable remains unchanged (past preservation, *PP*) while its future is modified to correspond to the given predicate $\mathcal{P}$ (predicate, *PR*). This assignment also ensures time is progressing along with the physical plant, going from time t to $t' > t$, and that on the resulting interval $[t, t']$ nothing else happens with regard to given invariant H (local invariant, *LI*).

Note: it is sometimes convenient to express this operator under a prefix form (instead of its infix form). We take, as a convention, the following equivalence:

$$x_p :|_{t\to t'} \mathcal{P}(x_p, x_p') \,\&\, H \ \equiv\ CBAP(t, t', x_p, x_p', \mathcal{P}, H)$$

Well-Definedness Correct use of this operator is conditioned by well-definedness requirements: for any given t, t', $\mathcal{P}$ and H, the assignment $x_p :|_{t\to t'} \mathcal{P}(x_p, x_p') \,\&\, H$ is well-defined if and only if:

$$\begin{aligned} & t < t' && (TP) \\ & \forall u, v \cdot \mathcal{P}(u, v) \Rightarrow u \in \mathbb{R}^+ \mapsto S \land v \in \mathbb{R}^+ \mapsto S \land [0,t[\subseteq \mathrm{dom}(u) \land [t,t'] \subseteq \mathrm{dom}(v) && (TD) \end{aligned}$$

This well-definedness condition ensures that, when the operator is used, time keeps moving forward (time progression property, *TP*), and that predicate $\mathcal{P}$ maintains the coherence of the variable's type and domain (type and domain coherence property, *TD*).

Feasibility In addition to the concept of well-definedness, we also need to observe the property of *feasibility*: indeed, a well-defined continuous before-after predicate is not always feasible. Given t, t', $\mathcal{P}$, H and x_p, and provided the assignment is well-defined, feasibility of the $:|_{t\to t'}$ consists in establishing that there exists a function (x_p^0) at least defined on $[t, t']$ that validates predicate $\mathcal{P}$ and does not violate evolution domain H. Formally, feasibility of the CBAP operator can be expressed as so:

$$\begin{aligned} & \exists x_p^0 \in \mathbb{R}^+ \mapsto S \land [t,t'] \subseteq \mathrm{dom}(x_p^0) \land \mathcal{P}([0,t] \lhd x_p, x_p^0) \\ & \qquad\qquad \land \forall t^* \in [t,t'], x_p^0(t^*) \in H \end{aligned} \qquad (CBAPFIS)$$

There is no universal answer to this general proof obligation: it has to be carried out manually and is highly dependent on the predicate's and evolution domain's properties.

In the more precise context of differential equations, we can at least start the proof by observing that, if a differential equation is solvable on $[t, t']$, then (by definition) it admits a solution at least defined on $[t, t']$, which takes care of the first part of the predicate.

The non-violation of the evolution domain ($x_p^0(t^*) \in H$) must still be carried out using special properties of the set or of the differential equations used in the model. On the generic level however, we encompass this feasibility condition in a special predicate, denoted **Feasible**.

Induction Principle on CBAP The CBAP operator generally appears in the actions of a model. As such, they appear in the hypotheses of invariant proof obligations (`INV`). Proving such a PO directly is possibly cumbersome, but the particular form of the CBAP operator can be exploited to simplify and greatly factorise the proof. We give the following theorem that allows easing the proving of the induction step, in machines where CBAP local invariants are stricter than global invariants.

Theorem 1 (CBAP Induction)**.** *Given time points t and t' in $\mathbb{R}^+$ with $t' > t$, before and after continuous states x_p and x'_p in $\mathbb{R} \nrightarrow S$ with $[0,t] \subseteq \mathrm{dom}(x_p)$ and $[0,t'] \subseteq \mathrm{dom}(x'_p)$, a predicate $\mathcal{P} \subseteq (\mathbb{R} \nrightarrow S) \times (\mathbb{R} \nrightarrow S)$, a local invariant $H \subseteq S$ and a global invariant $\mathcal{I} \subseteq S$, such that:*

1. $\forall t^* \in [0,t[, x_p(t^*) \in \mathcal{I}$
2. $\mathbf{CBAP}(t, t', x_p, x'_p, \mathcal{P}, H \cap \mathcal{I})$

Then $\forall t^* \in [0,t'], x'_p(t^*) \in \mathcal{I}$.

The proof of this theorem mainly relies on operator unfolding and basic set theory.

Proof. Let t, t', x_p, x'_p, $\mathcal{P}$, $\mathcal{I}$ and H as in the hypotheses of the theorem.
We recall hypothesis (2) of the theorem: $\mathbf{CBAP}(t, t', x_p, x'_p, \mathcal{P}, H \cap \mathcal{I})$. By unfolding **CBAP** and extract *(LI)* we obtain: $\forall t^* \in [t,t'], x_p(t^*) \in H \cap \mathcal{I}$.
Note that $H \cap \mathcal{I} \subseteq \mathcal{I}$; hence: $\forall t^* \in [t,t'], x_p(t^*) \in \mathcal{I}$.
In addition, hypothesis (1) of the theorem gives $\forall t^* \in [0,t[, x_p(t^*) \in \mathcal{I}$. We note that $[0,t[\cup [t,t'] = [0,t']$, and it follows that: $\forall t^* \in [0,t'], x_p(t^*) \in \mathcal{I}$. □

This theorem states that, if continuous state x_p does not violate global invariant $\mathcal{I}$ on $[0,t[$ (1), and if x'_p does not violate this invariant on $[t,t']$ (2), then x'_p does not violate the invariant on $[0,t']$.

Note that, in practice H and $\mathcal{I}$ are convex hulls, meaning $H \cap \mathcal{I}$ is also a convex hull.

4.1.2.2 Derived Operators

Just like in Event-B with $:=$ and $:\in$, we define convenient derived operators, namely:

Operator 8 (Continuous Assignment)**.** *Let $t, t' \in \mathbb{R}^+$ two time points, with $t' > t$. Let a continuous state variable $x_p \in \mathbb{R}^+ \to S$, a function $f \in \mathbb{R}^+ \to S$ and an evolution domain H. The continuous assignment of f to x_p with evolution domain H, denoted $x_p :=_{t \to t'} f \,\&\, H$, is defined as so:*

$$x_p :=_{t \to t'} f \,\&\, H \;\equiv\; x_p :|_{t \to t'} x'_p = f \,\&\, H \tag{4.1}$$

It "appends" a piece of an (explicit) function to the continuous state variable.

Operator 9 (Continuous Evolution)**.** *Let $t, t' \in \mathbb{R}^+$ two time points, with $t' > t$. Let a continuous state variable $x_p \in \mathbb{R}^+ \to S$, a differential equation $\mathcal{E}$ and an evolution domain H. The continuous evolution of x_p along differential equation $\mathcal{E}$ with evolution domain H, denoted $x_p :\sim_{t \to t'} \mathcal{E} \,\&\, H$, is defined as so:*

$$x_p :\sim_{t \to t'} \mathcal{E} \,\&\, H \;\equiv\; x_p :=_{t \to t'} \eta \,\&\, H \tag{4.2}$$

*where η is **any solution of $\mathcal{E}$ on** $[t,t']$ that does not violate evolution domain H.*

4.2 Theory of Approximation

Approximation is a major operation carried out on hybrid systems, and is one of the tool provided by our framework. For this reason, we have formalised a theory of approximation that defines the various concepts and operators needed to perform this operation. Such a theory is based on strong mathematical foundations that allows it to have multiple good properties, which are crucial when it comes to designing approximated models and, in particular, to prove them.

This section gives the mathematical grounds used to define approximation, and how it can be formalised in Event-B within our formal framework.

In the following, we consider a *metric space* (E, d) where d is a distance. Note that, in practice, since the continuous state variables handled by hybrid systems are valued in $S = \mathbb{R}^n$ which are normed vector spaces, these spaces are also complete metric spaces.

4.2.1 Approximation Operator

The approximation operator is the basic building block of the whole approximation theory. The idea behind this operator is to qualify two points that are "close enough", according to the given distance and an arbitrary bound operator, denoted δ.

Operator 10 (Approximation)**.** *Let $\delta \in \mathbb{R}^+$ (i.e. $\delta \in \mathbb{R}$ and $\delta \geq 0$) and $x, y \in E$. x is **approximately equal** to y by δ (or x is a δ-approximation of y), denoted $x \approx^\delta y$ if:*

$$x \overset{\delta}{\approx} y \;\equiv\; d(x, y) \leq \delta$$

Note that this operator is not an equivalence operator: it is reflexive and symmetric, but not transitive.

Proof of the properties of the approximation operator. The properties of $\approx^\delta$ mostly comes from the properties of the distance operator, and in particular *identity of the indiscernibles* and *symmetry*.

Reflexivity: let $\delta \in \mathbb{R}^+$ and $x \in E$. By the identity of the indiscernibles, $d(x, x) = 0$. Since we have $0 \leq \delta$ by hypothesis, we therefore have $d(x, x) \leq \delta$ and thus $x \approx^\delta x$ by definition $\approx^\delta$. $\square$

Symmetry: let $\delta \in \mathbb{R}^+$ and $x, y \in E$, with $x \approx^\delta y$. By symmetry of d, we have $d(x, y) = d(y, x)$, and since $d(x, y) \leq \delta$ (by definition of $\approx^\delta$), it follows naturally that $d(y, x) \leq \delta$, and thus that $y \approx^\delta x$. $\square$

Non-transitivity: let $\delta \in \mathbb{R}^+$ and $\delta > 0$. We take $x, y, z \in E$ *aligned*: $d(x, y) = \delta$, $d(y, z) = \delta$ and $d(x, z) = 2\delta$; this is possible by the triangle inequality of d ($d(x, z) \leq d(x, y) + d(y, z)$). We see here, by applying the definition of $\approx^\delta$, that we have $x \approx^\delta y$ and $y \approx^\delta z$, but, since $2\delta > \delta$ (because $\delta > 0$), we do *not* have $x \approx^\delta z$, proving that the operator is not transitive. $\square$

Note that despite not having transitivity, the operator features an interesting "pseudo-transitive" property.

Property 1 (Pseudo-Transitivity of the Approximation)**.** *Let $\delta_1, \delta_2 \in \mathbb{R}^+$ and $x, y, z \in E$, with $x \approx^{\delta_1} y$ and $y \approx^{\delta_2} z$. We have:*

$$x \overset{\delta_1+\delta_2}{\approx} z$$

Proof. By unfolding the definition, we can write: $d(x, y) \leq \delta_1$ and $d(y, z) \leq \delta_2$. Using the triangle inequation of d, we can establish that $d(x, z) \leq d(x, y) + d(y, z)$, and thus that $d(x, z) \leq \delta_1 + \delta_2$. It follows, again by definition of $\approx^{\delta}$, that $x \approx^{\delta_1+\delta_2} z$. □

Also, note that this operator has an interesting and useful property of relaxation.

Property 2 (Relaxation of Approximation)**.** *Let $\delta_1 \in \mathbb{R}^+$ and $x, y \in E$ with $x \approx^{\delta_1} y$. If we have $\delta_2 \in \mathbb{R}^+$ with $\delta_2 \geq \delta_1$, then we can write $x \approx^{\delta_2} y$.*

Proof. This is trivially proven by unfolding the definition of the operator. □

This operator can be extended (or "lifted") to be used on functions (on an interval), which is convenient when working with continuous states in a model:

Operator 11 (Function Approximation)**.** *Let $\delta \in \mathbb{R}^+$. Let D be a set and $X \subseteq D$. Let $f, g \in D \nrightarrow E$ with $X \subseteq \mathrm{dom}(f)$ and $X \subseteq \mathrm{dom}(g)$. f is a δ-approximation of g if:*

$$f \overset{\delta}{\underset{X}{\approx}} g \;\equiv\; \forall x \in X, f(x) \overset{\delta}{\approx} g(x)$$

Naturally, this operator preserve the properties of $\approx^{\delta}$.

Important note: $\approx^{\delta}$ can be seen as a relaxed version of the equality operator. In particular, we have the following interesting property:

Property 3 (0-Approximation)**.** *Let $x, y \in E$:*

$$x \overset{0}{\approx} y \Leftrightarrow x = y$$

Proof. This is proved using the identity of the indiscernibles of d: $x \approx^0 y$ is equivalent to $d(x, y) \leq 0$ by definition, and since d is a positive function ($d(x, y) \geq 0$ for any x, y) it follows than $d(x, y) = 0$, which entails that $x = y$. □

4.2.2 Expansion and Shrinking

Based on the approximation operator, two important operators on sets are defined that are used to relax set membership and inclusion, and serve as a base for defining approximation in general at the model level. Additional properties are introduced; they are crucial to establish the properties of this approximation and in helping to prove the related proof obligations that arise from its use.

Definitions The first operator is called *expansion* ($\mathcal{E}_{\delta}(S)$); the basic idea behind this operator is to define some kind of *hull* around a given set, so that the border of this hull is "far enough" (by δ) from the border of the set.

The second operator is called *shrinking* ($\mathcal{S}_{\delta}(S)$); it is basically the dual operation of the *expansion*, and allow the definition of a *subset* of a given set which border is "far enough" from the border of the set, but inside. It can also be seen as a way to define a subset that is always far enough from the complement of the given set.

Operator 12 (δ-Expansion)**.** *Let $\delta \in \mathbb{R}^+$ and $S \subseteq E$. The δ-expansion of S is denoted $\mathcal{E}_\delta(S)$ and is defined as so:*

$$\mathcal{E}_\delta(S) = \{y \in E \mid \exists x \in S, x \overset{\delta}{\approx} y\} = \{y \in E \mid \exists x \in S, d(x, y) \leq \delta\}$$

Operator 13 (δ-Shrinking)**.** *Let $\delta \in \mathbb{R}^+$ and $S \subseteq E$. The δ-shrinking of S is denoted $\mathcal{S}_\delta(S)$ and is defined as so:*

$$\mathcal{S}_\delta(S) = \{x \in S \mid \inf_{y \in E \setminus S} d(x, y) > \delta\} = \{x \in S \mid \forall y \in E \setminus S, d(x, y) > \delta\}$$

These two operators allow handling "approximations of sets", and thus to deal with objects in a way that is less strict (with more "headroom"). Figure 4.1 presents a graphical representation of these operator, synthesising the intuition behind their definitions.

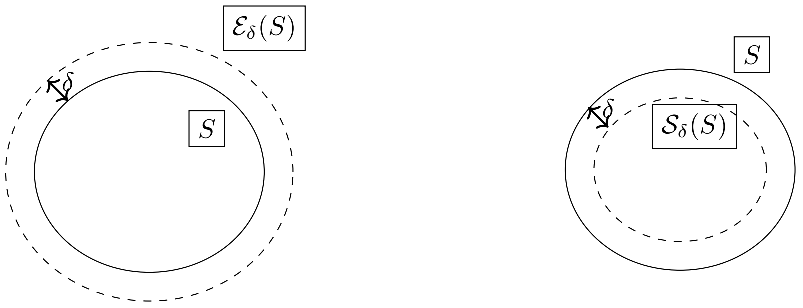


Figure 4.1: Graphical Representation of the Expansion and Shrinking Operators

Expansion and shrinking have multiple important properties that entail the correctness of approximation.

0-Approximation When $\delta = 0$, we show that approximation is, in fact, equality.

Property 4 (0-Expansion/Shrinking)**.** *Let $S \subseteq E$. The 0-expansion and 0-shrinking of S is equal to S:*

$$\mathcal{E}_0(S) = \mathcal{S}_0(S) = S$$

Alternatively, $\mathcal{E}_0$ and $\mathcal{S}_0$ are both the identity function on $\mathbb{P}(S)$.

Proof. These two properties are based on Property 3 and on the property of indiscernible of distance.
0-Expansion: by definition, $\mathcal{E}_0(S) = \{y \in E \mid \exists x \in S, x \approx^0 y\}$, and so, by Property 3, $\mathcal{E}_0(S) = \{y \in E \mid \exists x \in S, x = y\}$, which is trivially equal to S (i.e. the set of elements of E that are equal to an element of S is S). □
0-Shrinking: let us study $E \setminus \mathcal{S}_0(S)$, complement of $\mathcal{S}_0(S)$ in E. For that, we can negate the predicate in the set comprehension, which yields:

$$E \setminus \mathcal{S}_0(S) = \{x \in E \mid x \notin S \land \exists y \in E \setminus S, d(x, y) = 0\}$$

By the property of indiscernible of distance, $d(x, y) = 0$ entails $x = y$, and we see that $E \setminus \mathcal{S}_0(S)$ is in fact exactly $E \setminus S$ (the set of elements that have an equal in $E \setminus S$).
By the properties of complement, this entails naturally that $\mathcal{S}_0(S) = S$. □

δ-Shrinking Property We give an important theorem on $\mathcal{S}_\delta$ that is useful when discussing the proofs generated by approximation.

Let us first recall the definition of a *closed ball* of radius δ.

Definition 6 (Closed Ball). *Let $\delta \in \mathbb{R}^+$ and $x \in E$. The **closed ball** of centre x and radius δ is denoted $\overline{\mathcal{B}}(x, \delta)$ and is the set:*

$$\overline{\mathcal{B}}(x, \delta) = \{y \in E \mid d(x, y) \leq \delta\}$$

Intuitively, it is the set of points that are not farther than δ from x.

We introduce an intermediate lemma, useful to demonstrate the theorem.

Lemma 1 (Closed Ball and Complement). *Let $\delta \in \mathbb{R}^+$ and $S \subseteq E$. For any $x \in \mathcal{S}_\delta(S)$, we have*

$$\overline{\mathcal{B}}(x, \delta) \cap (E \setminus X) = \emptyset$$

Proof. Let $y \in \overline{\mathcal{B}}(x, \delta) \cap (E \setminus S)$. We have $y \in E \setminus S$ and $d(x, y) \leq \delta$ by definition of the closed ball. Because $y \in E \setminus S$, we know by definition of $\mathcal{S}_\delta$ and of the infimum that: $\forall z \in \mathcal{S}_\delta(S), d(z, y) > \inf_{y \in E \setminus S} d(x, y) > \delta$. We hence have both $d(x, y) \leq \delta$ and $d(x, y) > \delta$, which is impossible. □

From this lemma, the important theorem follows naturally.

Theorem 2 (δ-Shrinking and Closed Ball Inclusion). *Let $\delta \in \mathbb{R}^+$, $S \subseteq E$. For any $x \in \mathcal{S}_\delta(S)$,*

$$\overline{\mathcal{B}}(x, \delta) \subseteq S$$

Proof. This is trivially proven using Lemma 1: if $y \in \overline{\mathcal{B}}(x, \delta)$ then $y \notin (E \setminus S)$, hence $y \in (E \setminus (E \setminus S))$, i.e. $y \in S$. □

Intuitively, this theorem states that, if we consider a point x in S that is in $\mathcal{S}_\delta(S)$, then any point that is no further than δ from x, is still in S.

4.2.3 Approximated Predicates

The definition of the approximation operator ($\approx^\delta$, Operator 10) and of δ-expansion ($\mathcal{E}_\delta$, Operator 12) allows us to introduce an approximated version of the set-membership operator, $\in$.

The new operator is further extended in order to be able to use it on functions and sets (which is generally more useful when handling continuous states).

Operator 14 (δ-Membership). *Let $\delta \in \mathbb{R}^+$, $x \in E$ and $S \subseteq E$. x is a δ-member of S, denoted $x \in^\delta S$, if:*

$$x \overset{\delta}{\in} S \;\equiv\; x \in \mathcal{E}_\delta(S) \;\equiv\; \exists y \in S, d(x, y) \leq \delta$$

Intuitively, x is a δ-member of S if it is in S or its *neighbourhood*, up to δ away from S.

This operator can be extended in a number of convenient way, thus allowing the relaxation of a number of classical operators used in set theory.

Operator 15 (δ-Subset). *Let $\delta \in \mathbb{R}^+$ and $T, S \subseteq E$. T is a δ-subset of S, denoted $T \subseteq^\delta S$, if:*

$$\forall x \in T, x \overset{\delta}{\in} S$$

Operator 16 (Functions Extension)**.** *Let $\delta \in \mathbb{R}^+$ and $S \subseteq E$. Let $f \in D \mathrel{+\!\!\!\rightarrow} E$ and $X \subseteq D$ with $X \subseteq \text{dom}(f)$. f is a δ-member of S on X, denoted $f \in^{\delta}_{X} S$, if*

$$f \underset{X}{\overset{\delta}{\in}} S \;\equiv\; \forall x \in X, f(x) \overset{\delta}{\in} S$$

Operator 17 (Multi-Valued Functions Extension)**.** *Let $\delta \in \mathbb{R}^+$. Let $f \in D \mathrel{+\!\!\!\rightarrow} E$ and $\Sigma \in D \mathrel{+\!\!\!\rightarrow} \mathbb{P}(E)$, and let $X \subseteq D$ with $X \subseteq \text{dom}(f)$ and $X \subseteq \text{dom}(\Sigma)$. We define:*

$$f \underset{X}{\overset{\delta}{\in}} \Sigma \;\equiv\; \forall x \in X, f(x) \overset{\delta}{\in} \Sigma(x)$$

The extension of approximate membership to multi-valued function is useful to express, in a formal way, approximate gluing invariants (see Section 7.1.1).

Note that, as expected, with $\delta = 0$, δ-membership actually corresponds to the classical set-membership operator. The same reasoning applies to the derived operators.

4.3 Implementation with Theories

As suggested in this chapter's introduction, modelling hybrid systems require to incorporate, in Event-B, typically continuous mathematics as defined in Sections 4.1 and 4.2. As they rely on set theory and first-order logic, the defined concepts can be expressed using Event-B's expression language, in contexts typically. However, contexts are poorly generic, which would hinder the reusability of the framework. For this reason, we gather those required concepts in *theories* (see Section 2.5).

It is to be noted that a theory for reals has already been defined in the so-called *standard library* of Event-B theories[1], as well as an extension of this theory to incorporate *continuous functions* [BAB16]. However, those theories are insufficient for our needs, and do not fit in our theory collection in a coherent way. This justifies that we propose our own theory of reals, extended and slightly different from a design point of view.

In this section, we present the theories defined in the context of our work. Extracts of these theories are given to support our argumentation; their complete code is available in Appendix A.

4.3.1 Design Choices

As explained in Section 2.5.4, designing theories require a number of choices and trade-offs between usability, expressiveness and exhaustivity. In particular, theories allow providing objects and properties both constructively (with direct definitions) and axiomatically, the former being more sound but possibly more cumbersome, and the latter being more convenient but possibly a source of inconsistency.

The advantage of axioms, particularly in the context of our work, is that they allow the definition of advanced mathematical constructs (e.g. continuity) without having to formalise the entirety of the framework on which they rely (e.g. topology, limits, filters, etc.), and that is less relevant to our use. Once axioms have been defined, they form a basic level of abstraction for the theory, and other

[1]`https://wiki.event-b.org/index.php/Theory_Plug-in#Standard_Library`

operators and properties can be built upon them (in order to keep the number of axioms provided by the theories as low as possible).

Because theories do not include the complete mathematical framework of the operators they define, properties that are typically expressed as theorems are provided as axioms as well. As discussed in Section 2.5.4, these properties may be proven elsewhere (typically, using other proof assistants), and we assume that the *transcription* of the proven properties in Event-B expression language is correct.

4.3.2 An Overview of the Defined Theories

In the context of this thesis, a significant number of theories have been developed, which make up a major technical part of our work, and form the foundation of the framework presented in this manuscript. Defined theories are grouped and organised according to the pattern shown in Figure 4.2.

Each group of theory is built, by extension, upon the previous one, starting with the *general algebra* group, which only rely on bare set theory.

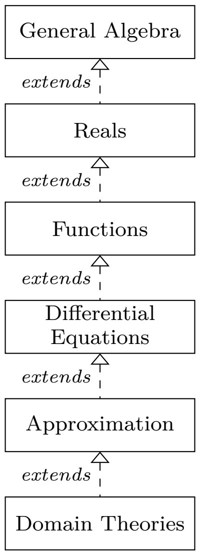


Figure 4.2: Theories Overall Architecture

Theories are categorised as follow:

- *General Algebra*: a set of theories that define the basic structures of general algebra: monoids, groups, rings, etc. (Section 4.3.2.1);
- *Reals*: two theories that define the real number set with its associated operators and properties, as well as intervals (Section 4.3.2.2);
- *Functions*: theories that bring additional structure and properties to function, and in particular continuity and derivatbility, as well as the overall concept of piecewise functions (Section 4.3.2.3);
- *Differential Equations*: a theory that defines the concept of differential equations as well as various useful satellite objects and properties to handle them (Section 4.3.2.4);
- *Approximation*: a pair of theories that define the fundamental concepts needed to handle approximations (Section 4.3.2.5);
- *Domain Theories*: various theories for handling specific types of systems: cars, water tanks, robots, etc. (Section 4.3.2.6);

In the following sections, we will discuss each theory category in detail.

4.3.2.1 Theories of General Algebra

At the root of the theory architecture are 5 theories that define the basic concepts and structures of general algebra. These theories exist mainly as a matter of segmenting the definition of reals, and also for factorising some properties of algebraic structures. Of course, they can nonetheless be used standalone.

These theories all rely on genericity and predicates: the generic type parameter is the underlying set of the structure, and the predicates qualify properties of operators and elements. For instance, the *Monoid* theory defines:

$$neutral(\star : M \times M \to M, e : M) \equiv \forall x \cdot x \in M \Rightarrow e \star x = x \star e = x$$

This can be interpreted as: if we have any set M, $neutral(\star, e)$ is true if and only if e is a neutral element of M for operator $\star$.

The theories defined start with the simple concept of *monoid*, then build the notion of *group* and Abelian/commutative group, and then the structure of *ring*, and finally of *field*.

At every step, various theorems and additional predicates are defined (e.g. uniqueness of neutral element, absorbing property of 0 in a ring, etc.) that are aimed at making proofs and later definitions easier.

Last in this category, a theory of order is defined. Note that this theory does not directly follow the hierarchy of algebraic structures of this category, but is relevant at this level. This theory defines the properties of ordering operators (transitivity, reflexivity, etc.) as well as fundamental additional properties such as well-foundation, and also compatibility properties with algebraic structures.

```
THEORY Monoid
TYPE PARAMETERS M
OPERATORS
    associative predicate (op: M × M → M)
        direct definition
            ∀x, y, z · x ∈ M ∧ y ∈ M ∧ z ∈ M ⇒
                op(x ↦ op(y ↦ z)) = op(op(x ↦ y) ↦ z)
    neutral predicate (op: M × M → M, e: M)
        direct definition
            ∀x · x ∈ M ⇒ op(x ↦ e) = x ∧ op(e ↦ x) = x
    Monoid predicate (op: M × M → M, e: M)
        direct definition
                associative(op) ∧ neutral(op, e)
THEOREMS
    neutralUnique:
        ∀o, e · o ∈ M × M → M ∧ e ∈ M ∧ Monoid(o, e) ⇒
            ∀x · x ∈ M ∧ neutral(o, x) => x = e
END
```

```
THEORY Group
IMPORT THEORY Monoid
TYPE PARAMETERS G
OPERATORS
    invertible predicate (op: M × M → M, e: M)
        direct definition ∀x · x ∈ G ⇒
            ∃y · y ∈ G ∧ op(x ↦ y) = e ∧ op(y ↦) = e
    Group predicate (op: M × M → M, e: M)
        direct definition
            Monoid(op, e) ∧ invertible(op, e)
    ...
THEOREMS
    leftCancel:
        ∀o, e · o ∈ G × G → G ∧ e ∈ G ∧ Group(o, e) ⇒
            ∀a, b, c · a ∈ G ∧ b ∈ G ∧ c ∈ G ⇒
                o(a ↦ b) = o(a ↦ c) ⇔ b = c
    ...
END
```

Listings 4.1: General Algebra Theory Examples

4.3.2.2 Theories of Real Numbers and Intervals

The second group in the theory hierarchy globally defines the set of real numbers, together with various operators and properties, as well as real intervals. An excerpt of it is given in Listing 4.2.

Real numbers in standard mathematics can be defined in multiple ways, and in particular using Dedekind cuts and Cauchy sequences, which require a complete formalisation of topology, rational numbers, sequences, etc. However, using real numbers constructed that way is quite cumbersome in models and proofs, while often only high-level properties are needed (operations, orders, etc.). The fact is, we know real numbers exist and have specific properties (theorems, in essence), and this is sufficient in the context of our work.

For this reason, we decided to define the theory of real in a pure axiomatic way, taking the (otherwise proven) properties of the real number set and taking them as axioms.

The *Reals* theory is based on the *Ring* and *Relations* theories. It defines an axiomatic type (*RReal*) on which a set of axioms is defined, to encode the essential properties of the real number set:

- $\mathbb{R}$ is a (commutative) field with two laws: $+$ (*plus*) and $\times$ (*times*), with two special elements, 0, neutral for $+$ and 1, neutral for $\times$;
- $\mathbb{R}$ is totally ordered by an order relation denoted $\leq$ (*leq*) which is *compatible* with its field structure;
- $\mathbb{R}$ has the least upper bound property;

Based on this, we also define the unary and binary minus operators (*uminus*, *minus*), division (*divides*), strict order (*lt* for $<$) and converse order operators (*geq* for $\geq$ and *gt* for $>$), as well as a few other convenient operators (such as square root, absolute value, etc.).

A few theorems have been defined to ease the handling of reals throughout models, as well as an extensive number of proof rules, as to make available standard simple transformations immediately in the prover.

```
THEORY Reals
IMPORT THEORY Relations , Ring
AXIOMATIC DEFINITIONS
  TYPE RReal
  OPERATORS
    Rzero expression ()  : RReal
    Rone expression ()  : RReal
    plus expression ()  : ℙ(RReal × RReal × RReal)
    times expression ()  : ℙ(RReal × RReal × RReal)
    leq expression ()  : ℙ(RReal × RReal)
  AXIOMS
    oneIsNotZero : Rzero ≠ Rone
    plus_def : plus ∈ RReal × RReal → RReal
    times_def : times ∈ RReal × RReal → RReal
    leq_def : order(leq) ∧ total(leq)
    realField : Field(plus, times, Rzero, Rone) ∧ integral(times, Rzero)
    ...
```

Listings 4.2: Real Theory Extract

The real number theory is accompanied by an extensive theory of real intervals, called *Intervals*, an extract of which is given in Listing 4.3. This theory defines every possible variety of intervals (i.e. open, closed, half-open, unbounded) as well as a number of theorems and proofs that allow performing standard operations on them in the prover – typically, union, intersection and degenerated intervals.

Note: in the following, we use the standard mathematical notation for the set of reals ($\mathbb{R}$) as a shorthand for the *RReal* type, as well as standard symbols for *Rzero* (0) and *Rone* (1), *plus* (+) and so on.

4.3.2.3 Theories of Functions and Piecewise Functions

Functions in Event-B are essentially defined as univalent or right-unique relations (otherwise known as “functional relations”). The language allows the characterisation of total, injective, surjective and bijective functions, but in general they are seen as low-level, set-theoretic objects.

```
THEORY Intervals
IMPORT THEORY Reals
OPERATORS
  Closed2Closed expression (a: RReal, b: RReal)
    direct definition {t | a ↦ t ∈ leq ∧ t ↦ b ∈ leq}
  Closed2Infinity expression (a: RReal)
    direct definition {t | a ↦ t ∈ leq}
  ...
THEOREMS
  c2c_existence: ∀a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    ∃x · x ∈ Closed2Closed(a, b)
  boundaryInC2C: ∀a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    a ∈ Closed2Closed(a, b) ∧ b ∈ Closed2Closed(a, b)
  ...
```

Listings 4.3: Interval Theory Extract

In particular, in order to be able to model hybrid systems, notions of topology, and in particular continuity and differentiability, are lacking. For this reason, we have defined an extensive theory to add to enrich Event-B's modelling language with a number of relevant characteristics and constructs on functions.

The *Functions* theory is based on the *Intervals* theory. An extract of it is given in Listing 4.4. Typically, it defines:

- Predicates on functions: *increasing*, *decreasing*, *bounded*, *positive*, etc. as well as a number of theorems to establish these predicates or use these predicates to deduce particular characteristics;
- Constructors: refined composition and decomposition operators, higher-order real operation (e.g. *Rfplus* for computing the $h = f + g$ where f and g are functions) as well as an extensive number of proof rules for typing the results of these operators, and to (partially) evaluate these operators within the prover;
- Continuity and differentiability: set of continuous/differentiable functions, derivative operator as well as a number of properties of functions and operators;
- Lipschitz continuity, required in particular in the Cauchy-Lipschitz theorem (used for establishing differential equations solution existence), and properties of functions and operators with regard to this type of continuity;
- Basic topology: openness of a set and properties of opens with regard to continuous functions;
- Trigonometric functions: π constant, sin and cos functions;

The *Functions* theory is the base of the *Piecewise* theory, which transposes some of the properties and characteristics of functions into *piece-wise* defined functions. In particular, the theory defines piece-wise continuous/differentiable functions and adapts standard theorems to this situation. This is especially interesting since, in practice, the functions describing differential equations are piece-wise defined.

```
THEORY Functions
IMPORT THEORY Intervals
TYPE PARAMETERS E, F, G
OPERATORS
  bind expression (f: E ⇸ F, g: E ⇸ G)
    direct definition (λx · x ∈ dom(f) ∩ dom(g) | f(x) ↦ g(x))
  fproj1 expression (f: E ⇸ F × G)
    direct definition (λx ↦ y · x ∈ F ∧ y ∈ G | x) ∘ f)
  ...
AXIOMATIC DEFINITIONS
  OPERATORS
    C0 expression (A: ℙ(E), B: ℙ(F)) : ℙ(ℙ(E × F))
    D1 expression (A: ℙ(ℝ), B: ℙ(F)) : ℙ(ℙ(ℝ × F))
    Der expression (A: ℙ(ℝ), B: ℙ(F), f: ℝ ⇸ F) : ℙ(ℝ × F)
      well−definedness condition f ∈ D1(A, B) ∧ A ⊆ dom(f)
    lipschitzContinuous predicate (A: ℙ(E), B: ℙ(F), f: E ⇸ F)
      well−definedness condition A ⊆ dom(f)
    ...
  AXIOMS
    derType: ∀A, B, f · A ⊆ ℝ ∧ B ⊆ F ∧ f ∈ E ⇸ F ∧ f ∈ D1(A, B) ∧ A ⊆ dom(f) ⇒
      Der(A, B, f) ∈ A → F
    lipschitzIsCue: ∀A, B, f · A ⊆ E ∧ B ⊆ F ∧ f ∈ E ⇸ F ∧ A ⊆ dom(f) ⇒
      lipschitzContinuous(A, B, f) ⇒ f ∈ C0(A, B)
    ...
```

Listings 4.4: Theory of Functions Extract

Domain Restriction of Predicates An important design choice in these two theories is to make explicit the domain on which a predicate is considered. In other words, instead of having predicates defined on the whole domain of functions and restricting this domain upon instantiation using restriction operations ($\lhd$ in particular), predicates explicitly require a domain of application.

Typically, for any property $\mathcal{P}$ on a function $f : E \nrightarrow F$, an associated operator $\mathbf{op}_{\mathcal{P}}$ in the theory would be defined as shown on the example of Listing 4.5.

```
OPERATORS
  ...
  op_𝒫 predicate (D: ℙ(E), f: E ⇸ F)
    well−definedness D ⊆ dom(f)
    direct definition 𝒫(D ◁ f)
  ...
```

Listings 4.5: Piece-wise Predicates Definition

Such a definition embeds the restriction of the predicate to a given domain. Note the use of the `well-definedness` clause provided by theories in order to enforce the correct use of the operator, and only use it on functions that defined (at least) on the provided domain D.

A typical example is given in Listing 4.6 for the *increasing* predicate.

```
THEORY Functions
IMPORT THEORY Intervals
 ...
OPERATORS
   ...
   increasing predicate (D: ℙ(ℝ), f: ℝ ⇸ ℝ)
      well−definedness D ⊆ dom(f)
      direct definition ∀x, y · x ∈ D ∧ y ∈ D ⇒
         (x ↦ y ∈ leq) ⇔ (f(x) ↦ f(y) ∈ leq)
   ...
```

Listings 4.6: Definition of the *increasing* predicate

Direct Product, Vectors The theory defines a **bind** operator that takes two functions f and g and builds a new function that outputs pairs of values of f and g:

$$\forall x \in \mathrm{dom}(f) \cap \mathrm{dom}(g), \mathbf{bind}(f, g)(x) = (f(x), g(x))$$

This function allows building complete continuous states from a number of separated state variables. Typically, a system that handles position p and speed v, that are both continuous variables, has a complete state expressed as $\mathrm{bind}(v, p)$. If acceleration must be considered as well, it is possible to nest the **bind** operator: $\mathrm{bind}(\mathrm{bind}(a, v), p)$.

In practice, continuous states are expressed using *vector functions* $\mathbb{R}^+ \to \mathbb{R}^n$ $(n \in \mathbb{N}^*)$. This allows defining as many variables as needed without any modelling overhead (e.g. nested operators) and allows expressing some categories of differential equations using matrices and matrix-vector products.

Ideally, we would need to define a vector and matrix theory in order to be able to handle such a useful mathematical tool; no such theory is available as for now, although we have studied the possibility of formalising it.

4.3.2.4 Theory of Differential Equations

The continuous part of hybrid systems are usually characterised using differential equations (i.e. equations which solutions are functions). The topic of differential equations is extremely vast, ranging from simple equations with trivial properties to intricate partial differential equations with complex properties and theorems; in the proposed theory, we have essentially made accessible the part of it we needed to model hybrid systems' plants behaviours. However we kept the theory both generic and extensible, so that other properties and overall categories of equation can be added later on, if and when required.

Differential Equation Datatype The differential equations theory (*DiffEq*) revolves around a custom datatype, $\mathbf{DE}(F)$, with F an Event-B type (that is, a carrier set, an other datatype, a powerset or a Cartesian product) that essentially represents the set of differential equations which solutions are valued in F. This type is accompanied by particular constructors to create actual differential equation "objects". We defined two constructors for this type:

- $\mathbf{ode}(\Phi, \eta_0, t_0)$, used to define ordinary differential equations (ODE) $\dot{\eta}(t) = \Phi(t, \eta(t))$, characterised by function $\Phi : \mathbb{R} \times F \to F$ with initial condition $\eta(t_0) = \eta_0$;

- **aode**(Φ, η_0, t_0), used to defined *autonomous* ODEs $\dot{\eta}(t) = \Phi(\eta(t))$, characterised by function $\Phi : F \to F$ (that does not rely directly on time) with initial condition $\eta(t_0) = \eta_0$;

The theory also provides a custom datatype for *controlled* differential equations, **CDE**(F, UF), with F and UF Event-B types that represent the set of controlled differential equations which solutions are valued in F and which control functions are valued in UF. Similarly to non-controlled differential equations, we defined two constructors for this type:

- **code**(Φ, η_0, t_0), used to defined controlled ODEs $\dot{\eta}(t) = \Phi(t, \eta(t), u(t))$, for any given control function $u : \mathbb{R} \to UF$, characterised by $\Phi : \mathbb{R} \times F \times UF \to F$ with initial condition $\eta(t_0) = \eta_0$;

- **caode**(Φ, η_0, t_0), used to defined controlled autonomous ODEs $\dot{\eta}(t) = \Phi(\eta(t), u(t))$, characterised by $\Phi : F \times UF \to F$ with initial condition $\eta(t_0) = \eta_0$;

The header of the theory with the defined types is presented in Listing 4.7.

```
THEORY DiffEq
IMPORT THEORY Piecewise
TYPE PARAMETERS E, F, UF, F1, F2
DATATYPES
  DE(F)
  constructors
    ode(fun : ℙ(ℝ × F × F), initial : F, initialArg : ℝ)
    aode(afun : ℙ(F × F), ainitial : F, ainitialArg : ℝ)
  CDE(F, UF)
  constructors
    code(cfun : ℙ(ℝ × F × UF × F), cinitial : F, cinitialArg : ℝ)
    caode(cafun : ℙ(F × UF × F), cainitial : F, cainitialArg : ℝ)
...
```

Listings 4.7: *DiffEq* Theory Header – Custom Datatypes

The majority of the operators and properties given in this theory refer to these types; the "lower level" operators refer directly to these constructors directly (like the cases of a pattern matching construct), but most of the operators have been thought in a generic way, i.e. agnostic of the content of the element.

Providing such a concept that way has numerous advantages. The fact that differential equations are a proper type forces the designer to treat them as a special kind of objects. This allows them to integrate smoothly within models while preventing misuse of these objects. Moreover, since each form of differential equation is essentially represented by a constructor, adding new forms is simply a matter of creating new constructors, and adding such new cases in these "lower level" operators.

Predicates on Differential Equations A large part of the theory relates to basic properties of differential equations, and in particular solvability and controllability. Examples of how these operators are implemented in the theory are given in Listing 4.8.

One of the most important operator is the predicate **solutionOf**$(D, \eta, \mathcal{E})$ that models the fact that $\eta : \mathbb{R} \rightarrow\!\!\!\!\mapsto F$, with $D \subseteq \mathrm{dom}(\eta)$ is a solution of equation $\mathcal{E} : \mathbf{DE}(F)$. This operator is at the core of continuous before-after predicates in particular, since in an implicit context it can mean *let η be a funciton* ***such that*** *it is a solution of equation $\mathcal{E}$.* It is accompanied by the operator **SolutionsOf**$(D, \mathcal{E})$, the set of functions defined on D (at least) that are solutions of equation $\mathcal{E}$.

Symmetrically, for well-definedness purposes, we define the **Solvable**$(D, \mathcal{E})$ predicate that represents the fact equation $\mathcal{E}$ admits solutions on domain D. This predicate is used as a well-definedness condition in many other operators, since we can manipulate solutions of equations if and only if said solutions exist.

As for controlled differential equations, we provide the **withControl**$(D, \mathcal{E}, \hat{u})$ operator that take a controlled differential equation $\mathcal{E} \in \mathbf{CDE}(F, UF)$ and a control function $\hat{u} : D \to UF$ and builds the (non-controlled) differential equation $\dot{\eta}(t) = \Phi(t, \eta(t), \hat{u}(t))$. We also define the **Controllable**$(D, \mathcal{E})$ predicate that models the fact that there exist a control function $u : D \to UF$ such that controlled differential equation $\mathcal{E}$ is solvable on D when controlled using u. A predicate **SolvableWith**$(D, \mathcal{E}, \hat{u})$ represents the fact controlled equation $\mathcal{E}$ is solvable on D when given control function $\hat{u}$.

Note that the proposed predicates all follow the remark of Section 4.3.2.3 by explicitly requiring a domain parameter on which they are to be considered. This is especially relevant here, since in practice we always handle differential equations on a very specific interval, and never on $\mathbb{R}$ as a whole.

```
OPERATORS
  solutionOf predicate (D: $\mathbb{P}(\mathbb{R})$, $\eta$: $\mathbb{R} \nrightarrow F$, $\mathcal{E}$: $\mathbf{DE}(F)$)
    well−definedness condition $D \subseteq \mathrm{dom}(\eta)$
    case $\mathcal{E}$
      ode($\Phi, \eta_0, t_0$) =>
          $\eta_0 \in F \land t_0 \in D \land \eta \in \mathbf{D1}(D, F) \land \eta(t_0) = \eta_0$
          $\land \mathbf{Der}(D, F, \eta) = (\lambda \hat{t} \cdot \hat{t} \in D \mid \Phi(\hat{t}, \eta(\hat{t})))$
      aode($\Phi, \eta_0, t_0$) => ...
  SolutionsOf expression (D: $\mathbb{P}(\mathbb{R})$, $\mathcal{E}$: $\mathbf{DE}(F)$)
    direct definition $\{\hat{\eta} \mid \hat{\eta} \in \mathbb{R} \nrightarrow F \land D \subseteq \mathrm{dom}(\hat{\eta}) \land \mathbf{solutionOf}(D, \hat{\eta}, \mathcal{E})\}$
  Solvable predicate (D: $\mathbb{P}(\mathbb{R})$, $\eta$: $\mathbf{DE}(F)$)
    direct definition $\exists \hat{\eta} \cdot \hat{\eta} \in \mathbb{R} \nrightarrow F \land D \subseteq \mathrm{dom}(\hat{\eta}) \land \mathbf{solutionOf}(D, \hat{\eta}, \mathcal{E})$
  withControl expression (D: $\mathbb{P}(\mathbb{R})$, $\mathcal{E}$: $\mathbf{CDE}(F, UF)$, $\hat{u}$: $\mathbb{R} \nrightarrow UF$)
    well−definedness condition $D \subseteq \mathrm{dom}(\hat{u})$
    case $\mathcal{E}$
      code($\Phi, \eta_0, t_0$) =>
          ode(($\lambda \hat{t} \mapsto \hat{\eta} \cdot \hat{t} \in D \land (\hat{t} \mapsto \hat{\eta} \mapsto \hat{u}(\hat{t})) \in \mathrm{dom}(\Phi) \mid \Phi(\hat{t} \mapsto \hat{\eta}(\hat{t}) \mapsto \hat{u}(\hat{t}))), \eta_0, t_0$)
      caode($\Phi, \eta_0, t_0$) => ...
  ...
```

Listings 4.8: *DiffEq* Theory – Predicates

Hybrid Assignment Operators The other important part of the theory is to define the hybrid assignment operators as presented and discussed in Section 4.1.2, allowing actually handling differential equations and continuous system state in models.

The most basic of these operators is **CBAP**$(t, t_p, \eta, \eta_p, \mathcal{P}, \mathcal{I})$, which essentially allows encoding the continuous before-after predicate (CBAP, $:|_{t \to t'}$) defined in Section 4.1.2.1. It is defined, in the *DiffEq* theory, as shown in Listing 4.9.

As $:|_{t \to t'}$ is accompanied by convenient "shortcut" operators (see Section 4.1.2.2), the theory likewise defines other convenient continuous before-after predicate extensions. For instance, the predi-

```
CBAP predicate (t: ℝ⁺, t_p: ℝ⁺, η: ℝ ⇸ F, η_p: ℝ ⇸ F,
                          𝒫: ℙ((ℝ ⇸ F) × (ℝ ⇸ F)), H: ℙ(F))
  well−definedness condition
    $[0,t] \subseteq \mathrm{dom}(\eta) \wedge [0,t_p] \subseteq \mathrm{dom}(\eta_p)$
  direct definition
    $t \mapsto t_p \in lt \wedge [0,t[\lhd\eta = [0,t[\lhd\eta_p\wedge$
    $([0,t] \lhd \eta) \mapsto ([t,t_p] \lhd \eta_p) \in \mathcal{P}\wedge$
    $\forall\hat{t}\cdot\hat{t} \in [t,t_p] \Rightarrow \eta_p(\hat{t}) \in H$
```

Listings 4.9: *DiffEq* Theory – CBAP

cate **CBAPsolutionOf**$(t, t_p, \eta, \eta_p, \mathcal{E}, H)$, represents $:\sim_{t\to t'}$, where $\mathcal{E} \in \mathbf{DE}(F)$. An implementation of such operator is given in Listing 4.10.

```
CBAPsolutionOf predicate (t: ℝ⁺, t_p: ℝ⁺, η: ℝ ⇸ F,
                                     η_p: ℝ ⇸ F, ℰ: DE(F), H: ℙ(F))
  well−definedness condition
    $[0,t] \subseteq \mathrm{dom}(eta) \wedge [0,t_p] \subseteq \mathrm{dom}(\eta_p) \wedge \mathbf{Solvable}([t,t_p],\mathcal{E})$
  direct definition
      $\mathbf{CBAP}(t,t_p,\eta,\eta_p,(\mathbb{R} \nrightarrow F) \times \mathbf{SolutionsOf}([t,t_p],\mathcal{E}),H)$
```

Listings 4.10: *DiffEq* Theory – CBAPsolutionOf

Other convenient predicates have also been defined, aimed at easing the writing of guards and witnesses and general. For example, **CBAPFIS**$(t, t_p, \eta, \mathcal{P}, H)$ synthesises the feasibility of the corresponding **CBAP** use: it states that there exists η_p such that **CBAP**$(t, t_p, \eta, \eta_p, \mathcal{P}, H)$ is true. This is typically used in a guard as hypothesis for the event's feasibility and as goal of the guard strengthening proof obligation when refining.

Likewise, the **CBAPsolutionOfFIS**$(t, t_p, \eta, \mathcal{E}, H)$ operator is defined, to encompass the feasibility of the **CBAPsolutionOf**$(t, t_p, \eta, \eta_p, \mathcal{E}, H)$ operator, i.e. that there exists a solution of $\mathcal{E}$ on domain $[t, t_p]$ that remains in H on that same domain.

In models, the **CBAP** predicate is used to encode the actual CBAP operator (Operator 7) using Event-B's before-after predicate. Typically, the CBAP assignment $x_p :|_{t\to t_p} \mathcal{P}\ \&\ H$ can be written in Event-B as shown on Listing 4.11. Note that, to enforce its well-definedness, **CBAPFIS** is used in the guards (`grd2`).

```
 Event
ANY t_p
WHERE
   grd1: $t \mapsto t_p \in lt$
   grd2: $\mathbf{CBAPFIS}(t,t_p,x_p,\mathcal{P},H)$
THEN
   act1: $t, x_p :\mid t' = t_p \wedge x_p \in \mathbb{R}^+ \nrightarrow S \wedge [0,t_p] \subseteq \mathrm{dom}(x_p)\wedge$
         $\mathbf{CBAP}(t,t',x_p,x_p',\mathcal{P},H)$
END
```

Listings 4.11: CBAP Encoding in Models

Properties and Theorems Similarly to other theories, the differential equations theory defines numerous theorems and axioms in order to use the defined operators efficiently in proofs. Most theorems defined allow deducing essential properties that often appear as proof obligations, in particular feasibility and invariant preservation.

Solvability Multiple theorems are defined to handle solvability and in particular to be able to deduce it. For example, we know that, if $\mathcal{E}$ is solvable on D, then it is solvable on any $D' \subseteq D$; this allows working with restrictions, and henceforth with broader solvable properties (e.g.: $y' = y$ is solvable on $\mathbb{R}$, therefore it is solvable on $[t, t']$).

Establishing the solvability of an equation is usually a per-case problem, and depends on the equation's form. Fortunately, for ODEs (which is the most common form of equation), the Cauchy-Lipschitz theorem may be used. This theorem is encoded in the theory, and it is used in proofs whenever an ODE is used.

First of all, let us define the concept of *Lipschitz continuity.*

Definition 7 (Lipschitz Continuity)**.** *Let (E, d_E) and (F, d_F) be two metric spaces (with d_E and d_F) their respective distances. Let $D \subseteq E$ and $f \in E \to F$.*

*f is **Lipschitz-continuous** on D if there exists $K \in \mathbb{R}^+$ such that:*

$$\forall x_1, x_2 \in D, d_F(f(x_1), f(x_2)) \leq K \cdot d_E(x_1, x_2)$$

This definition is used in the Cauchy-Lipschitz theorem.

Theorem 3 (Cauchy-Lipschitz Theorem)**.** *Let F be a **Banach space** (complete metric space). Given two domains $D \subseteq \mathbb{R}$ and $D_F \subseteq F$, a function $\Phi \in \mathbb{R} \times F \to F$ and initial conditions $(t_0, \eta_0) \in D \times D_F$ such that:*

1. *Φ is **continuous** on $D \times D_F$*
2. *for any t_0, $\Phi(t_0, \cdot)$ is **Lipschitz-continuous**[2] on D_F*

*then the ODE entailed by Φ admits solutions on $D \times D_F$ with $\eta(t_0) = \eta_0$, and in particular a unique **maximal** solution.*

Note that, in practice, hybrid systems are described using continuous state space of the form $\mathbb{R}^n$, $n \in \mathbb{N}^*$, which are Banach spaces.

The hypotheses of the Cauchy-Lipschitz theorem have been encompassed in one operator called **CauchyLipschitzCondition**. This condition is usually given as an axiom in contexts or (domain) theories that define differential equation.

Listing 4.12 gives an implementation of the Cauchy-Lipschitz theorem together with the definition of the **CauchyLipschitzCondition** operator.

Continuous Assignment The theory provides a number of useful theorems that help in establishing machine consistency when using continuous assignment oeprators. For example, we demonstrate that **CBAPFIS** entails feasibility of actions based on **CBAP**, to help discharging `FIS` POs arising from the use of our custom operators. This theorem is declined for **CBAPsolutionOf**.

The theory also encodes Theorem 1 as to help discharging invariant proof obligations (`INV`).

We give an implementation of these properties in the theories in Listing 4.13.

[2]It is sufficient that $\Phi(t_0, \cdot)$ is only *locally* Lipschitz-continuous on D (i.e. Lipschitz-continuous on any neihborhood of $x \in D$); this version of the theorem is stronger but easier to handle in proofs.

```
OPERATORS
  ...
  CauchyLipschitzCondition predicate (D: $\mathbb{P}(\mathbb{R})$, $D_F$: $\mathbb{P}(F)$, $\mathcal{E}$: $\mathbf{DE}(F)$)
    case $\mathcal{E}$
      $\mathbf{ode}(\Phi, \eta_0, t_0)$ => $D \times D_F \subseteq \mathrm{dom}(\Phi) \wedge t_0 \in D \wedge \eta_0 \in D_F \wedge \Phi \in \mathbf{C0}(D \times D_F, F) \wedge$
                          $\forall \hat{t} \cdot \hat{t} \in D \Rightarrow \mathbf{lipschitzContinuous}(D_F, F, (\lambda x \cdot x \in D_F \mid \Phi(\hat{t}, x)))$
      $\mathbf{aode}(\Phi, \eta_0, t_0)$ => ...
  ...
AXIOMATIC DEFINITIONS
  AXIOMS
    CauchyLipschitz:
      $\forall \mathcal{E}, D, D_F \cdot \mathcal{E} \in \mathbf{DE}(F) \wedge D \subseteq \mathbb{R} \wedge D_F \subseteq F \wedge \mathbf{CauchyLipschitzCondition}(D, D_F, \mathcal{E})$
        $\Rightarrow \mathbf{Solvable}(D, \mathcal{E})$
```

Listings 4.12: *DiffEq* Theory – Cauchy-Lipschitz Theorem and Operator

```
THEOREMS
  ...
  CBAPFIS_act_FIS:
    $\forall t, t_p, \eta, \mathcal{P}, H \cdot t \in \mathbb{R}^+ \wedge t_p \in \mathbb{R}^+ \wedge t \mapsto t_p \in lt \wedge \eta \in \mathbb{R} \nrightarrow F \wedge [0, t] \subseteq \mathrm{dom}(\eta) \wedge$
    $\mathcal{P} \in \mathbb{P}((\mathbb{R} \nrightarrow F) \times (\mathbb{R} \nrightarrow F)) \wedge H \in \mathbb{P}(F) \wedge \mathbf{CBAPFIS}(t, t_p, \eta, \mathcal{P}, H)$
      $\Rightarrow \exists \eta_p \cdot \eta_p \in \mathbb{R} \nrightarrow F \wedge [0, t_p] \subseteq \mathrm{dom}(\eta_p) \wedge \mathbf{CBAP}(t, t_p, \eta, \eta_p, \mathcal{P}, H)$
  CBAP_INF:
    $\forall t, t_p, \eta, \eta_p, \mathcal{P}, H, \mathcal{I} \cdot t \in \mathbb{R}^+ \wedge t_p \in \mathbb{R}^+ \wedge t \mapsto t_p \in lt \wedge H \in \mathbb{P}(F) \wedge \mathcal{I} \in \mathbb{P}(F) \wedge$
    $\eta \in \mathbb{R} \nrightarrow F \wedge [0, t] \subseteq \mathrm{dom}(\eta) \wedge \eta_p \in \mathbb{R} \nrightarrow F \wedge [0, t_p] \subseteq \mathrm{dom}(\eta_p) \wedge$
    $\mathcal{P} \in \mathbb{P}((\mathbb{R} \nrightarrow F) \times (\mathbb{R} \nrightarrow F)) \wedge \mathbf{CBAPFIS}(t, t_p, \eta, \mathcal{P}, H \cap \mathcal{I}) \wedge$
    $\forall \hat{t} \cdot \hat{t} \in [0, t] \Rightarrow \eta(\hat{t}) \in \mathcal{I}$
      $\Rightarrow \forall \hat{t} \cdot \hat{t} \in [0, t_p] \Rightarrow \eta(\hat{t}) \in \mathcal{I}$
  ...
```

Listings 4.13: *DiffEq* Theory – Continuous Assignment Properties

4.3.2.5 Theories of Approximation

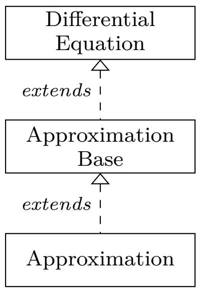


Figure 4.16: Theories of Approximation – Architecture

The last two theories we have defined relate to the handling of approximation, as discussed in Section 7. The first theory (*ApproximationBase*) defines the foundations of approximation (axiomatically) and the second (*Approximation*) elaborates on these definitions and makes accessible the various useful operators and theorems that ought to be used in models and proofs. The architecture of theories for approximation is detailed in Figure 4.16.

The core of the approximation theories is the operator **DeltaNeighborhood**(δ, x, y), which represents the fact that two points x and y are no farther than δ (δ positive real number) from each other. Note that no assumption is made on the set of x and y, and thus no assumption is made on the distance used by the operator, making this predicate generic. This is relevant as one may use particular metrics in developments (e.g. different types of norms and distances).

This operator (otherwise denoted $\approx^{\delta}$ and more thoroughly studied in Section 4.2.1) is accompanied by multiple properties, including commutativity and reflexivity, as well as a particular form of transitivity. It may also be "widened" (meaning if $a \approx^{\delta} b$ then $a \approx^{\delta'} b$ if $\delta' \geq \delta$).

An extract of this theory is given in Listing 4.14.

```
THEORY ApproximationBase
IMPORT THEORY DiffEq
TYPE PARAMETERS E , F
AXIOMATIC DEFINITIONS
  OPERATORS
    DeltaNeighborhood predicate (δ: ℝ⁺ , a: F , b: F )
  AXIOMS
    deltaN_commutative: $\forall \delta, a, b \cdot \delta \in \mathbb{R}^{+} \land a \in F \land b \in F \Rightarrow$
      $\textbf{DeltaNeighborhood}(\delta, a, b) \Leftrightarrow \textbf{DeltaNeighborhood}(\delta, b, a)$
    deltaN_refl: $\forall \delta, a \cdot \delta \in \mathbb{R}^{+} \land a \in F \Rightarrow \textbf{DeltaNeighborhood}(\delta, a, a)$
    deltaN_widen: $\forall \delta_1, a, b \cdot \delta_1 \in \mathbb{R}^{+} \land a \in F \land b \in F \land \textbf{DeltaNeighborhood}(\delta_1, a, b) \Rightarrow$
      $\forall \delta_2 \cdot \delta_2 \in \mathbb{R}^{+} \land \delta_1 \mapsto \delta_2 \in leq \Rightarrow \textbf{DeltaNeighborhood}(\delta_2, a, b)$
    ...
```

Listings 4.14: *ApproximationBase* Theory Excerpt

From this operator, the *Approximation* theory elaborates several useful operators, which are "lifted" versions of **DeltaNeighborhood**: comparison between a point and a function, between two functions, between a function and the solutions of a differential equation, between two differential equations. An extract of this theory is given in Listing 4.15.

Note that the comparison between two functions (**DeltaApproximation** operator) is the actual approximation relation that serves as the base for the approximate refinement studied in Chapter 7.

The theory also defines the **DeltaShrink**(δ, S) operator, corresponding to the $\mathcal{S}_{\delta}$ operator (Operator 13 of Section 4.2.2).

```
THEORY Approximation
IMPORT THEORY ApproximationBase
TYPE PARAMETERS E , F , F₁ , F₂ , UF₁ , UF₂
OPERATORS
  DeltaApproximation predicate (D: ℙ(E) , δ: ℝ⁺ , f₁: E ⇸ F , f₂: E ⇸ F )
    well−definedness condition $D \subseteq \mathrm{dom}(f_1) \land D \subseteq \mathrm{dom}(f_2)$
    direct definition $\forall x \cdot x \in D \Rightarrow \textbf{DeltaNeighborhood}(\delta, f_1(x), f_2(x))$
  DeltaShrink expression (δ: ℝ⁺ , S: ℙ(F))
    direct definition $\{y \mid y \in F \land \exists x \cdot x \in S \land \textbf{DeltaNeighborhood}(\delta, x, y)\}$
  ...
```

Listings 4.15: *Approximation* Theory Excerpt

Last, this theory of approximation gives a few axiomatic definition aimed at establishing approximation in particular cases, for instance using simulation functions (as used by Girard and Pappas [GP07; GJP08]).

The principle behind simulation functions is to find a function V that links two controlled autonomous ODEs (on a certain domain) and that is Lyapunov-stable. This function is called a *simulation function*, and allows deducing that said ODEs are δ-approximated from each other, i.e. their solutions are δ-approximated (with additional observation functions).

To model this, the **SimulationFunctions**$(D_{F_1}, D_{F_2}, D_{UF_1}, D_{UF_2}, f_1, f_2, g_1, g_2)$ operator is axiomatically defined, to represent the set of simulation functions between two autonomous controlled ODE characterised by $f_1 : F_1 \times UF_1 \nrightarrow F_1$ and $f_2 : F_2 \times UF_2 \nrightarrow F_2$, with domain $D_{F_1} \subseteq F_1$ (resp. $D_{F_2} \subseteq F_2$) and control domain $D_{UF_1} \subseteq UF_1$ (resp. $D_{UF_2} \subseteq UF_2$) and with appropriate observation functions $g_1 : F_1 \to F$ (resp. $g_2 : F_2 \to F$).

4.3.2.6 Domain Theories

Hybrid system models integrate continuous concepts that model physical phenomena. These physical phenomena, in addition to being constrained by continuous mathematics, are also usually constrained by physics itself (bounds, constants, etc.). Additionally, these phenomena are described using differential equations, and functionally close systems may define similar differential equations (e.g.: kinematics for moving objects, fluid mechanics for tanks and pipes, electricity theory for electronic circuits, ...).

Knowledge related to the particular systems to be modelled impact and enrich hybrid system models with constraints and specific reasonings. This knowledge is gathered in so-called *domain theories*. These theories extend the theory of differential equations (or approximation). They usually define special constants, generic shape of differential equations and control functions, as well as various axioms and theorems that can be used in conjunction with other theories, allowing establishing solvability, controllability, feasibility, and so on see Sections 4.3.2.4 and 4.3.2.5).

In the context of our work, we have proposed several domain theories to support our developments. They are described more thoroughly in the cases studies developed in this manuscript.

4.4 Discussion

Mathematical Background. The mathematical features defined in Sections 4.1 and 4.2 constitute the foundation of our framework. They are algebraically defined in *theories*, and allow the integration of continuous concepts to model continuous behaviours in an Event-B model, and handle them in the proofs.

Event-B is based on sound mathematics, and exploits the properties of these mathematics to define a correct-by-construction design method. Similarly, the formal definitions given in this chapter constitute the foundation of our framework. The properties of the defined operators allow building complex systems while mitigating the extra proof effort arising from the handling of advanced mathematical features, by providing generic proof patterns.

Theories. The theories presented in this chapter represent a substantial part of the technical work realised in this thesis, at least as important as the design of models (proof put apart).

Theorems are imported in the theories interactively, under the form of axioms. We assume that this operation is safe. Methods and tools have been proposed to check the correctness of this importation (e.g. Dedukti [Sai13]).

Chapter 5

A Generic Model for Hybrid Systems

In Chapter 3, we presented an overview of the formal framework for designing controller-plant loop hybrid system. The core of this framework is a *generic model*, an abstraction of controller-plant loop hybrid systems that encapsulates generic hybrid system features, such as time, discrete behaviours and continuous behaviours.

Concretely, this generic model takes the form of an Event-B machine that sets out the general structure and behaviour of controller-plant loop hybrid systems. It is based on continuous features, formalised in Chapter 4, and incorporated in Event-B using theories. These features are mainly used to handle continuous behaviours (time, continuous state variables, differential equations, etc.).

The model is expressed in a generic way using event parameters and abstract variables. The use of this model requires to instantiate it, that is, to refine it and to provide explicit substitutions (or additional constraints) to its parameters and variables, using witnesses and gluing invariants. Event-B then generates specific proof obligations associated with refinement, which ensure the consistency of the instantiation.

The generic model serves as an entry point for the use of the framework. It constraints hybrid system developments to enforce important properties, as well as to allow the use of the other formal tools. A number of proofs are realised, once and for all, at this generic level, and are not to be done again during development; this helps in segmenting and easing proofs in general.

In this chapter, we give a detailed presentation of the formalised generic model. We review the architecture of hybrid systems represented by this model in Section 5.1, while Section 5.2 develops its formulation in the form of an Event-B machine. Section 5.3 gives an overview the proofs associated with the developped Event-B model and how they are handled. Finally, Section 5.4 illustrates the use of the generic model to address two case studies taken from literature, and Section 5.5 concludes the chapter with a general discussion of the contribution.

5.1 Controller-Plant Loop Architectures

Hybrid systems may be structured in a number of different ways. One of the most common architecture is commonly described using the pattern shown in Figure 5.1. In this pattern, a

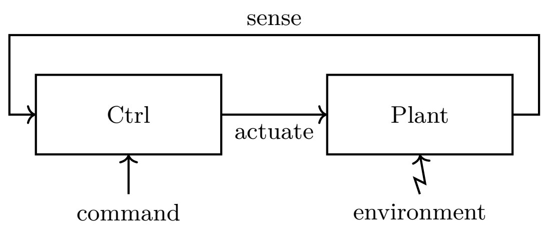


Figure 5.1: Hybrid System Global Architecture

(discrete) *controller* is linked to a (continuous) *plant* or, in other words, the physical phenomenon that the system shall control. The controller retrieves data from the plant via *sensors*, and is able to influence its behaviour via *actuators*. The controller may also change its state based on some internal calculus or user commands, and similarly the plant may change its behaviour because of its environment (wind, temperature, etc.).

We use this structure as a basis, together with the notion of *hybrid automata* [Alu+95] in order to devise a generic abstract model of hybrid systems, i.e. a generic model that represents any hybrid system conforming to this architecture. It is then possible to derive a specific hybrid system from this generic model through refinement.

This generic model revolves around four major concepts:

- An explicit modelling of **time**, as a positive real variable;
- **Discrete variables** to represent the state of the controller. These variables are used to model the general state of the controller, and in particular its *mode automaton* (the places of an hybrid automaton);
- **Continuous variables** to represent the state of the plant. These variables are defined as functions of time that follow differential equations (the behaviour is given in the places of the hybrid automaton);
- **Events** that model changes in both controllers and plants. Such events correspond to the arrows in Figure 5.1. They are classified as follows:
  - *Transition* events, for modelling changes to the internal controller and user commands (the user is considered as being part of the controller);
  - *Sensing* events for modelling sensor acquisitions and their effects on the controller's state;
  - *Behave* events to model the impacts of the environment on the plant;
  - *Actuate* events to model the changes in the plant induced by the controller (via actuators);

Note that, in the case of hybrid automata, transitions and sensing events correspond to the automaton's edges, with transition events close to synchronisation labels and sensing events close to guarded arrows. Behave and actuate events correspond to the content of the automaton's places (enactment of the continuous behaviour).

5.2 Generic Event-B Model

We base ourselves on the elements presented in Section 5.1, and on the mathematical features added to Event-B's expression language and presented in Chapter 4 and now detail the generic model we have designed for modelling controller-plant loop hybrid systems in Event-B. As stated before, this model consists of an Event-B machine, featuring a variable for time (denoted t), a discrete variable (denoted x_s) and a continuous variable (denoted x_p), as well as the four types of events discussed in Section 5.1.

5.2.1 Design Idea

The idea behind this model is to provide a generic structure that can be used as an entry point for designing any hybrid system. For this reason, the model features one event for every type of behaviour identified in Section 5.1 that essentially represents an abstract version of that type. These events are highly generic thanks to the use of *event parameters*, which can be instantiated in further refinements, using witnesses. These parameters are constrained by guards and, at instantiation, the property of guard strengthening will make sure that the concrete objects provided for the parameters are correct.

Similarly, the system revolves around abstract variables that ought to be substituted by the proper system variables using gluing invariants and witnesses.

By steering the design of hybrid systems with this generic model, we 1) force a formal structure for hybrid system, which makes it possible to prove its behaviour and 2) factorise a number of proofs that occur at the abstract level and would otherwise appear for any hybrid system.

5.2.2 Model

```
MACHINE Generic
VARIABLES $t$ , $x_s$ , $x_p$
INVARIANTS
   inv1 :  $t \in \mathbb{R}^+$
   inv2 :  $x_s \in \mathrm{STATES}$
   inv3 :  $x_p \in \mathbb{R}^+ \pfun S$
   inv4 :  $[0, t] \subseteq \mathrm{dom}(x_p)$
EVENTS
   INITIALISATION
   THEN
      act1 :  $t := 0$
      act2 :  $x_s :\in \mathrm{STATES}$
      act3 :  $x_p :\in \{0\} \to S$
   END
```

Listings 5.1: Generic Machine Header and Initialisation

Machine Header and Initialisation The machine header is given in Listing 5.1. It defines the three variables required to model hybrid systems, as well as the invariants constraining them. Time t is a variable valued in $\mathbb{R}^+$ (`inv1`). It is initially set to 0 (`act1`) arbitrarily: being homogeneous, taking $t = 0$ as its origin is convenient and does not introduce any restriction. The discrete variable

x_s evolves in a point-wise manner in set STATES (`inv2`), which corresponds to a generic encoding of a *mode automaton*; it is given an arbitrary value at initialisation (`act2`) using the non-deterministic set membership assignment operator ($:\in$).

Following the remarks of Section 5.1, the continuous state variable x_p is a function of time (`inv3`). To simplify later proofs, this function is partial on $\mathbb{R}$, and at least defined on $[0, t]$, i.e. from the origin of time to the present moment (`inv4`). At initialisation, it is given an arbitrary value, with the constraint that it shall be defined on $[0, 0] = \{0\}$ (`act3`).

```
Transition
ANY s
WHERE
   grd1 :  $s \in \mathbb{P}1(\text{STATES})$
THEN
   act1 :  $x_s :\in s$
END
```

Listings 5.2: Generic Transition Event

Transition Events Listing 5.2 gives the definition of transition events, the first variant of discrete events. They basically represent internal controller changes that are not caused directly by the plant: internal calculation, running timer, controller decision. They are also used to model a command issued by the user.

As a discrete event, `Transition` only modifies the current *discrete* state (`act1`). Because it does not relate to the plant's continuous state, the event is only guarded by a type predicate on its event parameter, and not by any reading of value.

```
Sense
ANY s , p
WHERE
   grd1 :  $s \in \mathbb{P}1(\text{STATES})$
   grd2 :  $p \in \mathbb{P}(\text{STATES} \times \mathbb{R} \times S)$
   grd3 :  $(x_s \mapsto t \mapsto x_p(t)) \in p$
THEN
   act1 :  $x_s :\in s$
END
```

Listings 5.3: Generic Sense Event

Sensing Events Sensing events, as shown in Listing 5.3, are the second kind of discrete events, and can be seen as refined transitions. They model any controller change that is induced by a change in the plant, detected through sensors. At this abstract level, sensing is considered to occur in an instantaneous way (when reaching continuous state C, the controller instantaneously changes its discrete state), but in later refinements, imprecision and delays may be introduced.

As a discrete event and just like `Transition`, `Sense` only modifies the *discrete* state of the controller (`act1`); but unlike for its counterpart, it observes both the current discrete state of the system and the continuous state of the plant. Thus, it is parameterised by a condition on these two kinds of variables, plus time (`grd2` and `grd3`).

```
Behave
ANY 𝒫 , t′
WHERE
   grd0 : $t' > t$
   grd1 : $\mathcal{P} \in (\mathbb{R}^+ \twoheadrightarrow S) \times (\mathbb{R}^+ \twoheadrightarrow S)$
   grd2 : $\textbf{Feasible}([t, t'], x_p, \mathcal{P}, \top)$
THEN
   act1 : $x_p :|_{t \to t'} \mathcal{P}(x_p, x_p') \;\&\; \top$
END
```

Listings 5.4: Generic Behave Event

Environment Events Environment events, given in Listing 5.4, are the first sort of continuous events. They represent changes in the plant that are not induced by the controller, but rather by the environment of the continuous phenomenon under control: gravity, wind, rain, etc.

As a continuous event, `Behave` modifies the *continuous* state of the system (i.e. the model's continuous variables). It does so using the continuous before-after predicate as described in Section 4.1.2 (`act1`). The well-definedness and feasibility of this action is enforced thanks to the **Feasible** predicate; when refining, guard strengthening will force the designer to establish this feasibility, effectively delegating the proof.

As being considered outside of (normal) control, the event is not initially impacted by the controller's discrete state; for this reason, its guards do not relate to the controller's state, and the local invariant given when using the $:|_{t \to t'}$ operator is $\top$ (meaning anything may happen).

Note that this event models situations where the controller may fall in a situation where it can no longer control the plant and is designed to model how the system responds to failure (closed-loop modelling). For systems where this issue is not to be addressed, it is always possible to disable this event during instantiation by setting its guard to $\bot$.

```
Actuate
ANY 𝒫 , s , H , t′
WHERE
   grd0 : $t' > t$
   grd1 : $\mathcal{P} \in (\mathbb{R}^+ \twoheadrightarrow S) \times (\mathbb{R}^+ \twoheadrightarrow S)$
   grd2 : $\textbf{Feasible}([t, t'], x_p, \mathcal{P}, H)$
   grd3 : $s \subseteq \text{STATES}$
   grd4 : $x_s \in s$
   grd5 : $H \subseteq S$
   grd6 : $x_p(t) \in H$
THEN
   act1 : $x_p :|_{t \to t'} \mathcal{P}(x_p, x_p') \;\&\; H$
END
```

Listings 5.5: Generic Actuate Event

Actuation Events Listing 5.5 gives the definition of actuation events, the second kind of continuous events. They model the changes induced on the plant by the controller, through actuators.

As for `Behave`, `Actuate` is a continuous event, and thus modifies the continuous state (x_p) of the system, using the continuous before-after predicate, as usual (`act1`). Note that, just like `Behave`, its unguarded counterpart, well-definedness and feasibility of the event are enforced by guard `grd2`.

The actuation being performed is directly impacted by the controller's discrete state; this fact is captured by the event's guards (`grd3` and `grd4`).

In general, such actuations are given using differential equations. In that case, this event is refined by replacing the $:|_{t \to t'}$ operator (and associated predicate $\mathcal{P}$) with the $:\sim_{t \to t'}$ operator (and a differential equation $eq \in \mathbf{DE}(S)$).

Unlike `Behave`, `Actuate` handles a special predicate/set, H, that prescribed the *evolution domain* or *local invariant* for a continuous before-after predicate (located right after the & symbol). H is given a type in `grd5`, while `grd6` asserts that, at time t (i.e. at the start of a continuous event), the continuous state already belongs to this domain.

5.2.3 Hybrid Events Semantics

Similarly to [Ban+15] with Hybrid Event-B and [Pla08] with hybrid programs, the semantics associated to hybrid models clearly distinguish between discrete and continuous events. Indeed, discrete events are timeless: they occur at a specific point in time and are instantaneous (i.e. do not have duration); this means, in particular, that a countable arbitrary number of such events may be triggered on any time interval. Conversely, continuous events – that use continuous before-after predicates – have a (strictly positive) duration.

After the initialisation, a continuous event runs until a discrete event is enabled. Such discrete events are preemptive and occur instantaneously, so that, as soon as they occur, a continuous event starts running. This protocol asserts that the controller is always able to trigger control actions in time to modify the plant's behaviour if needed, and that sensing never misses a change in the plant's state. Note that time is always progressing, meaning that there is always at least one continuous event enabled.

At some point, an environment event (`Behave`) may occur. This allows the designer to account for potential drifts, failures or even extreme conditions in which the system can find itself in (closed-loop modelling). They also allow addressing the requirement that a continuous event is always enabled: if no actuation is enabled, it generally means that the system cannot be controlled, and is thus described using this type of event.

Note that the duration of continuous events may be arbitrarily small, and even infinitely small, so that two discrete events are infinitely close. This may lead to a situation known as a *Zeno paradox*, where an infinite number of discrete event is triggered in a finite amount of time.

To avoid this situation, we constraints the semantics associated with our formalism to exclude traces containing infinite sequences of consecutive discrete events.

Note that these (implicit) assumptions shall be made explicit when discretising the system.

5.2.4 Using the Generic Model

The generic model serves as an entry point for a refinement chain. Any controller-plant loop hybrid system can be derived from it by refining it and by supplying it with is specific features in its generic parts (otherwise known as "instantiation"). In this section, we provide an overview of how to use the generic model to obtain a particular hybrid system model, and show how this model is at the core of the generic *method* and *framework* that we have devised for designing such systems.

To be able to model a hybrid system using this framework, it is first required to identify the important features of the system:

- **Continuous state**: the continuous variables characterising the system's plant (speed, position, temperature, etc.);
- **Discrete state**: the discrete variables used by the controller, and in general the controller's *mode automaton*;
- **System behaviour**: what the system does, in the form of guarded events and categorised according to the generic model, plus the system's **initialisation**;

Once this analysis is done, the generic model is used by providing concrete features for its variables and event parameters, using well-defined witnesses.

5.2.4.1 Continuous state

The main objective of hybrid systems is to control a physical, continuous phenomenon, generally requiring to keep track of its evolution and influencing it using actuators. This plant's physical quantities are typically what constitutes the system's continuous state.

The exact content of the continuous state depends on the hybrid system and in particular on how it is controlled and what is sensed from it, but on an abstract level we assume an exact and immediate reading (sensing) of the continuous phenomenon is possible, and thus that all the variables we need are accessible. Later refinements may introduce imprecision, latency and so on when approaching a more concrete implementation of the system.

The continuous variable's behaviour is usually described using a differential equation, which involves (some or all of) these variables, plus their derivatives. This means that, when modelling a hybrid system, we need to identify both its continuous variables and the laws that describe their respective behaviours, usually expressed in the form of differential equations.

The continuous state of the system is expressed as a set of functions of time,

$$\{x_{p,1} : \mathbb{R}^+ \to S_1, x_{p,2} : \mathbb{R}^+ \to S_2, \ldots, x_{p,k} : \mathbb{R}^+ \to S_k\}$$

where, for all i, S_i is the state space of function $x_{p,i}$ (the set containing the possible values of this function). In general, this state space is a real vector space, typically $S_i = \mathbb{R}^{m_i}$, with $m_i \in \mathbb{N}^*$.

On this basis, it is possible to express a generic relation between the abstract continuous variable x_p and its state space S, and the concrete continuous states and state spaces, and thus giving a general *gluing invariant* for the instantiation:

$$\begin{cases} S = S_1 \times S_2 \times \ldots \times S_k = \prod_i S_i \\ x_p = x_{p,1} \otimes x_{p,2} \otimes \ldots \otimes x_{p,k} = \bigotimes_i x_{p,i} \end{cases} \tag{5.1}$$

These equations state that 1) the system's overall state space is defined as the Cartesian product of all concrete state spaces (as denoted by the generalised Cartesian product $\prod$), and 2) the overall continuous state of the system is the *direct product* $\otimes$ (Operator 5) of each concrete continuous variable.

Note that, in order to be valid with regard to the generic machine's invariant, the following additional properties must be maintained:

$$\forall i, f_i \in \mathbb{R}^+ \nrightarrow S_i \wedge [0, t] \subseteq \mathrm{dom}(f_i)$$

5.2.4.2 Discrete state

The controller part of a hybrid system is, in essence, a discrete program; for this reason, it may handle variables, which are grouped under the denomination of *discrete state* of the system. Moreover, it is quite common to design hybrid system following the hybrid automata [Alu+95] formalism: the controller follows a set of states (the places of the automaton), each of which is associated to a continuous behaviour. During a run, this automaton may have its current state depending on sensing or other discrete transitions (the edges of the automaton).

The places of the hybrid automaton, together with its edges (without its hybrid parts) is a "standard" (discrete) automaton that essentially depicts the states of the controller and what may make it change. This automaton is called *mode automaton*, and is encoded *in extenso* in the discrete state of the model. In fact, every "type" of behaviour the system is capable of is called the *mode* of this system, and modes correspond to the places of the mode automaton.

When using the generic model, only the controller's mode is used. Any variable can be included in the generic discrete state, but since Event-B handles discrete variables "natively", this is not mandatory. Thus, linking the concrete hybrid model to the generic one only requires to populate the $STATES$ set containing the set of modes, and to keep the discrete variable x_s in the concrete machine.

Due to its nature, it is possible to use Event-B's partition operator to specify the content of the $STATES$ set.

5.2.4.3 Continuous Behaviour

The continuous behaviour of hybrid systems is described through the *actuate* and *behave* events. To model any such behaviour, it is thus required to refine one of these two events.

Behave events represent the environment, they should be used whenever modelling something that is out of the normal control of the system (e.g. possible failure/recovery mechanisms). As stated in Section 5.2.2, it is possible to disable this event by setting its guard to $\bot$. This makes it always disabled; it is a valid refinement since it abides by guard strengthening ($\vdash \bot \Rightarrow A$ for any A) and $\bot$ will be found as hypothesis of any related proof obligation (and $\bot \vdash A$ for any A following *ex falso quodlibet* or the *explosion* principle).

Actuation events implement the normal behaviour of a hybrid system. When refining this event, it is expected to instantiate the parameters of the event using witnesses.

Associated Mode For each actuation event, the event parameter s shall be replaced by the set of discrete states (or modes) in which this actuation can occur. If the system is deterministic, every actuation has a disjoint set of associated modes, but this is not mandatory.

Behavioural Equation The event parameter $\mathcal{P}$ shall be replaced by the actual predicate or specific differential equation associated with the actuation. Due to the way the abstract event has been written, providing a witness here will generate a WFIS (witness well-definedness) proof obligation: there must exist a combination of parameters (and in particular a t') such that $\mathcal{P}$ is feasible on $[t, t']$.

Local Invariant The event parameter H shall be replaced by a local invariant describing the limits of the current continuous section of the system. This set is typically filled-in with the negation of the potentially reachable sensing event's guards in order to enforce the fact that sensing events happen at the right time.

Gluing Invariant Preservation As the abstract continuous variable (x_p) handled by actuation and behave events disappears in the concrete model and is replaced by the set of functions $\{x_{p,1}, x_{p,2}, \dots, x_{p,k}\}$, Event-B suggests that a witness for the new value of x_p (x'_p) is provided, in order to prove the preservation of the gluing invariant.

This witness generally follows the form of the gluing invariant; typically, if the invariant is of the form $x_p \sim F(x_{p,1}, x_{p,2}, \dots, x_{p,k})$ (with F sa function and $\sim$ a relation such that refinement is correct, typically equality $=$), the witness for x'_p if of the form $x'_p \sim F(x'_{p1}, x'_{p2}, \dots, x'_{pk})$.

5.2.5 Formalisation

Although the model presented in this section is mathematically accurate, it is presented in a readable way, using formal abbreviations for abstracting Event-B notations.

For instance, $\mathbb{R}$ is modelled using the *RReal* abstract type defined in the *RReal* theory (see Section 4.3.2.2), and intervals such as $[0, t[$ are actually written using the interval operators of the *Interval* theory. For instance, *Closed2Open*$(Rzero, t)$ encodes interval $[0, t[$ (*closed* on the left to *open* on the right, with *Rzero* representing 0).

Continuous Assignment One of the most important differences however between the model shown above and the actual model written within Rodin lie in continuous assignments. We use the convenient shortcut $:|_{t \to t'}$, but we do not have the possibility to actually customise "language" operators such as assignment. That being said, thanks to the before-after predicate, it is enough to write a predicate in a $:|$ setting to actually encode this operator. Care has to be taken for the types of the different parameters though, to ensure well-definedness of the operators can be established.

Formally, an assignment of the form: $x_p :|_{t \to t'} \mathcal{P}(x_p, x'_p)\ \&\ H$ is encoded, in Event-B, as shown in Listing 5.6 (with *Pred* being $\mathcal{P}$, *tp* being t' in the parameters).

```
1  act1 : $t, x\_p :|$
2           $t = tp \wedge$
3           $x\_p' \in RReal \nrightarrow S \wedge$
4           $Closed2Closed(Rzero, t') \subseteq \mathrm{dom}(x\_p') \wedge$
5           $\mathbf{CBAP}(t, tp, x\_p, x\_p', Pred, H)$
```

Listings 5.6: Continuous BAP Encoding

In Listing 5.6, lines 3 and 4 are present for well-definedness purposes (without these statements, we would not be able to prove the PO associated with the use of **CBAP**). The "actual" encoding of $:|_{t\to t'}$ is indeed performed in line 2 (in order to actually set the time) and 5 for the core of the operator.

Feasibility The model uses a special predicate denoted **Feasible** to ensure that the $:|_{t\to t'}$ is feasible. In practice, this predicate is implemented using one of the **CBAP** $*$ **FIS**-type operators defined in Section 4.3.2.4.

The models presented throughout this thesis use these convenient mathematical notations, as to remain readable. Note that every model shown in Chapters 4 through 7 have been developed in classical Event-B, withing Rodin. They are given in Appendix B and can be accessed at the following link:
`https://irit.fr/~Guillaume.Dupont/models.php`.

5.3 Proofs

The generic model produced a total number of 25 proof obligations, which fall into three categories: well-definedness, invariant, and feasibility.

The main advantage of having a unique top-level model is that these proofs are made *once and for all.* In essence, they are the basis of correct hybrid systems, and the proofs that would typically be carried out every time otherwise.

5.3.1 Feasibility

Feasibility arises in actions that use Event-B's before-after predicate, to ensure that the described behaviour actually holds. At this level, feasibility is trivially established, as most of the invariants and guards exist for this sole purpose (e.g. **Feasible** in the actuation event).

Upon refining the generic model, the designer has to prove guard strengthening and witness well-definedness/feasibility (most of the time simpler than action feasibility POs). These properties, by virtue of the refinement and of the proofs done at the generic level, automatically entails the feasibility of the actions.

Establishing feasibility of a particular predicate in a refinement is completely tied to the nature of the predicate. When using differential equations, solvability needs to be asserted, using specific theorems such as the Cauchy-Lipschitz theorem, or any other property associated to the considered differential equation.

5.3.2 Invariants

The POs ensure the invariants are maintained by each event. They encompass the fundamental principle of induction: if given property holds at initialisation, and if, provided it holds at some point, it still hold no matter which event is being executed, then it holds on the whole model.

5.3.2.1 Proof in the Generic Model

The proofs at this level are classical. The hardest part is to establish `inv1`, `inv3` and `inv4` on continuous events, as they record the evolution of time t and update the continuous state x_p). The invariant PO for continuous event can be summarised as:

$$\begin{aligned} &\vdash t' \in \mathbb{R}^+ && (\mathit{TIME})\\ &\quad \land x'_p \in \mathbb{R}^+ \nrightarrow S && (\mathit{FUN})\\ &\quad \land [0,t'] \subseteq \mathrm{dom}(x'_p) && (\mathit{DOM}) \end{aligned}$$

Three parts have been identified in this PO, corresponding to each invariants.

Time Typing (TIME) This part, corresponding to `inv1` of Listing 5.1, is immediate: we have $t \in \mathbb{R}^+$, and so, by definition of $\mathbb{R}^+$: $t \in \mathbb{R}$ and $t \geq 0$. Additionally, we have $t' > t$ (given by TP, coming from the well-definedness of the $:|_{t \to t'}$ operator), and since $\geq$ is transitive, it follows that $t' \geq 0$. $t' \in \mathbb{R}$ is implied by the use of the operators, and it naturally follows that $t' \in \mathbb{R}^+$.

Functional Coherence (FUN) It corresponds to `inv3` of Listing 5.1 and represents the fact that x_p remains a *function* after the event, or in other words, that each value is mapped to a unique image:

$$\forall \hat{t} \in \mathbb{R}^+,\ x_1, x_2 \in S,\ \hat{t} \mapsto x_1 \in x_p \land \hat{t} \mapsto x_2 \in x_p \Rightarrow x_1 = x_2$$

This proof goal is trivially discharged by definition of the continuous before-after predicate (Operator 7): we build x_p incrementally and in a piece-wise manner, by appending functions defined on disjoint intervals ($[0,t[$ and $[t,t']$).

Domain Preservation (DOM) It corresponds to `inv4` of Listing 5.1 and is also immediate, as it follows the definition of the continuous before-after predicate operator (Operator 7): x'_p is defined on both domains $[0,t[$ and $[t,t']$, and we have $[0,t[\cup [t,t'] = [0,t']$; it follows that $[0,t'] \subseteq \mathrm{dom}(x'_p)$.

5.3.2.2 Induction on Continuous Events

The invariant proof obligation for continuous events highlights a schema that follows the structural induction principle, adapted to continuous behaviours and the way they are handled by the model. Let $\mathcal{I}(x_p)$ be an invariant on the continuous state x_p; the resulting invariant proof obligation on a continuous event with CBAP $x_p :|_{t \to t'} \mathcal{P}(x_p, x'_p) \mathbin{\&} H$ is:

$$\begin{aligned} \Gamma, \mathcal{I}([0,t] \lhd x_p), &\mathit{CBAP}(t, t', x_p, x'_p, \mathcal{P}, H)\\ &\vdash \mathcal{I}([0,t'] \lhd x'_p) \end{aligned}$$

Here Γ encompasses hypotheses originating from axioms and properties of the model, irrelevant to the formula.

By unfolding the CBAP operator and exposing its well-definedness properties, the following intermediate property arises:

$$\begin{aligned} \Gamma, \mathcal{I}([0,t] \lhd x_p), [0,t[\lhd x_p = [0,t[\lhd x'_p, \mathcal{P}([0,t] \lhd x_p, [t,t'] \lhd x'_p), \forall t^* \in [t,t'], x'_p(t^*) \in H,\\ \mathcal{I}([t,t'] \lhd x'_p) \vdash \mathcal{I}([0,t'] \lhd x'_p) \end{aligned} \tag{5.2}$$

The appended progression of the continuous state ($\mathcal{I}([t,t'] \lhd x'_p)$ corresponds to a restricted invariant preservation that only focuses on time interval $[t,t']$. Note that, additionally, the continuous state is not affected outside of this specific interval. It follows that the invariant proof obligation admits the following reformulation, more adapted to continuous events:

$$\begin{aligned}\Gamma, \mathcal{I}([0,t] \lhd x_p), &\mathcal{P}([0,t] \lhd x_p, [t,t'] \lhd x'_p), \forall t^* \in [t,t'], x'_p(t^*) \in H \\ &\vdash \mathcal{I}([t,t'] \lhd x'_p)\end{aligned} \tag{$CINV$}$$

By establishing Property *CINV*, then using Property 5.2, invariant preservation is proved.

The continuous invariant proof obligation *CINV* is another form of the invariant preservation one that takes advantage of the constraints of the model (i.e. using the CBAP operator in continuous events) to remove unnecessary proof steps that are true by construction and, in particular, the fact that the invariant holds on $[0,t[$ (the continuous state's "past").

This updated proof obligation is an induction step: if the invariant holds for x_p on $[0,t]$, then, knowing that x'_p is derived from x_p using $\mathcal{P}$ and that it remains in evolution domain H, then prove that the invariant still holds for x'_p on $[t,t']$. Doing so, it follows that the invariant holds for x'_p on the entirety of $[0,t']$.

On a side note, we can see that one way of establishing *CINV* is to demonstrate that H is stronger than $\mathcal{I}$. Note, however, that this is not always possible since $\mathcal{I}$ is a predicate on functions, and H a predicate on points.

5.3.3 Event Feasibility

Ideally, in a safe hybrid system, the actuation corresponding to the current discrete state/mode is always feasible; otherwise, the system is not controllable. This requirement is explicitly handled.

The main difficulty for this PO is to prove that guard `grd2` of actuation events is feasible; this is written:

$$\Gamma \vdash \exists t' \cdot t' \in \mathbb{R}^+ \wedge t' > t \wedge \mathbf{Feasible}([t,t'], x_p, \mathcal{P}, H)$$

By unfolding **Feasible**, we obtain:

$$\begin{aligned}\Gamma \vdash \exists t' \cdot t' \in \mathbb{R}^+ \wedge t' > t \wedge \\ (\exists x'_p \cdot x'_p \in \mathbb{R} \mathrel{+\!\!\!\rightarrow} S \wedge [t,t'] \subseteq \mathrm{dom}(x'_p) \wedge \\ \mathcal{P}([0,t] \lhd x_p, [t,t'] \lhd x'_p) \wedge \\ x'_p \underset{[t,t']}{\in} H \\ )\end{aligned}$$

This PO has to be proven on a case-by-case basis; it requires in-depth study of the involved predicate (or equation). However, when S is a topological space (which is entailed if $S = \mathbb{R}^n$) and when H has some specific properties, it can be simplified as follows.

Theorem 4 (Simplified Feasibility)**.** *Let S be a topological space and $t \in \mathbb{R}^+$. Let $x_p \in \mathbb{R} \mathrel{+\!\!\!\rightarrow} S$ a continuous function with $[0,t] \subseteq \mathrm{dom}(x_p)$. Let $\mathcal{P} \in (\mathbb{R} \mathrel{+\!\!\!\rightarrow} S) \times (\mathbb{R} \mathrel{+\!\!\!\rightarrow} S)$ and $H \subseteq S$, with $x_p(t) \in H$. If there exists $\hat{t} \in \mathbb{R}^+$ with $\hat{t} > t$ such that:*

$$\exists \hat{x}_p \cdot \hat{x}_p \in \mathbb{R} \mathrel{+\!\!\!\rightarrow} S \wedge [t,\hat{t}] \subseteq \mathrm{dom}(x_p) \wedge \mathcal{P}([0,t] \lhd x_p, [t,\hat{t}] \lhd \hat{x}_p)$$

*with $\hat{x}_p$ continuous on $[t,\hat{t}]$, and if H is **open**, then there exists $t' \in \mathbb{R}^+$, $t < t' < \hat{t}$, such that* **Feasible**$([t,t'], x_p, \mathcal{P}, H)$ *is true.*

Proof. **Preliminary notions:** the proof relies on the property that H is *open*. For the record, a set is open if and only if it is a neighbourhood of each of its point:

$$\forall x \in H, H \in \mathcal{N}(x)$$

where $\mathcal{N}(x)$ is the set of neighbourhoods of point x.
We also recall that a function f is continuous at point t if, for any neighbourhood W of $f(t)$, $f^{-1}[W]$ is a neighbourhood of t (where f^{-1} is the inverse of function f, see Operator 3).

Suppose that there exist $\hat{t}$ and $\hat{x}_p$ as in the hypotheses of the theorem. We have $\hat{x}_p(t) \in H$ by hypothesis, so we have $H \in \mathcal{N}(\eta(t))$. Because $\hat{x}_p$ is continuous in t, we obtain $\hat{x}_p^{-1}[H] \in \mathcal{N}(t)$. Since t is real, $\exists \mu > 0$ such that $[t, t+\mu[\subseteq \hat{x}_p^{-1}[H]$.
We take $t' \in \mathbb{R}$ such that $t < t'$, $t' < t + \mu$ and $t' < \hat{t}$. We observe that $[t, t'] \subseteq [t, t+\mu[$, and thus that $[t, t'] \subseteq \hat{x}_p^{-1}[H]$. By unfolding the definition of the inverse (Operator 3), we deduce that: $\forall t^* \in [t, t'], \hat{x}_p(t^*) \in H$.

Additionally, we have $[t, \hat{t}] \subseteq \mathrm{dom}(x_p)$ and $\mathcal{P}([0,t] \lhd x_p, [t, \hat{t}] \lhd \hat{x}_p)$ by hypothesis. By restriction, since $[t, t'] \subseteq [t, \hat{t}]$, we deduce that $[t, t'] \subseteq \mathrm{dom}(x_p)$ and thus that $\mathcal{P}([0,t] \lhd x_p, [t, t'] \lhd \hat{x}_p)$. ☐

The intuition behind this theorem is that, if $\mathcal{P}$ is feasible and H is open, it is always possible to find $t' > t$ (possibly very close to t) such that the action is feasible. In practice, this means that, as long as H is open, we only need to prove the feasibility of $\mathcal{P}$ and the continuity of any function obtained with it.

When using differential equations, function continuity is obtained for free (by definition of differential equations); what remains to do is to establish the solvability of the equation.

5.4 Case Study

In this section, we illustrate the use of the generic model by using it to design two models of hybrid systems, borrowed from literature.

The first one, taken from [Que+16], address the problem of car that must brake automatically in order to stop before a given point. The second one, taken from [Aré+12], proposes to model a controller to assist a car attempting to perform a signalised left-turn.

Note that to demonstrate our approach, we chose two case studies that were both developed using dynamic logic/hybrid programs and the KeYmaera X environment. We believe that KeYmaera is a widely used proof-based modelling technique, and it is especially relevant to compare that approach to ours.

5.4.1 First System: Automatic Brake

This case study is borrowed from [Que+16], where it was modelled as a hybrid program. Multiple properties have been expressed using differential dynamic logic, and proved using KeYmaera.

Figure 5.8 depicts a scenario for this system: a car shall stop before a given point (the *stopping point*, denoted *SP*). To achieve this goal, it is required to design a correct controller, able to influence the car's acceleration.

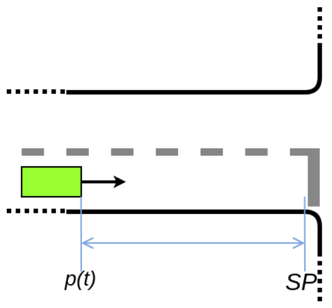


Figure 5.8: Automatic Brake Typical Scenario

5.4.1.1 Preliminary Study

Plant Description. The controlled plant is the car, modelled by its position, velocity (speed) and acceleration (p, v and a respectively). The car's speed is physically bounded between 0 and V_{max}. Acceleration is controlled directly. As a matter of simplification, it evolves *discretely* between the values 0, $-b$ and A, where $b > 0$ is car's braking power and $A > 0$ is the car's (forward) acceleration. We denote by SP the position of the stopping sign (see Figure 5.8).

The laws of physics provide a differential equation of the form:

$$\forall t \in \mathbb{R}^+, \dot{v}(t) = a, \dot{p}(t) = v(t) \tag{5.3}$$

or, in ODE form: $\Phi(t, [\,v\,p\,]^\top) = [\,a\,v\,]^\top$.

The goal of the system is to stop the car before reaching position SP. Physics gives the expression of the distance needed by the car to reach $v = 0$ from its current velocity $v(t)$ at time t using a braking force b:

$$\forall t \in \mathbb{R}^+, d(t) = \frac{v(t)^2}{2b} \tag{5.4}$$

As the car is required to stop before the stopping point (i.e. reach $v = 0$ with $p < SP$), we can hence determine when the car is required to start braking, or rather when it is safe for the car not to brake. This safety property is formalised as so:

$$\forall t \in \mathbb{R}^+, \mathit{safe} \Leftrightarrow p(t) + d(t) < SP \tag{5.5}$$

If ever *safe* evaluates to false, the car is required to brake.

Controller Description. The car's controller behaves in five modes, as shown in Figure 5.9, in the form of a hybrid automaton (see Section 1.1.1). These modes are in two categories.

1. *Free behaviour:* whenever *safe* is true, the car chooses freely one of the three following modes.
    - **Stabilising:** $a = 0$, the car's speed does not change.
    - **Accelerating:** $a = A$, the car's speed increases while remaining below V_{max}.
    - **Braking:** $a = -b$, the car's speed decreases while remaining above 0.
2. *Restricted behaviour:* whenever *safe* is false, the car obeys the following rules.

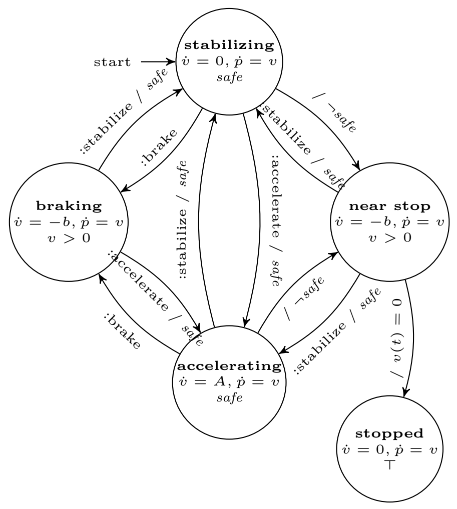


Figure 5.9: Hybrid Automaton for the System

- **Near stop:** the car is braking ($a = -b$); the car falls into this mode as soon as it approaches the stopping point.
- **Stopped:** the car stopped ($a = 0$, $v = 0$) while in **near stop** mode, presumably right before *SP*.

The controller may choose to change its mode among *stabilizing*, *braking* and *accelerating* via **transition events** as long as it is safe to do so. The system moves from *free behaviour* to *restricted behaviour* (hence entering *near stop* mode) via a **sense event** that detects if *SP* is too close, i.e. when the remaining distance between the car and *SP* is shorter than the distance it needs to brake.

Similarly, it moves from *near stop* mode to *stopped* mode via another **sense event** when it detects that the car has stopped. The system's behaviour is summed up in the hybrid automaton of Figure 5.9.

Requirements. The system's requirement can be summarised as follows:
FUN1 The car moves forward, and the controller controls its acceleration a;
ENV1 Speed always stays between 0 and V_{max};
SAF1 Whenever the car is stopped ($x_s =$ *stopped*), its position is before the stopping point (i.e. $p(t) <$ *SP*);

5.4.1.2 Event-B Development

Based on our preliminary study, we give an Event-B development fulfilling the requirements of the system. This development is based on the generic model presented in this chapter, and we use Event-B's refinement to instantiate it.

Constants and Axioms. The system requires the definition of multiple constants and properties that are contained in an Event-B *context* (later referred to as `Car_C1`). In particular, this context defines the acceleration A, braking power b, maximum speed V_{max} and initial speed of the car v_0. A set of discrete states `STATES` is introduced according to the mode automaton given in Figure 5.9. Moreover, this context also contains the definition of the ordinary differential equations corresponding to each possible continuous behaviour (as the functions characterising them):

- f_{stable} when $a = 0$ (in *stabilising* or *stopped* mode)
- $f_{acceleration}$ when $a = A$ (in *accelerating* mode)
- $f_{deceleration}$ when $a = -b$ (in *decelerating* or *near stop* mode)

These ODE functions are also accompanied with various properties regarding their regularity, required to prove they admit solutions later on.

```
MACHINE Car_M1 REFINES Generic SEES
    Car_C1
VARIABLES t , x_s , v , p
INVARIANTS
    inv1−2 : $v \in \mathbb{R}^+ \nrightarrow \mathbb{R}, p \in \mathbb{R}^+ \nrightarrow \mathbb{R}$
    inv3−4 : $[0,t] \subseteq \mathrm{dom}(v), [0,t] \subseteq \mathrm{dom}(p)$
    inv5 : $x_p = [\, v\; p\,]^\top$
    inv6 : $\forall \hat{t} \cdot \hat{t} \in \mathbb{R}^+ \wedge x_s = stopped \Rightarrow p(\hat{t}) \leq SP$
```

```
INITIALISATION
WITH $x_p'$ : $x_p' = [\, v'\; p'\,]^\top$
THEN
    act1 : $t := 0$
    act2 : $x_s := stabilizing$
    act3 : $v, p := \{0 \mapsto v_0\}, \{0 \mapsto 0\}$
END
```

Listings 5.7: Machine Header and Initialisation

Machine Header. Listing 5.7 excerpts the header of the machine and its initialisation. As expected the machine refines the generic one. The state space $S = \mathbb{R}^2$ is given as the machine deals with two continuous variables (speed v and position p). The abstract continuous state x_p is refined by the two-dimensional vector $[\, v\; p\,]^\top$ given in the gluing invariant `inv5`. `inv6` describes the system's *safety invariant*, encoding the property **SAF1**.

Initialisation is straightforward: the car starts at position 0 with an initial speed set to v_0, and the controller is initially in *stabilizing* mode.

```
ctrl_transition_accelerate
  REFINES Transition
WHERE
    grd1 : $p(t) + \frac{v(t)^2}{2b} < SP$
WITH
    s : $s = \{accelerating\}$
THEN
    act1 : $x_s := accelerating$
END
```

```
ctrl_sense_near_stop REFINES Sense
WHERE
    grd1 : $p(t) + \frac{v(t)^2}{2b} \geq SP$
    grd2 : $v(t) > 0$
WITH
    s : $s = \{nearing\_stop\}$
    p : $p = STATES \times \mathbb{R} \times \{\hat{v}, \hat{p} \mid \hat{p} + \frac{\hat{v}^2}{2b} \geq SP \wedge \hat{v} > 0\}$
THEN
    act1 : $x_s := nearing\_stop$
END
```

Listings 5.8: Machine Transition and Sensing Events

Transition and Sensing Events. Listing 5.8 shows a transition and sensing events taken from the system. The transition corresponds to the decision (most likely made by the user) to accelerate. It is guarded by the safety invariant to ensure that the system does not involve an unsafe behaviour. A witness for s (set of possible destination states) is provided.

The `ctrl_sense_near_stop` sensing event detects when the car is approaching the stopping sign and needs to brake. This instantaneous event is guarded by $\neg safe$ so as to be triggered when the controller detects that the car is entering an unsafe area. The actions of this sensing event enables an actuate event by setting the controller in *nearing stop* mode. Again, a witness is provided for s, as well as for p; the latter reflects the associated guard on the continuous state.

```
Behave REFINES Behave
ANY $eq$
WHERE
    grd1 : $eq \in \mathbf{DE}(S)$
    grd2 : $\mathbf{Solvable}([t, t'], eq)$
WITH
    $x_p'$ : $x_p' = [\, v' \; p' \,]^\top$
THEN
    act1 : $v, p, t :\sim_{t \to t'} eq \;\&\; \top$
END
```

```
ctrl_actuate_brake REFINES Actuate
ANY $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $x_s \in \{braking, nearing_s top\}$
WITH
    $eq$ : $eq = \mathbf{ode}(f_{deceleration}, [\, v(t) \; p(t) \,]^\top, t)$
    $s$ : $s = \{braking, nearing\_stop\}$
    $x_p'$ : $x_p' = [\, v' \; p' \,]^\top$
    $H$ : $H = \{v^*, p^* \mid v^* > 0\}$
THEN
    act1 : $v, p :\sim_{t \to t'} \mathbf{ode}(f_{deceleration}, [\, v(t) \; p(t) \,]^\top, t)$
        $\&\{\hat{v}, \hat{p} \mid \hat{v} > 0\}$
END
```

Listings 5.9: Machine Behave and Actuation

Behave and Actuation Events. Listing 5.9 shows the machine's *behave* event as well as an actuation event. The former is a direct refinement of the abstract *behave* event, replacing the abstract continuous state x_p with the concrete one $[\, v \; p \,]^\top$. The witness provided for x_p' follows the gluing invariant.

The presented actuation is tied to the *braking* and *nearing stop* mode, and causes changes in the car behaviour, so that it starts braking. In this refined event, the usual witnesses are provided for x_p' and s (i.e.: the concrete continuous state and the discrete state triggering the actuation). The relevant continuous behaviour for the system is given using an ordinary differential equation and the **ode** operator. We also give a value to the evolution domain (H) that ensures that speed remains positive.

5.4.1.3 Proofs

This system has yielded a number of 118 proof obligations. Around 27% of them result from well-definedness (in particular from operators defined in theories, used in the model). 26% are generated from invariants, most of which are actually fairly easy to prove (typically proving invariants). Refinement-related proofs make for 38% of the POs, and include both simulation and guard strengthening (i.e. making sure the provided witnesses are compatible with the system).

The most difficult proof to carry out is to establish the solvability of the provided ODEs (required by guard strengthening). In this case, we use the Cauchy-Lipschitz theorem (Theorem 3) to exploit the form of these ODEs and to establish the property.

5.4.2 Second System: Signalised Left-Turn Assist

This second case study is borrowed from [Aré+12], where it is again addressed using hybrid programs, and by proving properties expressed using differential dynamic logic using KeYmaera.

The problem is summed up in Figure 5.13: a car is trying to perform a left-turn without being protected by traffic lights (so-called *signalised* left-turn). To do so, it must cross the lane of opposite direction; if a car was to move on the said lane, this could lead to a collision. The goal is to design the correct controller to help the car to perform the left-turn safely.

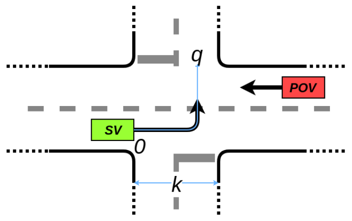


Figure 5.13: Automatic Brake Typical Scenario

5.4.2.1 Preliminary Study

Plant Description. The controlled plant is the car performing a turn and called *subject vehicle* (or *SV*), modelled by its *curvilinear abscissa*[1], speed and acceleration (p_{SV}, v_{SV} and a_{SV}, respectively). The car's speed is physically bounded between 0 and V_{max} and its acceleration is also bounded between $-b$ and A_{max}, where $b > 0$ is the maximum braking power. Changes in the acceleration are discrete and are controlled by the system. We note k the width of the intersection and q the length of the turn. Note that the axes of the frame are aligned in a standard way (upward and rightward) and the origin is positioned at the beginning of the turn.

As the model needs to determine whether or not there is a collision, we also model the first car on the lane in the opposite direction (i.e. the one that may hit the SV) called *primary object vehicle* (or *POV*). This car is merely "simulated", and we are only interested in its position and speed (p_{POV} and v_{POV}, respectively). The speed of the POV is also physically bounded between 0 and V_{max} and evolves discretely and randomly in the model.

Physics yields the following equations for the SV's dynamics:

$$\forall t \in \mathbb{R}^+, \begin{cases} \dot{v}_{SV}(t) = a_{SV} \\ \dot{p}_{SV}(t) = v_{SV}(t) \end{cases} \tag{5.6}$$

and for the POV's dynamics:

$$\forall t \in \mathbb{R}^+, \dot{p}_{POV}(t) = v_{POV}(t) \tag{5.7}$$

These equations are put together to form the overall system's dynamics. This can be expressed in the form of an ODE:

$$\Phi(t, [\, v_{SV}\; p_{SV}\; p_{POV} \,]^\top) = [\, a_{SV}\; v_{SV}\; v_{POV} \,]^\top$$

[1]That is, we do not model x and y coordinates but rather the position of the object on its supposed trajectory

Note that a negative acceleration simply means the car is *braking* (not moving in backward direction). Hence, when v_{SV} reaches 0 while braking, it does not go below. It is also important to note that the POV is moving *from right to left* (backward in relation to the x axis), meaning that its speed is negative.

The point of the system is to evaluate whether the SV has the time to attempt a turn. We want to establish safety in the worst case scenario; that is: the SV turns with minimal acceleration A_{min} and the POV drives at maximum speed V_{max}. Physics allows us to calculate the time needed for the SV to complete its left turn starting from time t, given its position $p_{SV}(t)$, speed $v_{SV}(t)$, acceleration a_{SV} and the length of the turn q as:

$$\forall t \in \mathbb{R}^+,\ T_{SV}(t) = \frac{-v_{SV} + \sqrt{v_{SV}^2 + 2a_{SV}(q - p_{SV})}}{a_{SV}} \tag{5.8}$$

Similarly, we can calculate the time for the POV to reach the intersection given its position $p_{SV}(t)$ and k at the end of the intersection (or at the beginning as for the POV's point of view), assuming it is driving at maximum speed V_{max}:

$$\forall t \in \mathbb{R}^+,\ T_{POV}(t) = \frac{p_{POV} - k}{V_{max}} \tag{5.9}$$

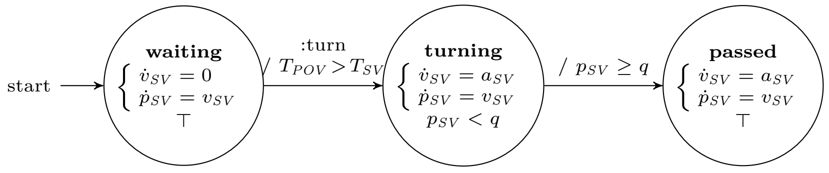
Hence, a sound condition for determining whether the SV has enough time to complete its turn before being hit by the POV is defined by the following property:

$$\mathit{safe} \Leftrightarrow T_{POV}(t) > T_{SV}(t) \tag{5.10}$$

Controller Description. The car's controller acts on the SV only. It operates in three modes:

- **Waiting:** $a_{SV} = 0$, the car is waiting for turning.
- **Turning:** $a_{SV} \in [A_{min}, A_{max}]$, the car is completing its turn.
- **Passed:** $a_{SV} \in [-b, A_{max}]$, the car has completed its turn and is free to move.

Note that passing from *waiting* to *turning* is not mandatory: the controller does so via a *transition* event involving a choice (generally a choice from the user). Conversely, the controller moves from *turning* mode to *passed* mode "automatically", simply by reading its position and comparing it to the end of the turn. The system's behaviour is summed up in the hybrid automaton of Figure 5.14.



start
waiting
$\begin{cases}\dot{v}_{SV} = 0 \\ \dot{p}_{SV} = v_{SV}\end{cases}$
$\top$
:turn
/ $T_{POV} > T_{SV}$
turning
$\begin{cases}\dot{v}_{SV} = a_{SV} \\ \dot{p}_{SV} = v_{SV}\end{cases}$
$p_{SV} < q$
/ $p_{SV} \geq q$
passed
$\begin{cases}\dot{v}_{SV} = a_{SV} \\ \dot{p}_{SV} = v_{SV}\end{cases}$
$\top$

Figure 5.14: Hybrid Automaton for the System

Requirements. The system's requirements are summarised as:

FUN1 The car waits on its lane;

FUN2 At some point, the car starts and completes a left-turn;

ENV1 Another vehicle (POV) travels the lane of opposite direction with a random, bounded speed;

SAF1 If the POV reaches the intersection ($p_{POV} < k$), the SV has either completed its turn ($p_{SV} > q$) or has not moved ($p_{SV} \leq 0$);

5.4.2.2 Event-B Development

Based on this study, we give here a possible Event-B development addressing this case study, and refining the generic model.

Constants and Axioms. In the same way as for our previous case study, we need to define a set of constants and axioms in an Event-B Context (named `LeftTurnAssistCtx`), used throughout the model. In particular, this context defines initial SV acceleration a^0_{SV}, POV speed v^0_{POV}, POV position p^0_{POV}, maximum braking power b, maximum acceleration A_{max} and minimal acceleration A_{min} for attempting the left turn. A set of discrete states `STATES` is defined similar to the mode automaton given in Figure 5.14. Moreover, the defined context also contains the ordinary differential equations for each continuous behaviour of the system as follows:

- f_{stable} when $a_{SV} = 0$ (in *waiting mode*).
- $f_{acceleration}$ when $a_{SV} > 0$ the new acceleration of the SV (in *turning mode*).
- $f_{deceleration}$ with $a_{SV} < 0$ the new (braking) acceleration of the SV (possibly in *passed mode* when the car is not constrained anymore).

Various properties are defined for these ODE functions. They are used to prove solutions existence.

```
MACHINE LeftTurnAssist REFINES Generic
SEES LeftTurnAssistCtx
VARIABLES $t$, $x_s$, $p_{POV}$, $p_{SV}$, $v_{SV}$, $v_{POV}$, $a_{SV}$
INVARIANTS
    inv1−3 : $p_{POV} \in \mathbb{R}^+ \rightarrow\!\!\!\!\!\!+ \mathbb{R}$,
             $p_{SV} \in \mathbb{R}^+ \rightarrow\!\!\!\!\!\!+ \mathbb{R}, v_{SV} \in \mathbb{R}^+ \rightarrow\!\!\!\!\!\!+ \mathbb{R}$
    inv4−7 : $[0, t] \subseteq \mathrm{dom}(p_{POV})$,
             $[0, t] \subseteq \mathrm{dom}(p_{SV}), [0, t] \subseteq \mathrm{dom}(v_{SV})$
    inv7 : $v_{POV} \in [-V_{max}, 0]$
    inv8 : $a_{SV} \in [-b, A_{max}]$
    inv9 : $x_p = [\, v_{SV}\; p_{SV}\; p_{POV} \,]^\top$
    inv10 : $\forall \hat{t} \cdot \hat{t} \in \mathbb{R}^+ \wedge p_{POV}(\hat{t}) < k$
              $\Rightarrow (p_{SV}(\hat{t}) \leq 0 \vee p_{SV}(\hat{t}) \geq q)$
```

```
INITIALISATION
WITH
    $x'_p$ : $x'_p = [\, v'_{SV}\; p'_{SV}\; p'_{POV} \,]^\top$
THEN
    act1 : $t := 0$
    act2 : $v_{SV}, p_{SV}, p_{POV} :=$
        $\{0 \mapsto 0\}, \{0 \mapsto 0\}, \{0 \mapsto p^0_{POV}\}$
    act3 : $x_s := waiting$
    act4 : $a_{SV}, v_{POV} := 0, v^0_{POV}$
END
```

Listings 5.10: Machine Header and Initialisation

Machine Header. Listing 5.10 gives the machine's header and initialisation event. Here, the state space is $S = \mathbb{R}^3$ as the controller deals with three continuous variables: SV speed v_{SV} and position

p_{SV}, and POV position p_{POV}. We also need to define the SV acceleration a_{SV} (i.e. actuated by the controller) and the POV speed v_{POV}, which is used in the POV differential equation.

The abstract continuous state x_p is refined by the 3-dimensional vector $[\, v_{SV}\, p_{SV}\, p_{POV}\,]^\top$ as given in the gluing invariant `inv9`. Finally, the fundamental system's safety invariant ensuring a safe left turn is given in `inv10`. It is a direct translation of the safety property given in the requirements.

Initialisation sets the controller in *waiting mode* according to the hybrid system's automaton. It also sets the other state variables of the system to their respective initial values. In particular, it sets the initial value of the continuous state variables as being the solutions of the ODE defined by f_{stable}. A witness is thus provided through derivation of the gluing invariant for the variable x_p' (new abstract continuous state).

```
ctrl_transition_attempt_turn
  REFINES Transition
WHERE
  grd1 : $x_s = waiting$
  grd2 : $T_{SV}(t) < T_{POV}(t)$
WITH
  s : $s = \{turning\}$
THEN
  act1 : $x_s := turning$
END
```

```
ctrl_sense_turn_end REFINES Sense
WHERE
  grd1 : $p_{SV}(t) \geq q$
WITH
  s : $s = \{passed\}$
  p : $p =$
      $STATES \times \mathbb{R} \times \{\hat{v}_{SV}, \hat{p}_{SV}, \hat{p}_{POV} \mid \hat{p}_{SV} \geq q\}$
THEN
  act1 : $x_s := passed$
END
```

Listings 5.11: Machine Transition and Sensing Events

Transition and Sensing Events. Listing 5.11 gives a transition and a sensing event of the machine. The transition corresponds to the *waiting–turning* arrow of the mode automaton, or in other words to the decision of attempting a left-turn. It occurs only if the system estimates it is safe to turn (enforced by guard `grd2`). A witness is provided for s, following the mode automaton.

The sensing event detects when the car has finished its turn (and goes in *passed* mode). Note how this event is guarded by a predicate on the continuous state (`grd1`) to represent the actual sensing. Witnesses are provided for s and p, the latter reflecting the event guards.

Behave and Actuation Events. Figure 5.12 depicts the system's *behave* event, as well as one of its actuation events. As usual, *behave* models environmental changes; this includes **ENV1** in the requirements, i.e. the changes of speed of the POV (`act2`).

The actuation follows classical actuation events; here, it represents the behaviour of the car when turning. The continuous state evolves according to the given differential equation (under the form of an ODE), constrained by an evolution domain. A value for a_{SV} is given: it represents the command issued by the controller. Witnesses are provided for x_p' following the gluing invariant, as well as for H and eq, where they represent the actual parameters of the model.

5.4.2.3 Proofs

This model yielded a number of 88 proof obligations. Well-definedness put aside, the majority of these POs relate to refinement and invariant preservation (32% each). The latter consists of a lot of

```
Behave REFINES Behave
ANY $eq$ , $v$ , $t'$
WHERE
    grd1 : $eq \in \mathbf{DE}(S)$
    grd2 : $\mathbf{Solvable}([t, t'], eq)$
    grd3 : $v \in [0, V_{max}]$
WITH
    $x'_p$ : $x'_p = [\, v'_{SV}\; p'_{SV}\; p'_{POV} \,]^\top$
THEN
    act1 : $v_{SV}, p_{SV}, p_{POV} :\sim_{t \to t'} eq \,\&\, \top$
    act2 : $v_{POV} := -v$
END
```

```
ctrl_actuate_turning REFINES Actuate
ANY $a$ , $t'$
WHERE
    grd1 : $x_s = turning$
    grd2 : $a \in [A_{min}, A_{max}]$
WITH
    $eq$ : $eq =$
        $\mathbf{ode}(f_{acceleration}, [\, v_{SV}(t)\; p_{SV}(t)\; p_{POV}(t) \,]^\top, t)$
    $s$ : $s = \{turning\}$
    $x'_p$ : $x'_p = [\, v'_{SV}\; p'_{SV}\; p'_{POV} \,]^\top$
    $H = \{v^*_{SV}, p^*_{SV}, p^*_{POV} \mid p^*_{SV} < q\}$
THEN
    act1 : $v_{SV}, p_{SV}, p_{POV} :\sim_{t \to t'}$
            $\mathbf{ode}(f_{acceleration}, [\, v_{SV}(t)\; p_{SV}(t)\; p_{POV}(t) \,]^\top, t)$
            $\&\{\hat{v}_{SV}, \hat{p}_{SV}, \hat{p}_{POV} \mid \hat{p}_{SV} < q\}$
    act2 : $a_{SV} := a$
END
```

Listings 5.12: Machine Behave and Actuation Events

typing invariant establishment, trivial to discharge. The hardest invariant proofs relate to invariant `inv10` (system's safety), particularly in the specific part where the car is turning.

Refinement-related proofs mainly consist in ensuring the witnesses provided for the generic model's parameters are coherent (e.g. solvable differential equations).

5.5 Discussion

The generic model is the entry point of the framework: controller-plant loop hybrid systems are built first by refining it. This generic model has been designed to be both restrictive and flexible; guards and witnesses enforce, in a way, a "correct" use of the model with proper separation of discrete and continuous behaviours on the one hand, and on the other hand, the highly generic level of the model's features (variables and parameters) allows a designer to refine the model in many different ways, corresponding to a large set of controller-plant hybrid systems.

Major proofs are realised, once and for all, at the generic model level, and the only remaining proofs relate to the instantiation of the model: simulation, guard strengthening, feasibility and witness proofs. This way, when the generic model is instantiated for a particular hybrid system, the resulting refinement must preserve the properties of the generic model (feasibility of actions/solvability of equations); this ensures the system fulfils the core requirements of a hybrid system.

The generic model is used for all of our work. It was first published in [Dup+18b], where it was successfully applied to the stopping car case under study, presented in Section 5.4.1. It was also used to address the signalised left-turn assist case study [Dup+18a], described in Section 5.4.1. Finally, [Sta+19] shows how this applies to formalising parts of the European Train Control System (ETCS).

Models for both cases studies as well as the generic model are given in their entirety in Appendix B. The generic model is given in Section B.1. The models for the automatic brake case study are given in Section B.2, and for the SLTA in Section B.3.

Chapter 6

Architecture Patterns

In the context of this thesis, we are interested in controller-plant loop hybrid systems. This specific type of systems consists of two components: a controller and a physical plant under control, communicating with each other via sensors and actuators. This layout can be extended, allowing for more complex hybrid systems, with more components.

Typically, a single controller may be able to handle multiple plants simultaneously, and pushing this further several controllers may each communicate with multiple plants and also communicate with one another, essentially forming a network of hybrid system. Note that the latter comes close to the broader concept of *cyber-physical system* [LS14].

To sum up, hybrid systems may consist in more than one component of each type, and these components may be organised in multiple ways, thus defining different hybrid system *architectures*. This is an essential part of hybrid system design that needs to be tackled in a formal way.

In order to model different hybrid system architectures, we propose an extension of our framework that allows the formal modelling of a hybrid system into various architectural structures, using different building blocks. The patterns are presented under the form of a refinement, which rules may be applied to existing hybrid systems (using instantiation).

In this chapter, we propose a set of architecture styles of hybrid systems based on how one or many plants are controlled. We formalise architecture patterns as basic models for different hybrid system components. Section 6.1 gives a general intuition behind the proposed architecture patterns, and in particular their common points and their place in the framework. Section 6.2 defines the single-to-single architecture pattern as a direct use of the generic model. Section 6.3 describes the single-to-many architecture pattern, where one controller controls multiple pants; Section 6.4 extends on this concept and proposes the many-to-many architecture pattern, where many plants are controlled by many controllers. Both architecture patterns are put to use to address two cases studies in Section 6.5. Finally, Section 6.6 concludes the chapter with a short discussion on the contribution.

6.1 General Idea

The hybrid system architecture patterns presented in this chapter are formalised as a refinement of the generic model. The generic model formalises a single controller, single plant hybrid system; it is

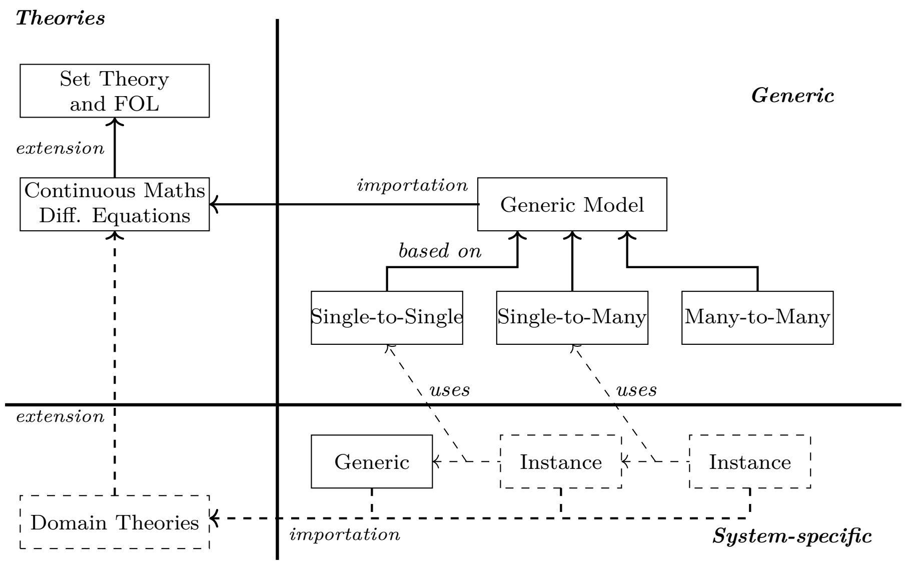


Figure 6.1: Framework Tools – Architecture Patterns

considered as an abstract model where both controller and plant can be refined, in order to model systems with many controllers and/or plants.

In general, refining a single-to-single type of model into any pattern poses two important issues:

1. how is the state of the concrete machine related to the one of the abstract machine?
2. how is the behaviour of the concrete machine allowed by the abstract one?

The first point is handled using a *gluing invariant*: the continuous state of the abstract system (x_p^A) is usually replaced by a set of concrete continuous states ($\{x_{p,1}^C, x_{p,2}^C, \ldots\}$), and the two are linked by a particular function (e.g. linear/polynomial combination, etc.). This together with additional constraints on the evolution domain of each continuous state variable, define the gluing invariant:

$$x_p^A = f(x_{p,1}^C, x_{p,2}^C, \ldots) \wedge P(x_{p,1}^C, x_{p,2}^C, \ldots)$$

This equality allows the *substitution* of x_p^A by $f(x_{p,1}^C, \ldots)$ in the refinement. Additionally, the form of f determines different mathematical reasoning that may be exploited during the proof process (e.g. linear combination, function composition, etc.).

The second point is taken care of of by the refinement rules, and in particular *guard strengthening* and *action simulation*:

- For the former, the core idea is that abstract guards, upon substituting the abstract variables (using the gluing invariant), must remain true;
- For the latter, the goal is to ensure that the behaviour of each of the concrete variables, once "processed" through the gluing invariant, is compatible with the behaviour of the abstract variables. This has to be realised while retaining some *liberty*, as over-constraining the continuous variables will hinder the expressiveness of refinement;

For the former, the core idea here is that abstract guards, upon substituting the abstract variable (using the gluing invariant) must remain true.

For action simulation, the goal is to ensure that the behaviour of each of the concrete variables, once “processed” through the gluing invariant, is compatible with the behaviour of the abstract variables. This must be done while maintaining a certain level of *liberty*, as over-constraining the continuous variables hinders the expressiveness of the refinement.

6.2 Single-to-Single Architecture Pattern

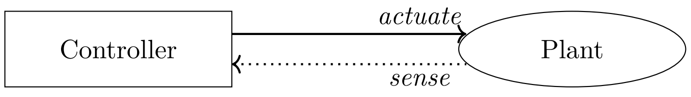


Figure 6.2: Single-to-single Pattern Typical Scenario

The *single-to-single* (S2S for short) architecture pattern is depicted in Figure 6.2. It is the general structure of hybrid systems handled by the generic model, and consists of a single controller, controlling a single plant.

This pattern corresponds to refining the generic model and instantiate it for a given hybrid system. It is presented in full details in Chapter 5. Note that it is used as the first step in the case studies proposed in this chapter (in Section 6.5.3).

6.3 Single-to-Many Architecture Pattern

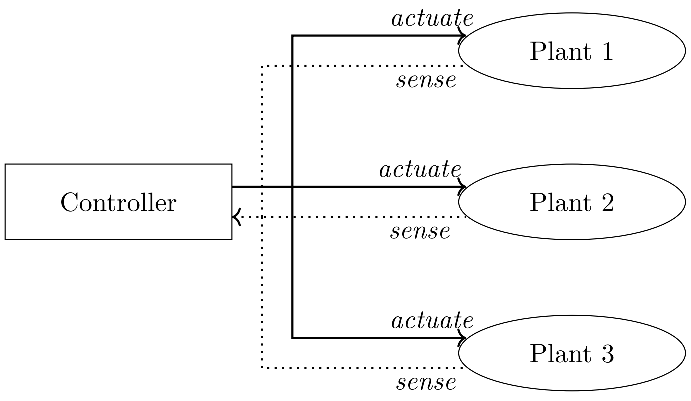


Figure 6.3: Single-to-many Pattern Typical Scenario

The *single-to-many* (S2M for short) pattern appears in scenarios such as the one depicted on Figure 6.3. They typically represent *centralised control* systems: a single controller is controlling multiple plants. It is able to access each plant’s state, and influence each plant using separate actuators. In this setup, the difficulty is to enforce a global property on the system, while having separate plants, each with their own properties.

As a simple example of this kind of situation, one can imagine a set of tanks which valves are being controlled by one single system. The goal is to maintain a certain volume of liquid overall (i.e. globally on every tanks) by opening output and input valves as needed. The difficulty of this system comes from the ability to change the volume of multiple tanks while enforcing this invariant (for example, we could have several completely empty tanks while still retaining the property).

6.3.1 Model

We present this pattern as a refinement, showing an abstract generic model (denoted by a superscript A) and a refinement (denoted by a superscript C) side by side.

As a matter of simplification, we make the assumption that the concrete machine handles 2 continuous state variables (i.e. two plants), but it can easily be extended to use as many continuous state variables as needed. We also consider that the discrete state does not change from one machine to the other.

6.3.1.1 Machine Headers

MACHINE M^A
VARIABLES t , x_s , x_p^A
INVARIANTS
inv1 : $t \in \mathbb{R}^+$
inv2 : $x_s \in \text{STATES}$
inv3 : $x_p^A \in \mathbb{R} \rightarrow\!\!\!\!\!+ S^A$
inv4 : $[0,t] \subseteq \text{dom}(x_p^A)$

MACHINE M^C **REFINES** M^A
VARIABLES t , x_s , $x_{p,1}^C$, $x_{p,2}^C$
INVARIANTS
inv31 : $x_{p,1}^C \in \mathbb{R} \rightarrow\!\!\!\!\!+ S_1^C$
inv32 : $x_{p,2}^C \in \mathbb{R} \rightarrow\!\!\!\!\!+ S_2^C$
inv41 : $[0,t] \subseteq \text{dom}(x_{p,1}^C)$
inv42 : $[0,t] \subseteq \text{dom}(x_{p,2}^C)$
inv5 : $x_p^A = f(x_{p,1}^C, x_{p,2}^C)$

Listings 6.1: S2M Pattern – Machine Header

Listing 6.1 gives the header of the two machines. Following the generic model presented in Chapter 5, the machines first define a variable for time (t) and a variable for the discrete state (x_s) with associated invariants. Machine M^A defines a continuous state x_p^A valued in S^A and machine M^C defines two continuous states $x_{p,1}^C$ and $x_{p,2}^C$ valued in S_1^C and S_2^C respectively.

The key point here is `inv5` of M^C, which is the *gluing invariant* of the refinement. It states that, on interval $[0,t]$, x_p^A, $x_{p,1}^C$ and $x_{p,2}$ are linked by function $f \in (S_1^C \times S_2^C) \to S^A$.

The initialisation event of both machines is given in Listing 6.2. The main difficulty here is that x_p^A "disappears" by refinement, and is replaced by the new continuous state $x_{p,1}^C, x_{p,2}^C$. It is subsequently substituted in the actions by these new variables, and a witness is provided for $x_p^{A\prime}$ that is based on the gluing invariant (and allows establishing it).

6.3.1.2 Discrete Events

It is assumed the system's discrete behaviour does not change from the abstract machine to the concrete one (i.e. the controller remains single). This entails that *transition* events are left unchanged. *Sensing* events, on the other hand, need to be addressed since they access the continuous state.

```
INITIALISATION^A
THEN
    act1 :  t := 0
    act2 :  x_s :∈ STATES
    act3 :  x_p^A :∈ {0} → S^A
END
```

```
INITIALISATION^C REFINES
        INITIALISATION^A
WITH x_p^A′ :  x_p^A′ = f(x_{p,1}^C′, x_{p,2}^C′)
THEN
    act1 :  t := 0
    act2 :  x_s :∈ STATES
    act31 :  x_{p,1}^C :∈ {0} → S_1^C
    act32 :  x_{p,2}^C :∈ {0} → S_2^C
END
```

Listings 6.2: S2M Pattern – Initialisation

```
Sense^A
ANY s , p^A
WHERE
    grd1 :  s ∈ ℙ1(STATES)
    grd2 :  p^A ∈ ℙ(STATES × ℝ × S^A)
    grd3 :  (x_s ↦ t ↦ x_p^A(t)) ∈ p^A
THEN
    act1 :  x_s :∈ s
END
```

```
Sense^C REFINES Sense^A
ANY s , p^A
WHERE
    grd1 :  s ∈ ℙ1(STATES)
    grd2 :  p^A ∈ ℙ(STATES × ℝ × S^A)
    grd3 :  (x_s ↦ t ↦ f(x_{p,1}^C(t), x_{p,2}^C(t))) ∈ p^A
THEN
    act1 :  x_s :∈ s
END
```

Listings 6.3: S2M Pattern – Sensing Event

Listing 6.3 presents the sensing events for the S2M pattern. The key point here is that the event is unchanged, except for the abstract state x_p^A that is substituted with the concrete state $(x_{p,1}^C, x_{p,2}^C)$ by exploiting the particular shape of the gluing invariant.

This replacement is sound (see Section 6.3.2.1) and the resulting guard is *equivalent* to the guard of the abstract sensing event. Note that, as for any refinement, it is possible to propose a stronger guard.

6.3.1.3 Continuous Events

Since the refinement for the S2M pattern affects the system's continuous state, continuous event present numerous changes. In this section, we address the case of the *actuation* event only, since *behave* can be seen as a simpler (i.e. less constrained) version of the actuation event.

Listing 6.4 shows the `Actuate` event for the pattern. Notice the use of the direct product $\otimes$ (Operator 5) to bind together two functions, in order to be able to use the theorie's operators on them.

The only parameter to change is $\mathcal{P}^A$, replaced by $\mathcal{P}^C$. In theory, we would need to provide a witness, but since this is a pattern, it is expected that, at this point, these predicates actually contain definite values, and a witness would heavily depend on these values.

Local invariant H^A is unchanged; instead, guards `grd2` and `grd6` are updated to use the concrete continuous state $(x_{p,1}^C, x_{p,2}^C)$ instead of the abstract one, again exploiting the gluing invariant. The evolution domain of the CBAP operator (right-hand side of the &) is modified in a similar way. It is used for establishing simulation (see Section 6.3.2.2). Note that, equivalently, predicate of the

```
**Actuate**$^A$
ANY $\mathcal{P}^A$ , $s$ , $H^A$ , $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}^A \in (\mathbb{R}^+ \nrightarrow S^A) \times (\mathbb{R}^+ \nrightarrow S^A)$
    grd2 : $\textbf{Feasible}(x_p^A, [t, t'], \mathcal{P}^A, H^A)$
    grd3 : $s \subseteq \text{STATES}$
    grd4 : $x_s \in s$
    grd5 : $H^A \subseteq S^A$
    grd6 : $x_p^A(t) \in H^A$
THEN
    act1 : $x_p^A :|_{t \to t'} \mathcal{P}^A(x_p^A, x_p^{A\prime}) \ \& \ H^A$
END
```

```
**Actuate**$^C$ REFINES **Actuate**$^A$
ANY $\mathcal{P}^C$ , $s$ , $H^A$ , $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}^C \in (\mathbb{R}^+ \nrightarrow (S_1^C \times S_2^C)) \times (\mathbb{R}^+ \nrightarrow (S_1^C \times S_2^C))$
    grd2 : $\textbf{Feasible}(x_{p,1}^C \otimes x_{p,2}^C, [t, t'], \mathcal{P}^C, f^{-1}[H^A])$
    grd3 : $s \subseteq \text{STATES}$
    grd4 : $x_s \in s$
    grd5 : $H^A \subseteq S^A$
    grd6 : $f(x_{p,1}^C(t), x_{p,2}^C(t)) \in H^A$
WITH
    $x_p^{A\prime}$ : $x_p^{A\prime} = f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime})$
THEN
    act1 : $x_{p,1}^C, x_{p,2}^C :|_{t \to t'} \mathcal{P}^C(x_{p,1}^C \otimes x_{p,2}^C, x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime}) \ \& \ f(x_{p,1}^C, x_{p,2}^C) \in H^A$
END
```

Listings 6.4: S2M Pattern – Actuation Event

form $f(x_{p,1}^C, x_{p,2}^C) \in H^A$ may be replaced by:

$$(x_{p,1}^C, x_{p,2}^C) \in f^{-1}[H^A]$$

Finally, a witness is provided for $x_p^{A\prime}$, recalling the gluing invariant.

6.3.2 Proofs

The single-to-many architecture pattern yields multiples proof obligations that ought to be proved. Since it is a pattern, these proof obligations are general, and discharging them often requires to rely on the particular shape and properties of the model's parameters (predicates, variables, and so on).

In this section, we study the shape of the POs associated with the pattern, and give general techniques to discharge them.

6.3.2.1 Guard Strengthening

Guard strengthening (see Section 2.3.3.1) is a proof obligation that is directly related to refinement, and thus is generated when instantiating the pattern. In our case, we address guard strengthening for guards that involve the continuous state, that is, the guards of sensing events and the evolution domain consistency of actuation events.

For instance, guard strengthening for sensing events yields the following PO:

$$\begin{aligned} A \wedge I \wedge W \wedge\ x_p^A \underset{[0,t]}{=} f(x_{p,1}^C, x_{p,2}^C) \wedge (x_s \mapsto t \mapsto f(x_{p,1}^C(t), x_{p,2}^C(t))) \in p^A \\ \Rightarrow (x_s \mapsto t \mapsto x_p^A(t)) \in p^A \end{aligned}$$

Discharging this PO is straightforward, using the gluing invariant to substitute x_p^A with $f(x_{p,1}^C, x_{p,2}^C)$ on the right-hand side of the implication (by noticing that $t \in [0,t]$).

Similarly, we consider the PO associated to evolution domain consistency (`grd6` of actuation events):

$$\begin{aligned} A \wedge I \wedge W \wedge\ x_p^A \underset{[0,t]}{=} f(x_{p,1}^C, x_{p,2}^C) \wedge f(x_{p,1}^C(t), x_{p,2}^C(t)) \in H^A \\ \Rightarrow x_p^A(t) \in H^A \end{aligned}$$

Using the same technique as for guard strengthening POs in sensing events (i.e. substitution using the gluing invariant) allows discharging the PO.

Note that the concrete and abstract guards are *equivalent*. It is always possible to propose a stronger guard in the refinement, provided it implies the abstract one (e.g. $f(x_{p,1}^C(t), x_{p,2}^C(t)) \in H \subseteq H^A$).

6.3.2.2 Simulation

Simulation (see Section 2.3.3.2) is a crucial proof obligation associated with refinement. It ensures that the behaviour of a concrete event simulates the one of the abstract event.

This PO is written as:

$$\begin{aligned} A \wedge I \wedge G \wedge x_p^{A\prime} \underset{[0,t']}{=} f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime}) \wedge \mathit{CBAP}(t, t', x_{p,1}^C \otimes x_{p,2}^C, x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime}, \mathcal{P}^C, f^{-1}[H^A]) \\ \Rightarrow \mathit{CBAP}(t, t', x_p^A, x_p^{A\prime}, \mathcal{P}^A, H^A) \end{aligned}$$

We can unfold the definition of CBAP (as given in Section 4.1.2.1) on each side of the implication:

$$\begin{array}{lll} A \wedge I \wedge G \wedge x_p^{A\prime} \underset{[0,t']}{=} f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime}) & & \\ \quad \wedge [0,t[\lhd (x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime}) = [0,t[\lhd (x_{p,1}^C \otimes x_{p,2}^C) & \Rightarrow [0,t[\lhd x_p^{A\prime} = [0,t[\lhd x_p^A & (PP) \\ \quad \wedge \mathcal{P}^C([0,t] \lhd (x_{p,1}^C \otimes x_{p,2}^C), [t,t'] \lhd (x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime})) & \wedge \mathcal{P}^A([0,t] \lhd x_p^A, [t,t'] \lhd x_p^{A\prime}) & (PR) \\ \quad \wedge (x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime}) \underset{[t,t']}{\in} f^{-1}[H^A] & \wedge x_p^{A\prime} \underset{[t,t']}{\in} H^A & (LI) \end{array}$$

Using the gluing invariant and the witness ($x_p^{A\prime} =_{[0,t']} f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime})$) to substitute x_p^A and $x_p^{A\prime}$ in the formula, and by exploiting the the properties of the domain restriction operator ($\lhd$, Operator 6), the proof for PP and LI are completed.

At this step, the *continuous predicate simulation* (CPSIM) part of the proof is left to prove:

$$\begin{aligned} &A \wedge I \wedge G \wedge x_p^{A\prime} \underset{[0,t']}{=} f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime}) \wedge CBAP(t, t', x_{p,1}^{C} \otimes x_{p,2}^{C}, x_{p,1}^{C\prime} \otimes x_{p,2}^{C\prime}, \mathcal{P}^C, f^{-1}[H^A]) \\ &\qquad \Rightarrow \mathcal{P}^A([0,t] \lhd x_p^A, [t,t'] \lhd x_p^{A\prime}) \end{aligned} \tag{CPSIM}$$

Such proof is to be conducted on a case-by-case basis, as it highly depends on the form of the predicates $\mathcal{P}^A$ and $\mathcal{P}^C$. Evolution domain $f^{-1}[H^A]$ can be strengthened in order to give additional properties, used as hypotheses for this proof.

In the case where the event uses the $:\sim_{t \to t'}$ operator, the proof consists in establishing that the solutions of the concrete equations, once processed through the gluing relation f, yields a function that is a potential solution of the abstract differential equation.

6.4 Many-to-Many Architecture Pattern

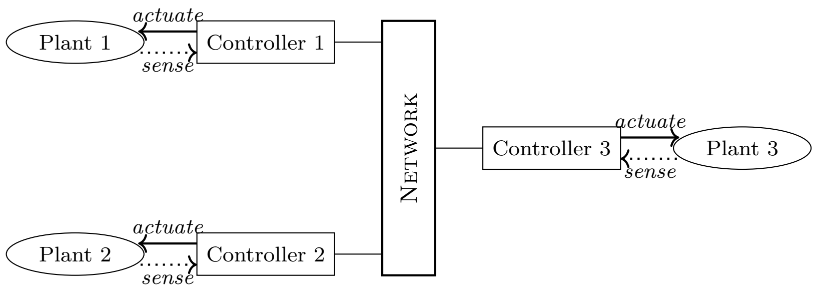


Figure 6.8: Many-to-many Pattern Typical Scenario

The *many-to-many* (M2M for short) architecture pattern corresponds to scenarios such as the one presented in Figure 6.8. These kind of systems consist of several communicating controllers that are linked together via a network (internet, radio or other wireless means, etc.), each of which controls a plant.

Compared to *single-to-many* systems where one controller is constantly aware of the state of every plant, in a *many-to-many* architecture, controllers only have direct access to their associated plant. Information on the other plants state is supplied by other controllers, through the communication network. Such exchange may generate imprecision (e.g. because of communication delay).

This imprecision must be taken into account when designing the system: controllers may have to take decisions depending on a global state, but this imprecision makes it difficult to establish such a global state accurately. This means that controllers must consider and anticipate potential incorrectness in the calculated state, and further constraint their behaviour.

6.4.1 Model

Like for the *single-to-many* pattern, we give the M2M pattern as a refinement. Similarly, we assume that the concrete machine only handles two continuous variables. Note that it can be easily be extended to handle larger systems.

6.4.1.1 Machine States and Working Hypotheses

Let M^A be a machine with discrete state $x_s^A \in \text{STATES}^A$ and continuous state $x_p^A \in \mathbb{R} \nrightarrow S^A$. Let M^C be a refinement of M^A consisting of two sub-components, i.e. two controllers, each being linked to one plant. The sub-components are modeled using two discrete states $x_{s,1}^C \in \text{STATES}_1^C$ and $x_{s,2}^C \in \text{STATES}_2^C$ and two continuous states $x_{p,1}^C \in \mathbb{R} \nrightarrow S_1^C$ and $x_{p,2}^C \in \mathbb{R} \nrightarrow S_2^C$.

Gluing Invariant Similarly to the S2M pattern, the gluing invariant between x_p^A and $(x_{p,1}^C, x_{p,2}^C)$ is of the form:

$$x_p^A = f(x_{p,1}^C, x_{p,2}^C)$$

with $f \in S_1^C \times S_2^C \rightarrow S^A$.

Global Continuous State Estimation In single-to-single systems, the controller has a direct access to the plant's state (sensing is instantaneous). In the case of many-to-many systems, each controller is directly linked to one plant, but cannot sense the other plant's state directly and instantaneously. Instead, each controller sends its associated plant's state to the other controllers through a network, and retrieves through this network the state of the other parts of the system.

This exchange unavoidably introduces imprecision, due to communication delay, rounding errors and so on. Consequently, controllers cannot access an exact state of the global system. Therefore, in order to maintain a global invariant on the system, each controller has to record a safe estimation of the state of the other components.

In practice, this is modelled using specific variables, $x_{p,i}^{sim}$, representing the estimation of the state of the remainder of the system.

This state is assumed to be 1) regularly updated, such that the delay between its current value and the actual value of this state is *bounded* (i.e. bounded delay in the communication) 2) the controller is able to *emulate* the behaviour of this variable between two updates, meaning it is able to estimate the behaviour of the remainder of the system, using physics simulation, for example.

Concretely, Controller 1 is keeping track of variable $x_{p,2}^{sim} \in \mathbb{R} \nrightarrow S_2^C$ that simulates Plant 2, and Controller 2 is keeping track of variable $x_{p,1}^{sim} \in \mathbb{R} \nrightarrow S_1^C$ that emulates Plant 1.

Since communication delay is bounded, we assume that the difference between the estimation of the remainder of the state $x_{p,i}^{sim}$ and its actual value $x_{p,i}^C$ is bounded. We note Δ_i^{sim} this bound, and we have:

$$\forall i \in \{1,2\}, \forall t^* \in [0,t], \|x_{p,i}^C(t^*) - x_{p,i}^{sim}(t^*)\| \leq \Delta_i^{sim}$$

where $\|\cdot\|$ denote a norm on the state space S_i^C.

This bound depends on the specific properties of the network and of the physical phenomena models of the plants. It is usually provided in a specific theory of domains, formalising the application's domain knowledge.

Discrete State Each controller of the concrete model has its own discrete state: it decides how its associated plant is controlled. However, the abstract model only handles one single state, the global state of the system. For the refinement to be consistent, it is required to provide a gluing invariant, linking the abstract discrete global state to the concrete "local" discrete states of each controller.

The relationship linking the local discrete states of each controller and the global discrete state of the abstract controller is specific to each particular system; it defines how the local controllers behave, depending on the state of each other local controller.

In the following, this link is called a *policy*. It is modelled using a particular set predicate of possible relations between abstract and concrete discrete states. It is available in a given context or in a domain theory. The policy predicate of the form

$$(x_s^A, x_{s,1}^C, x_{s,2}^C) \in \mathit{Policy}$$

establishes this relation.

It is worth noticing that the concept of *policy* arises with the distributed, multi-controller nature of the system, encoded by the two distinct concrete discrete variables $x_{s,1}^C$ and $x_{s,2}^C$. It is a way for the controllers to "agree" on a set of valid behaviours they shall follow. The definition of this predicate does not prevent from the checking of the refinement consistency as shown in Section 6.4.2.

Note that policy is not explicitly defined in the *single-to-many* architecture pattern, as it involves a single controller (equivalently, the policy predicate for a centralised controller is the identity: $x_s^A = x_s^C = x_s$).

6.4.1.2 Machine Header

```
MACHINE $M^A$
VARIABLES $t$ , $x_s^A$ , $x_p^A$
INVARIANTS
    inv1 :  $t \in \mathbb{R}^+$
    inv2 :  $x_s^A \in \mathrm{STATES}^A$
    inv3 :  $x_p^A \in \mathbb{R} \nrightarrow S^A$
    inv4 :  $[0,t] \subseteq \mathrm{dom}(x_p^A)$
```

```
MACHINE $M^C$ REFINES $M^A$
VARIABLES $t$ , $x_{s,1}^C$ , $x_{s,2}^C$ , $x_{p,1}^C$ , $x_{p,2}^C$ , $x_{p,1}^{sim}$ , $x_{p,2}^{sim}$
INVARIANTS
    inv21 :  $x_{s,1}^C \in \mathrm{STATES}^C$
    inv22 :  $x_{s,2}^C \in \mathrm{STATES}^C$
    inv31 :  $x_{p,1}^C \in \mathbb{R} \nrightarrow S_1^C$
    inv32 :  $x_{p,2}^C \in \mathbb{R} \nrightarrow S_2^C$
    inv41 :  $[0,t] \subseteq \mathrm{dom}(x_{p,1}^C)$
    inv42 :  $[0,t] \subseteq \mathrm{dom}(x_{p,2}^C)$
    inv51 :  $x_{p,1}^{sim} \in \mathbb{R} \nrightarrow S_1^C$
    inv52 :  $x_{p,2}^{sim} \in \mathbb{R} \nrightarrow S_2^C$
    inv61 :  $[0,t] \subseteq \mathrm{dom}(x_{p,1}^{sim})$
    inv62 :  $[0,t] \subseteq \mathrm{dom}(x_{p,2}^{sim})$
    inv7 :  $\forall t^* \cdot t^* \in [0,t] \Rightarrow \|x_{p,1}^C(t^*) - x_{p,1}^{sim}(t^*)\| \leq \Delta_1^{sim}$
            $\wedge \|x_{p,2}^C(t^*) - x_{p,2}^{sim}(t^*)\| \leq \Delta_2^{sim}$
    inv8 :  $(x_s^A, x_{s,1}^C, x_{s,2}^C) \in \mathit{Policy}$
    inv9 :  $x_p^A = f(x_{p,1}^C, x_{p,2}^C)$
```

Listings 6.5: M2M Pattern – Machine Header

Listing 6.5 gives the header of both abstract and refined machines. The main point here is the correct definition of every variable used by the models, namely discrete ($x_s^{A/C}$) and continuous ($x_p^{A/C}$) states, as well as the variables used for emulating the global state (x_p^{sim}).

Following the discussion of Section 6.4.1.1, the behaviour of the variables is constrained using invariants. `inv7` ensures the emulated states do not drift too far from the actual states, and `inv8` links the abstract and the concrete discrete states. Finally, `inv9` is the gluing invariant linking the abstract and concrete continuous states.

```
**INITIALISATION$^A$**
THEN
    act1 : $t := 0$
    act2 : $x_s^A :\in \mathrm{STATES}^A$
    act3 : $x_p^A :\in \{0\} \to S^A$
END
```

```
**INITIALISATION$^C$** REFINES **INITIALISATION$^A$**
WITH
    $x_s^{A\prime}$ : $x_s^{A\prime}, x_{p,1}^{C\prime}, x_{p,2}^{C\prime} \in Policy$
    $x_p^{A\prime}$ : $x_p^{A\prime} = f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime})$
THEN
    act1 : $t := 0$
    act2 : $x_{s,1}^C, x_{s,2}^C :\in \mathrm{STATES}^C$
    act31 : $x_{p,1}^C, x_{p,1}^{sim} :\mid x_{p,1}^C\{0\} \to S_1^C \wedge x_{p,1}^{sim} = x_{p,1}^{C\prime}$
    act32 : $x_{p,2}^C, x_{p,2}^{sim} :\mid x_{p,2}^C\{0\} \to S_2^C \wedge x_{p,2}^{sim} = x_{p,2}^{C\prime}$
END
```

Listings 6.6: M2M Pattern – Initialisation

Initialisation for the machines is given in Listing 6.6. Variables are initialised as usual; note that we have chosen the simulating variables ($x_{p,i}^{sim}$) be equal to the exact ones.

Because x_p^A and x_s^A disappear at refinement, witnesses have to be provided. They are based on the refinement's gluing invariants (`inv9` and `inv8`, respectively).

6.4.1.3 Discrete Events

```
**Sense$^A$**
ANY $s^A$ , $p^A$
WHERE
    grd1 : $s^A \in \mathbb{P}1(\mathrm{STATES}^A)$
    grd2 : $p^A \in \mathbb{P}(\mathrm{STATES}^A \times \mathbb{R} \times S^A)$
    grd3 : $(x_s^A \mapsto t \mapsto x_p^A(t)) \in p^A$
THEN
    act1 : $x_s^A :\in s^A$
END
```

```
**Sense$^C{}_1$** REFINES **Sense$^A$**
ANY $s_1^C$ , $p_1^C$
WITH
    $s^A$ , $x_s^{A\prime}$ : $x_s^{A\prime} \in s^A \wedge x_s^{A\prime}, x_{s,1}^{C\prime}, x_{s,2}^C \in Policy$
WHERE
    grd1 : $s_1^C \in \mathbb{P}1(\mathrm{STATES}^C)$
    grd2 : $p_1^C \in \mathbb{P}(\mathrm{STATES}^C \times \mathbb{R} \times (S_1^C \times S_2^C))$
    grd3 :
        $(x_{s,1}^C \mapsto t \mapsto (x_{p,1}^C(t) \mapsto x_{p,2}^{sim}(t))) \in p_1^C$
THEN
    act1 : $x_{s,1}^C :\in s_1^C$
END
```

```
**Sense$^C{}_2$** REFINES **Sense$^A$**
ANY $s_2^C$ , $p_2^C$
WITH
    $s^A$ , $x_s^{A\prime}$ : $x_s^{A\prime} \in s^A \wedge x_s^{A\prime}, x_{s,1}^C, x_{s,2}^{C\prime} \in Policy$
WHERE
    grd1 : $s_2^C \in \mathbb{P}1(\mathrm{STATES}^C)$
    grd2 : $p_2^C \in \mathbb{P}(\mathrm{STATES}^C \times \mathbb{R} \times (S_1^C \times S_2^C))$
    grd3 :
        $(x_{s,2}^C \mapsto t \mapsto (x_{p,1}^{sim}(t) \mapsto x_{p,2}^C(t))) \in p_2^C$
THEN
    act1 : $x_{s,2}^C :\in s_2^C$
END
```

Listings 6.7: M2M Pattern – Sensing Events

Listing 6.7 shows the sensing event for the abstract and concrete machines. The key point is that the abstract sensing event is "split" in two events, one for each controller ($Sense^C{}_1$ and $Sense^C{}_2$).

The main difficulty is to establish a correct relation between abstract and concrete discrete states. This is done using an adequate policy predicate that is used to express the witness for $x_s^{A\prime}$ and s^A.

Correct values must be provded for $p_{1/2}^C$ to substitute p^A. In particular, these concrete events are required to *strengthen* `grd3`.

Note that the transition event follows the same pattern, without the sensing predicate p.

6.4.1.4 Continuous Events

Listing 6.8 shows the actuation event for both systems. This actuation updates both the continuous state ($x_{p,1}^C$ and $x_{p,2}^C$) and the emulated state variable of each controller ($x_{p,1}^{sim}$ and $x_{p,2}^{sim}$).

Continuous state is updated with two predicates, $\mathcal{P}_1^C$ and $\mathcal{P}_2^C$, each of which relates to a continuous variable and the simulated continuous variable of the other (i.e. $(x_{p,1}^C, x_{p,2}^{sim})$ and $(x_{p,1}^{sim}, x_{p,2}^C)$), thus effectively representing two separate and independent actuations, although they are executed simultaneously.

On the other hand, the functions associated to the simulated continuous variables are updated directly with functions ($x_{p,1}^{sim*}$ and $x_{p,2}^{sim*}$), which models the simulations managed and calculated by the controllers themselves.

The predicate employed in the CBAP operator as well as in the **Feasible** predicate is the Cartesian product $\mathcal{P}_1^C \times \mathcal{P}_2^C$, intersected with an additional predicate enforcing the assignments of $x_{p,1}^{sim}$ and $x_{p,2}^{sim}$.

Similarly, H^A is substituted with H^C, with the additional constraints that each continuous state variable simulation must remain close by Δ_i^{sim} from the associated state variable's actual behaviour (i.e. $\|x_{p,i}^C - x_{p,i}^{sim}\| \leq \Delta_i^{sim}$). This additional constraints ensures the provided simulation functions are sufficient for the controller to behave correctly.

Witnesses are provided for s^A and $x_p^{A\prime}$, following the machine's gluing invariants. Note that we do not give witnesses for H^A and $\mathcal{P}^A$; this fact is discussed further in Section 6.4.2.

6.4.2 Proofs

The single-to-many architecture pattern yields multiple POs, mostly related to refinement. It is to be noted, again, that since it is a pattern, there is no general method for discharging all of these POs, since they rely on the specific features of the system.

In this section, we focus on guard strengthening and simulation POs, and we study how they are handled.

6.4.2.1 Guard Strengthening

Guard strengthening appears in particular in sensing events. It ensures the concrete sensing can only occur when there is an abstract sensing is.

In concrete sensing event 1 for instance, the guard strengthening PO has the form:

$$\begin{gathered} A \wedge I \wedge x_s^A, x_{s,1}^C, x_{s,2}^C \in \mathit{Policy} \wedge x_p^A = f(x_{p,1}^C, x_{p,2}^C) \wedge \|x_{p,2}^C - x_{p,2}^{sim}\| \leq \Delta_2^{sim} \\ \wedge (x_{s,1}^C, t, (x_{p,1}^C(t), x_{p,2}^{sim}(t))) \in p_1^C \Rightarrow (x_s^A, t, x_p^A(t)) \in p^A \end{gathered}$$

The difficulty is to devise an updated guard (i.e. a value for p_1^C) that allows establishing guard strengthening. In the particular case where the guard is of the form $x_p^A \in \hat{p}^A$ (and $(x_{p,1}^C, x_{p,2}^{sim}) \in \hat{p}_1^C$),

```
**Actuate**$^A$
ANY $\mathcal{P}^A$ , $s$ , $H^A$ , $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}^A \in (\mathbb{R}^+ \mapsto S^A) \times (\mathbb{R}^+ \mapsto S^A)$
    grd2 : $\textbf{Feasible}(x_p^A, [t, t'], \mathcal{P}^A, H^A)$
    grd3 : $s \subseteq \mathrm{STATES}$
    grd4 : $x_s \in s$
    grd5 : $H^A \subseteq S^A$
    grd6 : $x_p^A(t) \in H^A$
THEN
    act1 : $x_p^A :|_{t \to t'} \mathcal{P}^A(x_p^A, x_p^{A\prime})\ \&\ H^A$
END
```

```
**Actuate**$^C$ REFINES **Actuate**$^A$
ANY $\mathcal{P}_1^C$ , $\mathcal{P}_2^C$ , $s_1^C$ , $s_2^C$ , $x_{p,1}^{sim*}$ , $x_{p,2}^{sim*}$ , $H^C$ , $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}_1^C \in (\mathbb{R}^+ \mapsto (S_1^C \times S_2^C)) \times (\mathbb{R}^+ \mapsto (S_1^C \times S_2^C)) \wedge$
           $\mathcal{P}_2^C \in (\mathbb{R}^+ \mapsto (S_1^C \times S_2^C)) \times (\mathbb{R}^+ \mapsto (S_1^C \times S_2^C))$
    grd2 : $\textbf{Feasible}(x_{p,1}^C \otimes x_{p,2}^{sim*} \otimes x_{p,1}^{sim*} \otimes x_{p,2}^C, [t, t'],$
           $\mathcal{P}_1^C \times \mathcal{P}_2^C \cap \{\hat{x}_{p,1}, \hat{x}_{p,2}^{sim}, \hat{x}_{p,1}^{sim}, \hat{x}_{p,2} \mid x_{p,1}^{sim} =_{[t,t']} x_{p,1}^{sim*}, x_{p,2}^{sim} =_{[t,t']} x_{p,2}^{sim*}\},$
           $H^C \cap \{\hat{x}_{p,1}, \hat{x}_{p,2}^{sim}, \hat{x}_{p,1}^{sim}, \hat{x}_{p,2} \mid \|\hat{x}_{p,1} - \hat{x}_{p,1}^{sim}\| \leq \Delta_1^{sim} \wedge \|\hat{x}_{p,2} - \hat{x}_{p,2}^{sim}\| \leq \Delta_2^{sim}\})$
    grd3 : $s_1^C \subseteq \mathrm{STATES}^C \wedge s_2^C \subseteq \mathrm{STATES}^C$
    grd4 : $x_{s,1}^C \in s_1^C \wedge x_{s,2}^C \in s_2^C$
    grd5 : $x_1^{sim*} \in \mathbb{R} \mapsto S_1^C \wedge [t, t'] \subseteq \mathrm{dom}(x_1^{sim*})$
    grd6 : $x_2^{sim*} \in \mathbb{R} \mapsto S_2^C \wedge [t, t'] \subseteq \mathrm{dom}(x_2^{sim*})$
    grd7 : $H^C \subseteq S_1^C \times S_2^C \times S_1^C \times S_2^C$
    grd8 : $x_{p,1}^C(t), x_{p,2}^{sim*}(t), x_{p,1}^{sim*}(t), x_{p,2}^C(t) \in H^C$
WITH
    $x_p^{A\prime}$ : $x_p^{A\prime} = f(x_{p,1}^{C\prime}, x_{p,2}^{C\prime})$
    $s^A$ : $x_s^A, x_{s,1}^C, x_{s,2}^C \in \mathit{Policy} \wedge x_s^A \in s^A$
THEN
    act1 : $x_{p,1}^C, x_{p,1}^{sim}, x_{p,2}^C, x_{p,1}^{sim} :|_{t \to t'}$
               $\mathcal{P}_1^C(x_{p,1}^C \otimes x_{p,2}^{sim}, x_{p,1}^{C\prime} \otimes x_{p,2}^{sim\prime}) \wedge \mathcal{P}_2^C(x_{p,1}^{sim} \otimes x_{p,2}^C, x_{p,1}^{sim\prime} \otimes x_{p,2}^{C\prime})$
               $\wedge x_{p,1}^{sim} =_{[t,t']} x_{p,1}^{sim*} \wedge x_{p,2}^{sim} =_{[t,t']} x_{p,2}^{sim*}$
               $\&\{\hat{x}_{p,1}^C, \hat{x}_{p,2}^{sim}, \hat{x}_{p,1}^{sim}, \hat{x}_{p,2}^{sim*} \mid$
                   $\hat{x}_{p,1}^C, \hat{x}_{p,2}^{sim}, \hat{x}_{p,1}^{sim}, \hat{x}_{p,2}^C \in H^C$
                   $\wedge \|\hat{x}_{p,1}^C - \hat{x}_{p,1}^{sim}\| \leq \Delta_1^{sim} \wedge \|\hat{x}_{p,2}^C - \hat{x}_{p,2}^{sim}\| \leq \Delta_2^{sim}\}$
END
```

Listings 6.8: M2M Pattern – Actuation Event

we exploit the shape of the gluing invariant and the properties of $x_{p,2}^{sim}$. We then have to find $\hat{p}_1^C$ such that:

$$
\begin{aligned}
(x_{p,1}^C, x_{p,2}^{sim}) \in \hat{p}_1^C \Rightarrow \forall x_{p,2}^* \cdot x_{p,2}^* \in S_C^2 \wedge \|x_{p,2}^* - x_{p,2}^{sim}\| \leq \Delta_2^{sim} \\
\Rightarrow f(x_{p,1}^C, x_{p,2}^*) \in \hat{p}^A
\end{aligned}
$$

Indeed, if we have this property, $x^*_{p,2}$ is substituted with $x^C_{p,2}$, and the gluing invariant is exploited to substitute $f(x^C_{p,1}, x^C_{p,2})$ with x^A_p. This means that, to avoid missing the sensing of an event, the guard needs to take into account the simulation error bound Δ^{sim}_2.

Note that the constraints induced by the communication network and the continuous behaviours of the plants may entail a value of Δ^{sim}_i for which the only correct value for $\hat{p}_C$ is $\emptyset$, i.e. the only correct guard for the concrete event with regard to guard strengthening is $\perp$. In this case, the system loses features (some concrete events are disabled).

For instance, if Δ^{sim}_i is to big, it may be impossible to find a predicate $\hat{p}^C$ strong enough to imply $\hat{p}^A$.

6.4.2.2 Simulation

As for the single-to-many pattern, this pattern generates simulation POs, in particular associated with continuous events. A simulation proof obligation for the system's actuation (with unfolded CBAP) is written as:

$$\begin{gathered}
A \wedge I \wedge G \wedge x^{A\prime}_p \underset{[0,t']}{=} f(x^{C\prime}_{p,1}, x^{C\prime}_{p,2}) \wedge \mathcal{P}^C_1(x^C_{p,1} \otimes x^{sim}_{p,2}, x^{C\prime}_{p,1} \otimes x^{sim\prime}_{p,2}) \wedge \mathcal{P}^C_2(x^{sim}_{p,1} \otimes x^C_{p,2}, x^{sim\prime}_{p,1} \otimes x^{C\prime}_{p,2}) \wedge \\
\forall t^* \cdot t^* \in [0,t'] \Rightarrow \|x^{C\prime}_{p,1}(t^*) - x^{sim\prime}_{p,1}(t^*)\| \leq \Delta^{sim}_1 \wedge \|x^{C\prime}_{p,2}(t^*) - x^{sim\prime}_{p,2}(t^*)\| \leq \Delta^{sim}_2 \\
\wedge x^{C\prime}_{p,1}(t^*), x^{sim\prime}_{p,2}(t^*), x^{sim\prime}_{p,1}(t^*), x^{C\prime}_{p,2}(t^*) \in H^C \\
\Rightarrow \mathcal{P}^A(x^A_p, x^{A\prime}_p) \wedge \forall t^* \cdot t^* \in [t,t'] \Rightarrow x^{A\prime}_p(t^*) \in H^A
\end{gathered}$$

This PO drives the design of $\mathcal{P}^C_i$ and H^C, in a way similar to the single-to-many pattern, with the additional inclusion of the simulation error bound. Just like for guard strengthening, the idea is that these predicates shall be strict enough that, even considering the possible gap between simulated and direct variables, simulation still holds.

Note that, for H^C, the same technique as for guard strengthening is used.

6.5 Application to Cases Studies

To demonstrate the use of these two patterns, we propose a general case study, classically exemplified in control theory literature. The objective is to handle one or more *water tanks*, using various forms of control, and ensuring a global invariant (e.g. that the total volume is comprised between two set bounds).

This case study is versatile, and it can be adapted to illustrate the architecture patterns presented in this chapter.

6.5.1 Description

Figure 6.13 depicts an abstract representation of the problem. The system consists of some tank or tanks containing a volume of liquid, of arbitrary shape and architecture. This (these) tanks is (are) linked to a generic controller that is able to actuate pumps (an input and an output pump) to fill or empty the tank(s). This controller can also sense the volume in the tank(s).

The goal of the system is to maintain the volume of liquid inside the tank(s) between too given constants (**SAF1**), and also to ensure that the flow in the tank is never too strong, in order to avoid turmoil (**SAF2**).

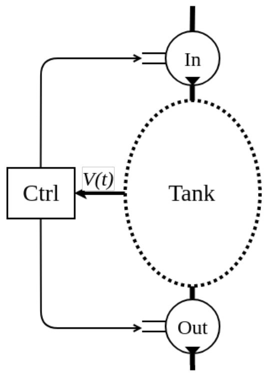


Figure 6.13: Abstract Tank Model

Last, we suppose that activating the input pump will fill up the tank(s), and activating the output pump will empty it. Also, when no pump is active, the volume in the tank does not vary. Apart from that, no additional hypothesis is done on the system at this level.

In the following sections, we will study this system under different hypotheses, and most importantly using different architectures.

6.5.2 A Theory of Tanks

Water tanks are associated with a number of specific phenomena and properties: pumps physical modelling, specific differential equations, etc. These features have been formalised in regrouped in a *domain specific theory* of water tanks.

The theory is split in two parts: a theory for *valves* (Listing 6.9) and a theory for *flows* (Listing 6.10). It defines a number of operators used in the model when instantiating the patterns, as well as several theorems and axioms, used during the proving process.

Valves. The theory first defines the **Status** datatype as an enumeration for identifying whenever a valve is open (*ValveOpen*) or closed (*ValveClosed*). Secondly, the theory defines the **InOutValve** product type that encompass the status of two valves (an input and an output). This type is associated with a single constructor, *InOut* that allows setting both statuses.

These types are associated with convenient operators for opening and closing the valves (*closeIn*, *openOut*, etc.). The theory also provides a way to associate a real number to a valve status, *ValveClosed* being associated with 0 and *ValveOpen* with 1. Finally, the theory defines a set of possible valve combinations, *InOutPossible*.

Flow. This second theory is based on the previous one and on the differential equation theory. It defines several operators to characterise the various features of any tank. In particular, it proposes an enumeration datatype for the possible tank states **TankState** (see Section 6.5.3.1). It also makes accessible various operators to handle the generic concept of *flow*.

The predicate *isFlow* is defined, that is true for a domain $D \subseteq \mathbb{R}$, a state $s \in$ **TankState**, a function $\Phi \in \mathbb{R} \nrightarrow \mathbb{R}$ with $D \subseteq \mathrm{dom}(\Phi)$ and two bounds Q_{min} and Q_{max} (in $\mathbb{R}$) if and only if Φ *behaves according* to s on D, and is bounded by Q_{min} and Q_{max}.

```
THEORY Valves IMPORT RReal
DATATYPES
  Status ≙ ValveOpen | ValveClosed
  InOutValve ≙ InOut(in_status : Status, out_status : Status)
OPERATORS
  closeIn expression (iov: InOutValve)
    recursive definition
      cases iov
        − InOut(i, o) ⇒ InOut(ValveOpen, o)
  . . .
  in_rstatus expression (iov: InOutValve)
    recursive definition
      cases iov
        − InOut(ValveOpen, o) ⇒ 1.0
        − InOut(ValveClosed, o) ⇒ 0.0
  InOutPossible expression ()
    direct definition {InOut(ValveOpen, ValveOpen), . . .}
  . . .
```

Listings 6.9: Valves Theory

```
THEORY Flow IMPORT Valves, DiffEq
DATATYPES
  TankState ≙ Stable | Emptying | Filling | Normal
OPERATORS
  isFlow <predicate> (s: TankState, D: ℙ(ℝ), Φ: ℝ ⇸ ℝ, Q_min: ℝ,
      Q_max: ℝ)
    well−definedness D ⊆ dom(Φ)
    recursive definition
      cases s
        − Stable ⇒ constant(D, Φ)
        − Emptying ⇒ decreasing(D, Φ) ∧ Q_min ≤ Φ ≤ Q_max
  . . .
  isFlowEq <predicate> (s: TankState, D: ℙ(ℝ), e: DE(S), Q_min: ℝ
      , Q_max: ℝ)
    well−definedness Solvable(D, e)
    direct definition ∀η · η ∈ ℝ ⇸ ℝ ∧ D ⊆ dom(η) ∧ solutionOf(D, η, e)
                          ⇒ isFlow(s, D, η, Q_min, Q_max)
  . . .
  FlowODE <expression> (Q_min: ℝ, Q_max: ℝ, δ^in: ℝ, δ^out: ℝ, io:
      InOutValve)
    well−definedness 0 < Q_min, 0 < Q_max, δ^in > 0, δ^out > 0
    direct−definition . . .
```

Listings 6.10: Flow Theory

In this context, *behaving according to s* typically means that Φ is decreasing on D if state s is *Emptying*, constant if state is *Stable* and so on.

This predicate is adapted to be used on differential equations (*isFlowEq*), to check that the solutions of given equation are adequate with regard to the given state. This operator is further specialised to ODE, since the ODE's function's sign gives the monotony of the solution. Additionally, the theory gives a simple ODE for modelling the flow of a tank (*FlowODE*, corresponding to the one presented in Section 6.5.4). It also proposes an ODE for a tank with no flow, used for initialisation.

Finally, we define various theorems aimed at handling the defined operators, especially the ODEs and their solvability.

6.5.3 Generic Model Instantiation – One Controller And One Tank

The first step for addressing this case study is to model the abstraction of the system depicted on Figure 6.13. This initial model indeed serves as a base for the application of the single-to-many and many-to-many patterns.

Formally, the abstract tank model is a refinement of the generic model. It consists of one controller linked to one plant, in a single-to-single architecture. Note that no hypothesis is done on the tank; this allows the refinement of this tank using multiple tank (and thus to implement the different studied architectures).

6.5.3.1 Preliminary Study

Plant Description. The controlled plant is an abstract tank containing a volume $V(t) \geq 0$ of liquid. This tank is finite and can contain a maximum volume of $V_{max} > 0$. The plant is tied to two pumps, one input pumps able to fill-up the tank (i.e. increase $V(t)$) and another to empty it (i.e. decrease $V(t)$). Note that the behaviour of $V(t)$ is unspecified when both pumps are open at the same time.

At this point, no hypothesis is made on the shape of the tank, and no hypothesis is made on the exact behaviour of the pumps, apart that 1) when the input pump is open the volume increases, 2) when the output pump is open the volume decreases, and finally 3) when no pump is open the volume does not change.

Consequently, for this model, the differential equation describing the volume's behaviour is *unknown*, although it is constrained by the hypotheses on the pumps' behaviour.

Note that, for convenience, it is assumed that tank initially contains some arbitrary initial volume V_0, such that $V_{low} \leq V_0 \leq V_{high}$.

Controller Description. The controller's task is to actuate the input and output pumps, in order to fill or empty the tank. Concretely, it operates in 4 modes, and switches from one to another when needed, to preserve safety. These modes are:

- *stable*, the volume does not change (i.e. V is a constant function). Provided V remains within the bounds, the controller can always switch to this mode;
- *emptying*, the volume is decreasing. The controller may switch to this mode if $V > V_{low}$, and it switches automatically to it when $V = V_{high}$;
- *filling*, the volume is increasing. The controller may switch to this mode if $V < V_{high}$, and it switches automatically to it when $V = V_{low}$;
- *normal*, the volume varies arbitrarily. The controller may switch to this mode if V is within the bounds. It is a default mode, where the liquid is flooding freely;

These modes induce 4 transition events, allowing each state to be visited, and 2 sensing events that detect when the volume reaches a bound and puts the system in the corresponding correcting mode. Note that, when actuating the pump, the controller must make sure that the global variation of volume in the tank ($\dot{V}$) remains below ΔV_{max}.

Requirements. To summarise, the system's requirement are described as:

FUN1 the volume varies in the tank, it is decreasing in *emptying* mode, increasing in *filling* mode, constant in *stable* mode, and safely varying in *normal* mode;

ENV1 the volume is physically bounded by 0 and some constant V_{max}:

$$\forall t \in \mathbb{R}, 0 \leq V(t) \land V(t) \leq V_{max}$$

SAF1 the volume must always remain within the set bounds, V_{low} and V_{high}:

$$\forall t \in \mathbb{R}, V_{low} \leq V(t) \land V(t) \leq V_{high}$$

SAF2 the variation of the volume ($\dot{V}(t)$) is always below the maximum allowed variation ΔV_{max}:

$$\forall t \in \mathbb{R}, |\dot{V}(t)| \leq \Delta V_{max}$$

6.5.3.2 Event-B Development

The preliminary study for this systems leads to the definition of an Event-B model. This first model (the *abstract tank*) is, as expected, based on the generic model presented in Chapter 5.

```
CONTEXT AbstractTankCtx EXTENDS GenericCtx
CONSTANTS V_max , V_low , V_high , ΔV_max , V_0
AXIOMS
    axm1 :  $S = \mathbb{R}$
    axm2 :  partition($STATES, \{Stable\}, \{Emptying\},$
                $\{Filling\}, \{Normal\}$)
    axm3–6 :  $V_{max}, V_{low}, V_{high}, \Delta V_{max} \in \mathbb{R}$
    axm7 :  $0 \leq V_{low} \leq V_{high} \leq V_{max}$
    axm8 :  $\Delta V_{max} > 0$
    axm9–11 :  $V_0 \in \mathbb{R} \land V_{low} \leq V_0 \land V_0 \leq V_{high}$
END
```

Listings 6.11: Abstract Tank – Context

Constants and Axioms. The system requires a number of properties to be written (in addition to the theories of valves and flows presented in Section 6.5.2) that are found in an Event-B context, given in Listing 6.11. This context defines the needed constants with associated properties (`axm3-11`). We also give the system's state-space ($\mathbb{R}$ in `axm1`) as well as the controller's mode (`axm2`), based on those defined in the tank theories (*flow* and *valves*).

```
MACHINE AbstractTank REFINES Generic
SEES AbstractTankCtx
VARIABLES t, x_s, V
INVARIANTS
    inv1: V ∈ ℝ ⇸ S
    inv2: [0,t] ⊆ dom(V)
    inv3: V = x_p
    inv4: ∀t* · t* ∈ [0,t] ⇒ 0 ≤ V(t) ∧ V(t) ≤ V_max
    inv5: ∀t* · t* ∈ [0,t] ⇒ V_low ≤ V(t*) ∧ V(t*) ≤ V_high
    inv6: ∀t* · t* ∈ [0,t] ⇒ |V̇(t*)| ≤ ΔV_max
```

```
INITIALISATION
WITH
    x'_p: V' = x'_p
THEN
    act1: t := 0
    act2: x_s := Stable
    act3: V := {0 ↦ V_0}
END
```

Listings 6.12: Abstract Tank – Machine Header

Machine Header. Listing 6.12 presents the machine's header and initialisation event. The header defines the variables of the system (mainly the continuous state V) with associated properties (`inv1-2`). `inv3` is the gluing invariant of the system (a substitution) while `inv4` encodes the environment property **ENV1** of the requirements section. Finally, `inv5` and `inv6` model the safety requirement **SAF1** and **SAF2** respecitvely.

The system's initialisation is straightforward. The controller starts in *stable* mode, and the volume is set to the constant V_0, initial volume contained in the tank.

```
ctrl_transition_normal REFINES
    Transition
WHERE
    grd1: V(t) < V_high
    grd2: V_low < V(t)
WITH
    s: s = {Normal}
THEN
    act1: x_s := Normal
END
```

```
ctrl_sense_too_high REFINES Sense
WHERE
    grd1: V_high ≤ V(t)
WITH
    s: s = {Emptying}
    p: p = STATES × ℝ × {V* | V_high ≤ V*}
THEN
    act1: x_s := Emptying
END
```

Listings 6.13: Abstract Tank – Transition and Sensing

Discrete Events. Listing 6.13 gives a transition and sensing events from the system. The transition `ctrl_transition_normal` allows the controller to move to *normal* mode. It is guarded by a condition to preserve safety in this mode.

The sensing event `ctrl_sensing_too_high` detects when the volume reaches V_{high} (through guard `grd1`), and causes the controller to move to *emptying* mode.

For both events, witnesses are provided for s and p, reflecting their guards and target state.

Continuous Event. Listing 6.14 presents the actuation of the system. It refines the abstract `Actuate` event and constrain it in order to ensure the system's invariant holds.

x_p substituted with V using the gluing invariant, and a concrete evolution domain H is given that forces volume V to remain in the given bounds V_{low} and V_{high}. At this level, the differential

EVENT **ctrl_actuate_pumps** **REFINES** **Actuate**
ANY eq , s , t'
WHERE
grd0 : $t' \in \mathbb{R} \land t < t'$
grd1 : $eq \in \mathbf{DE}(S)$
grd2 : $\mathbf{Solvable}([t, t'], eq, \{V* \mid V_{low} \leq V^* \land V^* \leq V_{high}\}$
grd3 : $isFlowEq(s, [t, t'], eq, 0, V_{max})$
grd4 : $s \in STATES$
grd5 : $x_s = s$
grd6 : $V_{low} \leq V(t) \land V(t) \leq V_{high}$
WITH
x'_p : $x'_p = V'$
H : $H = \{V* \mid V_{low} \leq V^* \land V^* \leq V_{high}\}$
THEN
act1 : $V :\sim_{t \to t'} eq \,\&\, \{V* \mid V_{low} \leq V^* \land V^* \leq V_{high}\}$
END

Listings 6.14: Abstract Tank – Actuation Event

equation is still unknown, but it is constrained by the required behaviour of the pumps. These constraints are encapsulated in the *isFlowEq* predicate (see Section 6.5.2).

6.5.3.3 Proofs

At this point, proofs are straightforward. The model generated 124 proof obligations, most of them (around 34%) coming from well-definedness. Simulation and guard strengthening POs (37%) are discharged by exploiting the fact that the events are built by substituting variables/parameters.

Last, invariant proofs (29%), putting apart trivial type-related invariants, rely on the specific constraints given to the model (values for H and use of the *isFlowEq* predicate), together with the properties these constraints entail and that are given in the tank theories.

6.5.4 One Controller with Two Tanks

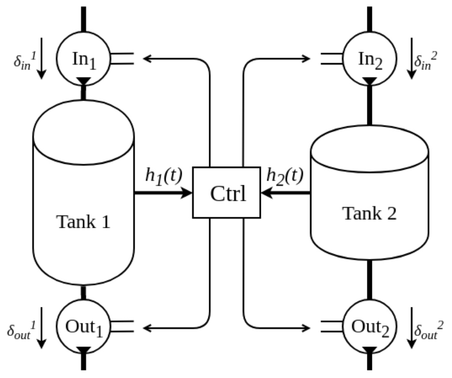


Figure 6.20: One Controllers Two Tanks Configuration

We propose a refinement of the abstract tank model developed in 6.5.3 into a system that consists of one controller controlling two separate tanks of definite shape (see Figure 6.20), using the *single-to-many* architecture pattern presented in Section 6.3. The goal of the system remains the same, but the abstract tank is replaced by two cylindrical tanks of known bases and heights. The sensing is also updated to better reflect "real world" situations: only the height is accessible (e.g. thanks to a float), and the controller needs to derive the actual volume based on this height and the parameters of the tank.

Last, the pumps' behaviour is fixed, they are either fully open or fully closed, and deliver a fixed flow.

6.5.4.1 Preliminary Study

Plants Description. The two plants (T1 and T2) are controlled simultaneously. Each plant consists of a cylinder tank with associated pumps. The volume of each tank cannot be accessed directly; instead the controller senses the *height of liquid* inside, denoted $h_{1/2}(t)$.

T1 has a base of B_1 and a maximum height of $H_{1,max} > 0$. It is hooked up to an input pump of flow $\delta_1^{in} > 0$ and an output pump of flow $\delta_1^{out} > 0$. Initially, it contains a height h_1^0 of liquid. Likewise, tank 2 operates in a similar fashion, replacing the index 1 by 2 in the given parameters.

Note that the volume $V_i(t)$ of tank i is given by the formula $V_i(t) = B_i \times h_i(t)$. The global volume of the system $V(t)$ is then equal to:

$$V(t) = B_1 \times h_1(t) + B_2 \times h_2(t) \tag{6.1}$$

This expression serves as gluing invariant for the system. Note that it is of the form $x_p^A = f(x_{p,1}^C, x_{p,2}^C)$, with $x_p^A = V$, $x_{p,i}^C = h_i$ and $f(h_1, h_2) = B_1 \times h_1 + B_2 \times h_2$, following the notations of Section 6.3.

The defined pump behaviour is modelled by a differential equation for the h_i, based on their status. Let In_i and Out_i the status of the input and of the output pump respectively (with $In_i = 1$ if the pump is open and 0 otherwise); the variation of liquid height in the tank is expressed as:

$$\Delta_i = In_i \times \delta_i^{in} - Out_i \times \delta_i^{out} \tag{6.2}$$

Concretely, the variation of the liquid height is equal to liquid input minus liquid output from the tank. Using this expression, we derive the tanks' ODE:

$$\Phi_i(t, h) = \Delta_i = In_i \times \delta_i^{in} - Out_i \times \delta_i^{out}$$

Note that, in term of pattern application, we have $S^A = S_1^C = S_2^C = \mathbb{R}$.

Controller Description. Apart from the fact that volume is not directly accessible, the controller for this system remains the same as for the abstract one. It operates in the same modes, which are triggered in the same way, replacing V by $B_1h_1 + B_2h_2$, following the system's gluing invariant.

Additionally, as the behaviour of the pumps is known, we define a set of rules that associate the modes of the controller to the state of the pumps:

- *emptying* mode: input pumps closed ($In_i = 0$), output pumps open ($Out_i = 1$);
- *filling* mode: input pumps open ($In_i = 1$), output pumps closed ($Out_i = 0$);

- *stable* mode: every pump closed ($In_i = Out_i = 0$);
- *normal* mode: pumps are closed or open;

This behaviour is summed up as the *TankModeChange* predicate, defined in the *Valve* theory:

$$\begin{aligned} &\mathit{TankModeChange}(x_s, (In_1, Out_1), (In_2, Out_2)) \Leftrightarrow \\ &\qquad (x_s = \mathit{emptying} \Rightarrow In_1 = 0 \land Out_1 = 1 \land In_2 = 0 \land Out_2 = 1) \\ &\qquad \land (x_s = \mathit{filling} \Rightarrow In_1 = 1 \land Out_1 = 0 \land In_2 = 1 \land Out_2 = 0) \\ &\qquad \land \ldots \end{aligned} \tag{6.3}$$

This particular predicate, together with the concrete ODE for the tank allows enforcing requirement **FUN1** of the abstract tank, encoded in `grd3` off event `ctrl_actuate_pumps` of the `AbstractTank` model (Listing 6.14). Concretely, in *emptying* mode, the differential equation yields decreasing solutions, in *filling* mode, the differential equation yields increasing solutions, etc.

Note that it is perfectly possible to define other, richer behaviours; the difficulty is then to enforce requirement **FUN1**, or in other words to establish guard strengthening of guard `grd3` of the abstract event `ctrl_actuate_pumps`.

Requirements. The requirements of the abstract tank are refined and constrained. In particular, the following items are added:

FUN2 The plant consists of 2 tanks with independent pumps that operate with a constant flow;
FUN3 The controller can only sense the height of liquid in each tank; it computes the volume using the sensed height and the parameters of the tanks;
ENV2 Both tanks have a given base B_1 and B_2, and a given maximum height $H_{1,max}$ and $H_{2,max}$; the maximal global volume is then $\hat{V}_{max} = B_1 H_{1,max} + B_2 H_{2,max}$;

Refinement Feasibility. The added requirement **ENV2** implies that the concrete tanks may not have the same total maximum volume as the abstract tank. This means in particular that there exist some combination of tanks that violate requirements **SAF1** of the abstract tank.

To avoid this situation and enforce a correct refinement, it is required that the total maximum volume of the tanks $\hat{V}_{max}$ is greater than the higher bound for the volume V_{high}:

$$\hat{V}_{max} \geq V_{high}$$

Without this constraint, the concrete system may be too constrained, and reject states of $V(t)$ that are correct in the abstract model. In particular, if $V_{max} < V_{low}$, the resulting concrete system is unable to enforce invariant **SAF1** at all.

Similarly, the gluing invariant allows deducing $\dot{V}(t) = B_1 \dot{h}_1(t) + B_2 \dot{h}_2(t)$, and the ODE for the tanks gives the following expression for $\dot{V}$:

$$\dot{V}(t) = B_1 \Delta_1 + B_2 \Delta_2$$

By rewriting this expression, we observe that:

$$|\dot{V}(t)| \leq B_1 \cdot \max(\delta_1^{in}, \delta_1^{out}) + B_2 \cdot \max(\delta_2^{in}, \delta_2^{out})$$

It follows that **SAF2** is equivalent to the following condition, where every symbol is constant:

$$B_1 \cdot \max(\delta_1^{in}, \delta_1^{out}) + B_2 \cdot \max(\delta_2^{in}, \delta_2^{out}) \leq \Delta V_{max}$$

6.5.4.2 Event-B Development

```
CONTEXT 1Ctrl_2Tanks_Ctx EXTENDS AbstractTankCtx
CONSTANTS $B_1$ , $B_2$ , $H_{1,max}$ , $H_{2,max}$ , $h_1^0$ , $h_2^0$ ,
          $\delta_1^{in}$ , $\delta_1^{out}$ , $\delta_2^{in}$ , $\delta_2^{out}$
AXIOMS
    axm1−4 : $B_1, B_2 \in \mathbb{R}$ , $0 < B_1$ , $0 < B_2$
    axm5−8 : $H_{1,max}, H_{2,max}, h_1^0, h_2^0 \in \mathbb{R}$
    axm9−10 : $0 < H_{1,max}$ , $0 < H_{2,max}$
    axm11−12 : $0 < h_1^0 < H_{1,max}$ , $0 < h_2^0 < H_{2,max}$
    axm13 : $V_{high} \leq B_1 H_{1,max} + B_2 H_{2,max}$
    axm14 : $V_0 = B_1 h_1^0 + B_2 h_2^0$
    axm15−18 : $\delta_1^{in}, \delta_1^{out}, \delta_2^{in}, \delta_2^{out} \in \mathbb{R}$
    axm19−22 : $\delta_1^{in} > 0$ , $\delta_1^{out} > 0$ , $\delta_2^{in} > 0$ , $\delta_2^{out} > 0$
    axm23 : $B_1 \max(\delta_1^{in}, \delta_1^{out}) + B_2 \max(\delta_2^{in}, \delta_2^{out}) \leq \Delta V_{max}$
END
```

Listings 6.15: 1 Controller 2 Tanks – Context

Context and Axioms. Listing 6.15 gives the context for the designed system. It encompasses the various constants presented during the preliminary study and their associated properties. In particular, `axm13` and `axm23` encode the properties discussed in the requirement section of the preliminary study, regarding the feasibility of this refinement.

Moreover, `axm14` links the initial value for V (V_0) to the initial values of h_1 and h_2. This is so that we can establish simulation for the initialisation in the machine.

Note that additional useful parameters are found in the tank theories (see Section 6.5.2).

```
MACHINE 1Ctrl_2Tanks REFINES
      AbstractTank
SEES 1Ctrl_2Tanks_Ctx
VARIABLES $t$ , $x_s$ , $h_1$ , $h_2$
INVARIANTS
    inv1 : $h_1 \in \mathbb{R} \rightarrow\!\!\!\!\!+ \mathbb{R}$
    inv2 : $[0, t] \subseteq \mathrm{dom}(h_1)$
    inv3 : $h_2 \in \mathbb{R} \rightarrow\!\!\!\!\!+ \mathbb{R}$
    inv4 : $[0, t] \subseteq \mathrm{dom}(h_2)$
    inv5 : $V = B_1 h_1 + B_2 h_2$
```

```
INITIALISATION
WITH
    $V'$ : $V' = B_1 h_1' + B_2 h_2'$
THEN
    act1 : $t := 0$
    act2 : $x_s := Stable$
    act3 : $h1 := \{0 \mapsto h_1^0\}$
    act4 : $h2 := \{0 \mapsto h_2^0\}$
END
```

Listings 6.16: 1 Controller 2 Tanks – Machine Header

Machine Header. The machine's header and initialisation are shown on Listing 6.16. The machine is, as expected, a refinement of the abstract tank model. Variable V is substituted with variables h_1 and h_2, and the three are linked via the gluing invariant `inv5`, directly taken from Equation 6.1.

Initialisation is built in the same way as for the abstract system, replacing V by the two states h_1 and h_2.

```
ctrl_transition_normal REFINES
    ctrl_transition_normal
WHERE
   grd1 : $B_1h_1(t) + B_2h_2(t) < V_{high}$
   grd2 : $V_{low} < B_1h_1(t) + B_2h_2(t)$
THEN
   act1 : $x_s := Normal$
END
```

```
ctrl_sense_too_high REFINES
    ctrl_sense_too_high
WHERE
   grd1 : $V_{high} \leq B_1h_1(t) + B_2h_2(t)$
THEN
   act1 : $x_s := Emptying$
END
```

Listings 6.17: Abstract Tank – Transition and Sensing

Discrete Events. Listing 6.17 shows the transition and sensing refined events of the abstract tank model. This refinemen follows the principle given in Section 6.3. It relies on substituting V with $B_1h_1 + B_2h_2$ using the gluing invariant.

```
EVENT ctrl_actuate_pumps REFINES ctrl_actuate_pumps
ANY $s$ , $t'$ , $io_1$ , $io_2$
WHERE
   grd0 : $t' \in \mathbb{R} \land t < t'$
   grd1 : $\mathbf{Feasible}([t,t'], h_1 \otimes h_2,$
  $\{\hat{h}_1 \otimes \hat{h}_2, \hat{h}'_1 \otimes \hat{h}'_2 \mid \mathbf{solutionOf}([t,t'], \hat{h}'_1, \mathbf{ode}(FlowODE(0, H_{1,max}, \delta_1^{in}, \delta_1^{out}, io_1), h_1(t), t))$
    $\land \mathbf{solutionOf}([t,t'], \hat{h}'_2, \mathbf{ode}(FlowODE(0, H_{2,max}, \delta_2^{in}, \delta_2^{out}, io_2), h_2(t), t))\},$
  $\{h_1^*, h_2^* \mid V_{low} < B_1h_1 + B_2h_2 \land B_1h_1 + B_2h_2 < V_{high}\})$
   grd2 : $TankModeChange(s, io_1, io_2)$
   grd4 : $s \in STATES$
   grd5 : $x_s = s$
   grd6 : $V_{low} \leq B_1h_1(t) + B_2h_2(t) \land B_1h_1(t) + B_2h_2(t) \leq V_{high}$
WITH
   $V'$ : $V' = B_1h'_1 + B_2h'_2$
   $eq$ : $\mathbf{Solvable}([t,t'], eq, \{V^* \mid V_{low} < V^* \land V^* < V_{high}\}) \land isFlowEq(s, [t,t'], eq, 0, V_{max})$
        $\land \mathbf{solutionOf}([t,t'], B_1h'_1 + B_2h'_2, eq)$
THEN
   act1 : $h_1, h_2 :|_{t \to t'}$
            $\mathbf{solutionOf}([t,t'], h'_1, \mathbf{ode}(FlowODE(0, H_{1,max}, \delta_1^{in}, \delta_1^{out}, io_1), h_1(t), t)) \land$
            $\mathbf{solutionOf}([t,t'], h'_2, \mathbf{ode}(FlowODE(0, H_{2,max}, \delta_2^{in}, \delta_2^{out}, io_2), h_2(t), t))$
            $\&\{h_1^*, h_2^* \mid V_{low} < B_1h_1 + B_2h_2 \land B_1h_1 + B_2h_2 < V_{high}\}$
END
```

Listings 6.18: 1 Controller 2 Tanks – Actuation Event

Continuous Events. Listing 6.18 shows the system's actuation. It is built following the method given in Section 6.3: abstract plant state V is replaced using the gluing invariant, and continuous predicates are updated in order to handle h_1 and h_2 (using too separate ODE) while retaining the exact same evolution domain.

A witness is provided for eq to establish simulation. Note also that the state of the valves are determined using the *TankModeChange* operator as defined in the preliminary study for this development (Section 6.5.4.1).

6.5.4.3 Proofs

This model is associated with 73 proof obligations. They are mostly related to well-definedness (37%), because of the extensive use of theories in the model (both for continuous features and for the domain-specific tank features). POs relating to invariant (21%) are proven using the various results defined in the tank domain theories, and the remaining POs mainly comes from *refinement* (36%).

Guard strengthening is immediate, following the remarks of Section 6.3.2.1: it consists in substituting the abstract and the concrete continuous states using the gluing invariants. For other guards that do not use continuous state, the concrete model defines guards that are stricter than the guards of the abstract model. Guard strengthening proofs is then proven by substituting parameters and variables using the provided witnesses and gluing invariants, and by using the theorems defined in the theory of tanks.

Following the remarks of Section 6.3.2.2, we notice that the simulation proof obligations (associated with the actuation event) boil down to establishing *continuous predicate simulation* (*CPSIM*). Unfolding the associated equation and filling its parameters, we obtain the following equation:

$$\begin{aligned} A \wedge I \wedge G \wedge V' \underset{[0,t']}{=} B_1 h_1' + B_2 h_2' \wedge \mathbf{solutionOf}([t,t'], h_1', \mathbf{ode}(\Phi_1)) \wedge \\ \mathbf{solutionOf}([t,t'], h_2', \mathbf{ode}(\Phi_2)) \wedge \mathbf{solutionOf}([t,t'], B_1 h_1' + B_2 h_2', eq) \\ \Rightarrow \mathbf{solutionOf}([t,t'], V', eq) \end{aligned} \tag{6.4}$$

The witness provided for eq ($\mathbf{solutionOf}([t,t'], B_1 h_1' + B_2 h_2', eq)$) constraints the equations of the concrete tanks to be "compatible" with the equations of the abstract tank. This allows discharging the PO by substituting V' with $B_1 h_1' + B_2 h_2'$ (using the witness for V').

The expression of this witness is associated to a feasibility proof obligation: we are required to prove that there exists eq such that 6.4 holds.

6.5.5 Two Controllers with Two Tanks

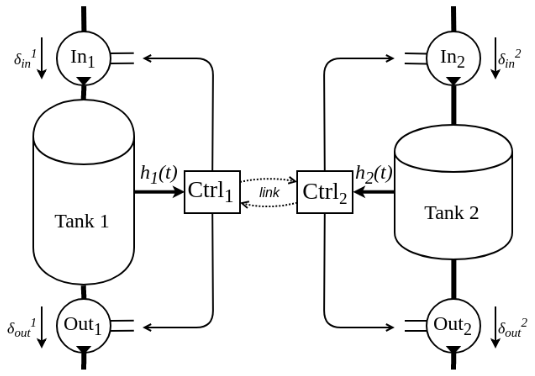


Figure 6.25: 2 Controllers 2 Tanks Configuration

In this section, we propose another refinement of the abstract tank, corresponding to the *many-to-many* architecture pattern presented in Section 6.4. It models a system consisting of two tanks of definite shape, each of which is controlled by an independent controller. Figure 6.25 gives the typical configuration of the system to be designed.

The core of the system is close to the S2M (1 controller 2 tanks) system designed in Section 6.5.4. It consists of two cylinder tanks with known bases and maximum heights, and the controller accesses the height of liquid in the tank rather than the direct volume (which is calculated). The behaviour of the pumps is identical to the S2M case study.

Contrary to the S2M case study where one controller controls several plants (centralised control with the capability to build a global state of the controlled plants), the goal of this refinement is to model a situation where each tank is equiped with a controller, and both controllers *communicate* with each other through a communication network with a bounded delay time.

The difficulty of this development is to the computation of a global state (ideal case) or of a safe estimation of a global state, despite the latency of the network and the error that it generates; this is the main use of the many-to-many architecture pattern with distributed control.

6.5.5.1 Preliminary Study

Plant Description. The plant is exactly the same as for the single-to-many case study (see Section 6.5.4.1: two cylinder tanks of bases B_1 and B_2 and maximum heights $H_{1,max}$ and $H_{2,max}$, containing a height of liquid $h_1(t)$ and $h_2(t)$ and hooked up to pumps of fixed flow ($\delta_{1/2}^{in/out}$).

Consequently, the gluing invariant and the equations used in the model are identical to the S2M case.

Note that in term of pattern application, we again have $S^A = S_1^C = S_2^C = \mathbb{R}$.

Controllers Description. The controllers for this system are more complex than for the S2M case study. The system consists of two controllers, each of which is able to control the pumps and sense the height of liquid in one tank.

The controllers communicate with each other and exchange their current (discrete and continuous) states. The communication is delayed by an arbitrary time that is assumed to be bounded.

In addition to the pumps they control, and in order to maintain a global safety invariant, each controller sees a global state, or a safe approximation of it. For that, it estimates the continuous state of the other controller, basically by *emulating* it using knowledge of its behaviour and the information it gets from the network. Thus, controller 1 keeps track of the emulated height of tank 2 (h_2^{sim}) and similarly controller 2 keeps track of the emulated height of tank 1 (h_1^{sim}).

We assume that, at any given point, the difference between the real height and the simulated height is bounded by a constant that basically encompasses the simulation errors and the network's delay:

$$|h_1 - h_1^{sim}| \leq \Delta_1^{sim} \wedge |h_2 - h_2^{sim}| \leq \Delta_2^{sim}$$

Note that no assumption is made on *how* the controller calculates h_i^{sim}.

Finally, each subsystem has its own discrete state, taken in the **TankState** type (i.e. $STATES_1^C = STATES_2^C = STATES^A =$ **TankState**). These states together with the discrete state of the abstract machine are linked via the *Policy* predicate, left abstract in this development.

Requirements. The goal of the system is, as for the abstract one, to keep the global volume ($B_1h_1 + B_2h_2$) within the bounds V_{low}, V_{high}. The requirements are enriched to reflect the particular architecture of the system:

ENV3 each tank is controlled independently by communicating controllers; communication introduces bounded delays and errors, encapsulated in a constant, Δ_i^{sim}, for each controller i;

6.5.5.2 Event-B Development

CONTEXT 2Ctrl_2Tanks_Ctx **EXTENDS** AbstractTankCtx
CONSTANTS B_1 , B_2 , $H_{1,max}$, $H_{2,max}$, h_1^0 , h_2^0 ,
δ_1^{in} , δ_1^{out} , δ_2^{in} , δ_2^{out} ,
Δ_1^{sim} , Δ_2^{sim} ,
Policy
AXIOMS
axm1–4: $B_1, B_2 \in \mathbb{R}$, $0 < B_1$, $0 < B_2$
axm5–8: $H_{1,max}, H_{2,max}, h_1^0, h_2^0 \in \mathbb{R}$
axm9–10: $0 < H_{1,max}$, $0 < H_{2,max}$
axm11–12: $0 < h_1^0 < H_{1,max}$, $0 < h_2^0 < H_{2,max}$
axm13: $V_{low} \leq B_1 H_{1,max} + B_2 H_{2,max} \leq V_{max}$
axm14: $V_0 = B_1 h_1^0 + B_2 h_2^0$
axm15–18: $\delta_1^{in}, \delta_1^{out}, \delta_2^{in}, \delta_2^{out} \in \mathbb{R}$
axm19–22: $\delta_1^{in} > 0$, $\delta_1^{out} > 0$, $\delta_2^{in} > 0$, $\delta_2^{out} > 0$
axm23: $B_1 \max(\delta_1^{in}, \delta_1^{out}) + B_2 \max(\delta_2^{in}, \delta_2^{out}) \leq \Delta V_{max}$
axm24–25: $\Delta_1^{sim} \in \mathbb{R}$, $\Delta_2^{sim} \in \mathbb{R}$
axm26–27: $\Delta_1^{sim} > 0$, $\Delta_2^{sim} > 0$
axm28: $Policy \subseteq \text{STATES} \times \text{STATES} \times \text{STATES}$
END

Listings 6.19: 2 Controllers 2 Tanks – Context

Constants and Axioms. The context for this machine is given in Listing 6.19. It is very much based on the context of the single-to-many case study (Listing 6.15) with the added constants Δ_i^{sim} and *Policy* required by the pattern.

We recall that other operators are found in the tank theories (see Section 6.5.2).

Machine Header. Listing 6.20 shows the machine's header and initialisation. Note that, as a matter of readability, we often give the invariants of one of the controllers; the others are abbreviated with dots and can be obtained by switching the indexes.

The machine first defines variables for each system, and invariants `inv1-8` and `inv13-14` define their associated types and basic properties. `inv9-10` encode the properties of the simulated variables (h_i^{sim}) that must not drift away too far from their real counterpart.

`inv11-12` gives an additional interesting properties, built on the principle explained in Section 6.4.2.1. The idea is that, if the estimated volume, despite the potential error Δ_i^{sim} is in the bounds, then the actual volume is as well.

Finally, `inv15` and `inv16` encode the gluing invariant of this refinement, for the continuous and discrete states, respectively.

In addition to this header, the initialisation is straightforward: both sub-components are in *stable* mode with a given initial amount of liquid inside. The emulated variables are given the same initial value as the starting point (but could also get another value if the invariants are not violated). A witness is provided for the substituted variables, following the gluing invariant.

```
MACHINE 2Ctrl_2Tanks REFINES
    AbstractTank
SEES 2Ctrl_2Tanks_Ctx
VARIABLES $t$ , $x^C_{s,1}$ , $x^C_{s,2}$ , $h_1$ , $h_2$ , $h_1^{sim}$ , $h_2^{sim}$
INVARIANTS
  inv1−4 : $h_1, h_2, h_1^{sim}, h_2^{sim} \in \mathbb{R} \mapsto \mathbb{R}$
  inv5−8 : $[0,t] \subseteq \mathrm{dom}(h_1)$ , ...
  inv9−10 : $\forall t^* \cdot t^* \in [0,t]$
    $\Rightarrow |h_1(t^*) - h_1^{sim}(t^*)| \leq \Delta_1^{sim}$ , ...
  inv11−12 : $\forall t^* \cdot t^* \in [0,t]$
    $\Rightarrow B_1 h_1(t^*) + B_2 h_2^{sim}(t^*) \geq V_{low} + B_2 \Delta_2^{sim}$
    $\wedge B_1 h_1(t^*) + B_2 h_2^{sim}(t^*) \leq V_{high} - B_2 \Delta_2^{sim}$ , ...
  inv13−14 : $x_{s,1} \in STATES$ , $x_{s,2} \in STATES$
  inv15 : $V =_{[0,t]} B_1 h_1 + B_2 h_2$
  inv16 : $x_s, x_{s,1}, x_{s,2} \in Policy$
```

```
INITIALISATION
WITH
  $V'$ : $V' =_{[0,t']} B_1 h_1 + B_2 h_2$
  $x'_s$ : $x'_s, x'_{s,1}, x'_{s,2} \in Policy$
THEN
  act1 : $t := 0$
  act2 : $x_{s,1}, x_{s,2} := Stable, Stable$
  act3 : $h_1, h_2 := 0 \mapsto h_1^0, 0 \mapsto h_2^0$
  act4 : $h_1^{sim}, h_2^{sim} := 0 \mapsto h_1^0, 0 \mapsto h_2^0$
END
```

Listings 6.20: 2 Controllers 2 Tanks – Machine Header

```
ctrl_transition_normal_1 REFINES
    ctrl_transition_normal
WHERE
  grd1 :
      $B_1 h_1(t) + B_2 h_2^{sim}(t) < V_{high} - B_2 \Delta_2^{sim}$
  grd2 :
      $V_{low} + B_2 \Delta_2^{sim} < B_1 h_1(t) + B_2 h_2^{sim}(t)$
THEN
  act1 : $x_s^1 := Normal$
END
```

```
ctrl_transition_normal_2 REFINES
    ctrl_transition_normal
WHERE
  grd1 :
      $B_1 h_1^{sim}(t) + B_2 h_2(t) < V_{high} - B_1 \Delta_1^{sim}$
  grd2 :
      $V_{low} + B_1 \Delta_1^{sim} < B_1 h_1^{sim}(t) + B_2 h_2(t)$
THEN
  act1 : $x_s^2 := Normal$
END
```

Listings 6.21: 2 Controllers 2 Tanks – Transition Events

Discrete Events. Listing 6.21 presents two transitions events of the machine. Following the pattern, these events are duplicates of the abstract `ctrl_transition_normal` transition event, and each event references only the variables of one controller at a time.

Following the remarks on guard strengthening, we need to ensure that, even if the emulated variables are the furthest allowed from their real counterpart, the guard is still at least as strong as the abstract one. Note that:

$$B_1 h_1(t) + B_2 h_2^{sim}(t) \leq V_{high} - B_2 \Delta_2^{sim}$$
$$\Rightarrow B_1 h_1(t) + B_2 (h_2^{sim}(t) + \Delta_2^{sim}) \leq V_{high}$$

With $|h_2(t) - h_2^{sim}(t)| \leq \Delta_2^{sim}$, we have $h_2^{sim}(t) + \Delta_2^{sim} \geq h_2(t)$ and deduce:

$$B_1 h_1(t) + B_2 h_2(t) \leq B_1 h_1(t) + B_2 (h_2^{sim}(t) + \Delta_2^{sim}) \leq V_{high}$$

We proceed symmetrically for controller 2 and for the other predicate (with V_{low}).

For the record, we give the sensing events of the machine (Listing 6.22). They are constructed in the same way as the transition events, and following the methodology of the many-to-many architecture pattern given in Section 6.4.

```
ctrl_sense_too_high_1 REFINES
    ctrl_sense_too_high
WHERE
  grd1 :
      $V_{high} - B_2\Delta_2^{sim} \leq B_1h_1(t) + B_2h_2^{sim}(t)$
THEN
  act1 : $x_s^1 := Emptying$
END
```

```
ctrl_sense_too_high_2 REFINES
    ctrl_sense_too_high
WHERE
  grd1 :
      $V_{high} - B_1\Delta_1^{sim} \leq B_1h_1^{sim}(t) + B_2h_2(t)$
THEN
  act1 : $x_s^2 := Emptying$
END
```

Listings 6.22: 2 Controllers 2 Tanks – Sensing Events

```
EVENT ctrl_actuate_pumps REFINES ctrl_actuate_pumps
ANY $s_1$ , $s_2$ , $t'$ , $io_1$ , $io_2$ , $h_1^{sim*}$ , $h_2^{sim*}$
WHERE
  grd0 : $t' \in \mathbb{R} \wedge t < t'$
  grd1 : $\mathbf{Feasible}([t,t'], h_1 \otimes h_2^{sim},$
           $\{\hat{h}_1, \hat{h}_2^{sim}, \hat{h}_1', \hat{h}_2^{sim'} \mid$
             $\mathbf{solutionOf}(\hat{h}_1', \mathbf{ode}(FlowODE(0, H_{1,max}, \delta_1^{in}, \delta_1^{out}, io_1), h_1(t), t))$
             $\wedge \hat{h}_2^{sim'} =_{[t,t']} = h_2^{sim*}\},$
           $\{\hat{h}_1, \hat{h}_2^{sim} \mid V_{low} + B_2\Delta_2^{sim} \leq B_1\hat{h}_1 + B_2\hat{h}_2^{sim} \wedge V_{high} - B_2\Delta_2^{sim} \geq B_1\hat{h}_1 + B_2\hat{h}_2^{sim}\})$
  grd2 : ... -- similar to grd1
  grd3 : $s_1, s_2 \in STATES$
  grd4 : $x_s^1 = s_1 \wedge x_s^2 = s_2$
  grd5 : $s_1, io_1, io_2 \in Policy$
  grd6 : $V_{low} + B_2\Delta_2^{sim} \leq B_1h_1(t) + B_2h_2^{sim}(t) \wedge V_{high} - B_1\Delta_1^{sim} \geq B_1h_1(t) + B_2h_2^{sim}(t)$
  grd7 : $V_{low} + B_1\Delta_1^{sim} \leq B_1h_1^{sim}(t) + B_2h_2(t) \wedge V_{high} - B_2\Delta_2^{sim} \geq B_1h_1^{sim}(t) + B_2h_2(t)$
WITH
  $V'$ : $V' =_{[0,t']} B_1h_1' + B_2h_2'$
  $s$ : $s, s_1, s_2 \in Policy$
  $eq$ : $\mathbf{Solvable}([t,t'], eq, \{V^* \mid V_{low} < V^* \wedge V^* < V_{high}\}) \wedge isFlowEq(s, [t,t'], eq, 0, V_{max})$
       $\wedge \mathbf{solutionOf}([t,t'], B_1h_1' + B_2h_2', eq)$
THEN
  act1 : $h_1, h_2, h_1^{sim}, h_2^{sim} :|_{t \to t'}$
           $\mathbf{solutionOf}([t,t'], h_1', \mathbf{ode}(FlowODE(0, H_{1,max}, \delta_1^{in}, \delta_1^{out}, io_1), h_1(t), t)) \wedge$
           $\mathbf{solutionOf}([t,t'], h_2', \mathbf{ode}(FlowODE(0, H_{2,max}, \delta_2^{in}, \delta_2^{out}, io_2), h_2(t), t)) \wedge$
           $h_1^{sim'} = h_1^{sim*} \wedge h_2^{sim'} = h_2^{sim*}$
           $\&\{\hat{h}_1, \hat{h}_2^{sim}, \hat{h}_1^{sim}, \hat{h}_2 \mid$
             $V_{low} + B_2\Delta_2^{sim} \leq B_1\hat{h}_1 + B_2\hat{h}_2^{sim} \wedge V_{high} - B_2\Delta_2^{sim} \geq B_1\hat{h}_1 + B_2\hat{h}_2^{sim} \wedge$
             $V_{low} + B_1\Delta_1^{sim} \leq B_1\hat{h}_1^{sim} + B_2\hat{h}_2 \wedge V_{high} - B_1\Delta_1^{sim} \geq B_1\hat{h}_1^{sim} + B_2\hat{h}_2$
           }
END
```

Listings 6.23: 2 Controllers 2 Tanks – Actuation Event

Continuous Events. Listing 6.23 shows the system's actuation, built from the concepts presented in the single-to-many pattern case study, but following the many-to-many architecture pattern. The goal of this actuation is to update the behaviour of the two continuous states and two emulated continuous state, according to the current discrete state of each controller.

The evolution domain ensures that the estimated global volume remains in the bound, including the allowed bounded imprecision.

A witness is given for *eq*, the same as for the single-to-many case study, so that we can establish simulation.

Note that the functions associated with the emulated variables are not given explicitly; the way they are calculated could be specified in a later refinement (linear/polynomial approximation, stochastic models, etc.).

6.5.5.3 Proofs

This model is associated with 152 POs in total, most of which are related to well-definedness (31%) of the use of the operators defined in the theories and to invariants (43%), and in particular typing invariants. The other POs relate to refinement mainly (21%), and to feasibility of actions and witnesses (5%).

Guard strengthening is addressed when presenting the discrete events: by narrowing the bounds, we ensure that, despite the imprecision caused by the estimation of the global state, the tanks are safely controlled.

Simulation of the Event-B machines is ensured by the particular shape of the witnesses; this witness is associated with a witness feasibility proof obligation.

6.6 Discussion

Architecture, in the sense of how the components of a hybrid system interact, is an essential part of hybrid system design. The work presented in this chapter formalises this concept of architecture in the form of a refinement pattern, allowing the introduction of various structures as a development step. These patterns are presented in the form of a refinement, to be used for any existing system, either as a whole or for one of its sub-component. This makes these patterns adaptable and useful, as they may be used at any point during the refinement chain, and can be composed in every ways to produce rich and complex architectures.

In particular, the many-to-many pattern can be seen as a model of a class of cyber-physical or autonomous systems, or in other words sets of hybrid systems that communicate with each other with decentralised control.

This work has been published in [Dup+19; Dup+20c]. The single-to-many pattern has been successfully applied to the case study of controlling the filling and emptying of two water tanks connected to a centralised controller. The many-to-many pattern has been applied to a similar case study, with two water tanks independently controlled by two controllers, communicating with bounded delay. The complete Event-B models for both cases studies are available in the appendix, Section B.4.

In this chapter, the case studies also demonstrate the use of *domain theories* to encapsulate domain-specific knowledge, and in particular hypotheses issued from physics, and constraints specific to the system being developed (e.g. communication delay).

Note that the presented patterns only allow the refinement of one abstract component with two concrete components. Achieving decomposition in more than two components is realised by successive applications of the patterns.

Instantiating the pattern directly for an arbitrary number of components (greater than two) can be achieved using additional theories, in particular a theory of vectors.

Chapter 7

Approximation

The design of hybrid systems revolves around the definition and handling of differential equations, complex mathematical objects that often have subtle properties, difficult to establish and exploit. In particular, many “real-world” hybrid systems involve complex differential equations, with properties that are difficult to handle for safety. Moreover, these differential equations often do not even have an analytical (i.e. explicit) solution.

Such equations are common in control theory, since the real physical world is difficult to model. Controller designers addressed this problem using various techniques, and in particular the safe reduction of the original problem to another problem, with better properties or even for which an analytical solution exists. One of these reduction techniques is *approximation*.

Instinctively, approximation is an operation or category of operations that safely substitutes a differential equation with another one, generally simpler or easier to handle, such that both equations model behaviours that are *close enough* from each other, while retaining properties (e.g. safety).

Approximation has numerous applications: it can be used to simplify a system in order to find an adequate control, to simulate it or to model-check it, or even to find (approximated) analytical solutions. It is widely used in control theory, and any framework that proposes to design hybrid systems shall make it available.

Approximation revolves around mathematical theories. In this chapter, we present a formalised approximation operation, relying on an extension of the classical Event-B refinement operation. The general idea is to relax traditional refinement in order to allow the systems to drift from each other, in a safe and controlled way, while retaining their properties, and in particular safety.

The idea of relaxing the relation between two systems has been explored by [GP07; GJP08]. In this work, the authors propose the concept of approximate (bi-)simulation relation, to simplify (abstract) a system by another (approximated) system while retaining some properties. This technique is set up in order to use hybrid model-checkers to verify properties on the system, which is otherwise unfruitful due to its nature (non-linear differential equations).

In [GP07; GJP08], the authors only establish the approximate (bi-)simulation and a form of guard strengthening (in fact, local invariants) to ensure the approximated system does not violate local properties the original system abides by. However, bi-simulation relation between a system and its approximated version is not formalised in a specific proof assistant or any other formal framework. Only mathematical justifications is provided.

We extend this approach: the approximate (bi-)simulation principle is taken one step further to define approximate *refinement*, and the overall framework for approximation is designed to carry out *proofs*, and in particular global invariants. In addition, our approach is completely formalised in Event-B, and supported by its associated IDE, Rodin.

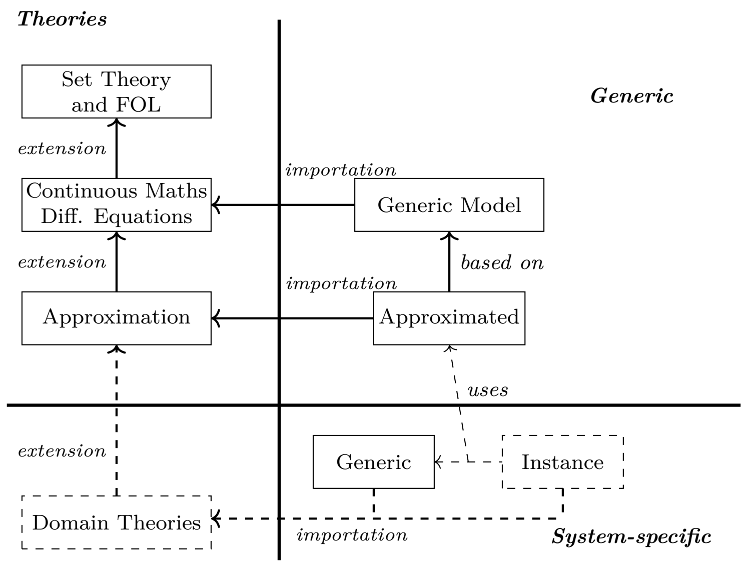


Figure 7.1: Framework – Approximation Pattern

This chapter investigates the formalisation and use of approximation in the context of our framework. Figure 7.1 gives a summary of the place of approximation with regard to the other components of the framework. Note that approximation is both a formal mathematical theory (*Approximation* theory) a development pattern for hybrid systems (*Approximated* pattern/refinement).

Section 7.1 presents a relaxed version of refinement that allows approximation as a refinement operation. Section 7.3 discusses the proofs associated with this refinement operation, and Section 7.4 shows how it is formalised in classical Event-B. Finally, Section 7.5 illustrates the use of approximation on two cases studies borrowed from litterature, and Section 7.6 concludes the chapter with a discussion on the contribution.

7.1 Handling Approximation as a Refinement Operation

Approximation is an operation that is carried out as a design step: the designer creates a model, and approximates it with another one. Approximation shall be established by proofs, and in particular the approximated model shall preserve properties of the original one, in particular safety.

We claim that refinement, as offered by refinement-based techniques such as Event-B, is capable of handling approximation of a system by another, explicitly, while providing proof obligations ensuring its correctness, and the preservation of the original model's properties.

The actual concept subject to approximation being functions, the idea is to *glue* continuous states using an approximate operator, instead of gluing them using (exact) equality or set-membership.

In addition, care should be taken whenever this continuous state appears in the model; in continuous events, of course, which require specific witnesses to achieve simulation (and gluing invariant preservation), but also when continuous state is *sensed*.

7.1.1 Continuous Gluing Invariant

The gluing invariant links the state of an abstract machine to the state of a (refining) concrete machine. In the context of hybrid systems, the gluing invariant is naturally split in two parts: one part related to the discrete state, and the other one related to the continuous state.

Discrete state gluing invariant is not different from the usual gluing invariant of discrete systems; it is expressed as a predicate linking abstract and discrete state variables.

The continuous state gluing invariant has a more restricted structure. It involves (piece-wise) continuous (plant) state variables that are continuous functions, and it must be maintained by continuous events.

Overall, a continuous gluing invariant can be expressed as follows:

Definition 8 (Continuous Gluing Invariant)**.** *Let S^A (resp. S^C) be the state-space and $x_p^A \in \mathbb{R}^+ \nrightarrow S^A$ (resp. $x_p^C \in \mathbb{R}^+ \nrightarrow S^C$) the continuous state of the abstract (resp. concrete) machine. Let $\mathcal{O} \in S^C \leftrightarrow S^A$ be a relation linking both abstract and concrete state spaces, generally called **observation mapping** between the two systems. Finally, let $t \in \mathbb{R}^+$ be the machines' time variable. A **continuous gluing invariant** between the abstract and the concrete model is of the form:*

$$x_p^A \underset{[0,t]}{\in} \mathcal{O} \circ x_p^C$$

In practice, such invariants have a simpler, functional form, e.g.:

$$x_p^A \underset{[0,t]}{=} o \circ x_p^C$$

where $o \in S^C \rightarrow S^A$ is a function instead of a relation.

By substituting the operators used in the gluing invariant predicate ($\in$ or $=$) with their approximate version ($\in^\delta$ or $\approx^\delta$) defined in Section 4.2.1, an approximate version of the invariant is obtained:

Definition 9 (Approximate Continuous Gluing Invariant)**.** *With the same parameters as in Definition 8, and with $\delta \in \mathbb{R}^+$, an **approximate continuous gluing invariant** between the abstract and the concrete model is of the form:*

$$x_p^A \underset{[0,t]}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^C$$

Again, this invariant may have a simpler form using a function o in place of the relation $\mathcal{O}$, i.e.:

$$x_p^A \underset{[0,t]}{\overset{\delta}{\approx}} o \circ x_p^C$$

The *approximate refinement operation* is a refinement where the continuous gluing invariant is approximated.

In essence, approximate refinement relates to continuous states only (i.e. continuous variables of the system). It does not *add* events or actions. Consequently, it is encoded using standard data refinement, and is thus written in standard Event-B (plus the relevant theories).

7.2 Approximation Pattern

The approximation pattern carries out an approximation operation on a given system. It is presented as a *refinement* (on the generic model) and relies on the use of the approximate gluing invariant discussed in Section 7.1.1, Definition 9.

Since approximation only relates to the continuous state, only the events that access continuous variables, namely `Actuate/Behave` and `Sense` are impacted by this type of refinement.

In the following, objects relating to the *abstract* model are denoted with a superscript A, and objects relating to the *concrete* model by a superscript C.

7.2.1 Variables, Invariants and Initialisation

Two machines are defined: M^A, the abstract one is the generic model (Chapter 5), and M^C, the concrete one, refines M^A.

```
MACHINE $M^A$
VARIABLES $t$ , $x_s$ , $x_p^A$
INVARIANTS
    inv1 : $t \in \mathbb{R}^+$
    inv2 : $x_s \in STATES$
    inv3 : $x_p^A \in \mathbb{R} \nrightarrow S^A$
    inv4 : $[0,t] \subseteq \mathrm{dom}(x^A)$
```

```
MACHINE $M^C$ REFINES $M_A$
VARIABLES $t$ , $x_s$ , $x_p^C$
INVARIANTS
    inv3 : $x_p^C \in \mathbb{R} \nrightarrow S^C$
    inv4 : $[0,t] \subseteq \mathrm{dom}(x_p^C)$
    inv5 : $\boldsymbol{x_p^A \in^{\delta}_{[0,t]} \mathcal{O} \circ x_p^C}$
```

Listings 7.1: Machines' Header

Listing 7.1 gives the header of each machine. The machines' variables are the one of the generic model: time t, discrete state x_s and continuous state $x_p^{A/C}$. Note that this refinement only affects the system's continuous state. In practice the discrete state may also be handled during the refinement process, but this is out of approximate refinement.

Invariants `inv1-4` are also taken straight from the generic model. They provide a type to each variable and guarantees that the continuous state variable is defined at least on $[0, t]$.

The central part of this refinement is invariant `inv5` of M^C: it defines the approximate continuous gluing invariant as discussed in Section 7.1.1.

```
INITIALISATION$^A$
THEN
    act1 : $t := 0$
    act2 : $x_s :\in$ STATES
    act3 : $x_p^A :\in \{0\} \to S^A$
END
```

```
INITIALISATION$^C$ REFINES
      INITIALISATION$^A$
WITH
    $x_p^{A\prime}$ : $x_p^{A\prime}(0) \in^{\delta} \mathcal{O}[\{x_p^{C\prime}(0)\}]$
THEN
    act1 : $t := 0$
    act2 : $x_s :\in$ STATES
    act3 : $x_p^C :\in \{0\} \to S^C$
END
```

Listings 7.2: Initialisation

Listing 7.2 presents the initialisation of the machines. It follows the generic model.

A witness is provided for $x_p^{A\prime}$ since x_p^A is subtituted with x_p^C in the refinement. This witness follows the (approximate) gluing invariant. Note that it is unfolded for better readability.

7.2.2 Continuous Events

Continuous events represent the core part of the model impacted by approximate refinement. In this section, we only focus on the `Actuate` event, since `Behave` is similar.

```
Actuate^A
ANY 𝒫^A , s , H^A , t'
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}^A \in (\mathbb{R}^+ \twoheadrightarrow S^A) \times (\mathbb{R}^+ \twoheadrightarrow S^A)$
    grd2 : $\mathbf{Feasible}(x_p^A, [t,t'], \mathcal{P}^A, H^A)$
    grd3 : $s \subseteq \mathrm{STATES}$
    grd4 : $x_s \in s$
    grd5 : $H^A \subseteq S^A$
    grd6 : $x_p^A(t) \in H^A$
THEN
    act1 : $x_p^A :|_{t \to t'} \mathcal{P}^A(x_p^A, x_p^{A\prime}) \ \&\ H^A$
END
```

```
Actuate^C REFINES Actuate^A
ANY 𝒫^C , s , H^C , t'
WHERE
    grd0 : $t' > t$
    grd1 : $\mathcal{P}^C \in (\mathbb{R}^+ \twoheadrightarrow S^C) \times (\mathbb{R}^+ \twoheadrightarrow S^C)$
    grd2 : $\mathbf{Feasible}(x_p^C, [t,t'], \mathcal{P}^C, H^C)$
    grd3 : $s \subseteq \mathrm{STATES}$
    grd4 : $x_s \in s$
    grd5 : $H^C \subseteq S^C$
    grd6 : $x_p^C(t) \in H^C$
WITH
    $x_p^{A\prime}$ : $x_p^{A\prime} \in_{[0,t']}^{\delta} \mathcal{O} \circ x_p^{C\prime}$
THEN
    act1 : $x_p^C :|_{t \to t'} \mathcal{P}^C(x_p^C, x_p^{C\prime}) \ \&\ H^C$
END
```

Listings 7.3: Actuate Event

The system's actuation is shown in Listing 7.3. The witness for $x_p^{A\prime}$ is needed to prove continuous gluing invariant preservation, as well as the simulation proof obligation arising from this event. When instantiating the pattern, a proof obligation of *witness feasibility* is generated, requiring the existence of $x_p^{A\prime}$ and $x_p^{C\prime}$ such that predicate $x_p^{A\prime} \in_{[0,t']}^{\delta} \mathcal{O} \circ x_p^{C\prime}$ holds.

Witnesses for H^A and $\mathcal{P}^A$ are not given here as, in practice, such witnesses depend on the actual system being designed. The refinement of these parameters is discussed in Section 7.3 nonetheless, when studying approximate refinement correctness.

7.2.3 Sensing

Sensing events handle the system's continuous state, and are thus impacted by approximate refinement.

Listing 7.4 presents the system's sensing event. In contrast with the presentation of the sensing event in Chapter 5, the guard on the continuous state is given under the form of a set, $\mathcal{G}^{A/C}$. This is useful when dealing with the guard strengthening proofs (Section 7.3.2).

Note that it is always possible to express a guard given as a predicate as a set using the axiom of comprehension.

Approximate refinement of the sensing event is performed by substituting the abstract continuous state variable with the concrete one, and updating the guard adequately. The design of $\mathcal{G}^C$ relative to $\mathcal{G}^A$ is the central part of this refinement, as care should be taken that guard strengthening holds for this event, that is:

$$x_p^C(t) \in \mathcal{G}^C \Rightarrow x_p^A(t) \in \mathcal{G}^A$$

```
EVENT Sense^A
ANY s , 𝒢^A
WHEN
    grd1 : s ∈ ℙ1(STATES)
    grd2 : x_p^A(t) ∈ 𝒢^A
THEN
    act1 : x_s :∈ s
END
```

```
EVENT Sense^C REFINES
      Sense^A
ANY s , 𝒢^C
WHEN
    grd1 : s ∈ ℙ1(STATES)
    grd2 : x_p^C(t) ∈ 𝒢^C
THEN
    act1 : x_s :∈ s
END
```

Listings 7.4: Sensing Event

This proof depends on the form of the variables, there evolution and the form of the guards. However, a sufficient condition to guard strengthening is discussed in Section 7.3.2, that helps to design $\mathcal{G}^C$ so that refinement is correct.

7.3 Proofs

The approximation pattern is described under the form of a refinement, but its essential features are found in the proof obligations associated with it. The idea is that, when the pattern is applied, the generated proof obligations follow a set schema. In this section, we study and discuss this schema, and use it to give a general method to handle these POs.

In general, it is impossible to give a holistic solution to such POs. However, thanks to the properties of the operators used in the model, it is possible to simplify these POs, and establish, "for free", some parts of the proofs. In other words, POs associated to refinement can be *refactored*, specialised for approximate refinement.

7.3.1 Simulation

Simulation (see Section 2.3.3.2 is a proof obligation associated with refinement. It ensures the behaviour of the refining machine is allowed by the specification of the abstract machine.

We reacll the general form of the simulation proof obligation:

$$A \wedge I^A \wedge I^C \wedge G^C \wedge W \wedge BAP^C \Rightarrow BAP^A$$

In the particular case of continuous events, restricted in their writing, before-after predicates $\mathcal{BAP}^{A/C}$ are substituted in the PO:

$$\begin{aligned} A \wedge I^A \wedge I^C \wedge G^C \wedge x_p^A \underset{[0,t]}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^C \wedge x_p^{A\prime} \underset{[0,t']}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^{C\prime} \wedge CBAP(t,t',x_p^C,x_p^{C\prime},\mathcal{P}^C,H^C) \\ \Rightarrow CBAP(t,t',x_p^A,x_p^{A\prime},\mathcal{P}^A,H^A) \end{aligned} \tag{7.1}$$

The *CBAP* operator on the right-hand side of the implication is unfolded using the definition given in Operator 7:

$$\begin{aligned} \ldots \Rightarrow [0,t] \lhd x_p^{A\prime} = [0,t] \lhd x_p^A && (PP) \\ \mathcal{P}^A([0,t] \lhd x_p^A, [t,t'] \lhd x_p^{A\prime}) && (PR) \\ \land \forall t^* \in [t,t'], x_p^{A\prime}(t^*) \in H^A && (LI) \end{aligned}$$

In this context, the *past preservation* property of the operator (PP) is true by the way the operator is constructed and intended. In practice however, this part of the proof is ensured by the provided witness (see Section 7.4).

It remains to establish that 1) the predicate in the abstract CBAP is approximately simulated by the predicate in the concrete one and (approximate continuous predicate simulation, $ACPSIM$) and 2) the local invariant (evolution domain) of the abstract machine is implied by the one of the concrete event, modulo approximations (approximate local invariant strengthening, $ALIS$). To summarise, we write:

$$\begin{aligned} A \land I^A \land I^C \land G^C \land x_p^A \underset{[0,t]}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^C \land x_p^{A\prime} \underset{[0,t']}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^{C\prime} \land CBAP(t,t',x_p^C,x_p^{C\prime},\mathcal{P}^C,H^C) \\ \Rightarrow \mathcal{P}^A([0,t] \lhd x_p^A, [t,t'] \lhd x_p^{A\prime}) && (ACPSIM) \\ \land \forall t^* \in [t,t'], x_p^{A\prime}(t^*) \in H^A && (ALIS) \end{aligned}$$

Approximate local invariant strengthening ($ACPSIM$) is discussed more thoroughly in Section 7.3.2. Approximate continuous predicate simulation ($ACPSIM$) depends on the form of the predicates used in the operators.

In practice, models use differential equations and the $:\sim_{t \to t'}$ operator both in the abstract and continuous actuation event. In this case, the PO effectively consists in ensuring that the solutions of these differential equations remain close from each other with regard to the approximation.

7.3.2 Guard and Invariant Strengthening

A number of proof obligations associated with refinement revolve around the general concept of *predicate strengthening.* Formally, such proof obligations are of the form:

$$\ldots \land P^C \Rightarrow P^A$$

This type of predicate appears, in particular, in guard strengthening proof obligations. It also appears in approximate local invariant strengthening, and may appear in invariant preservation POs of the concrete machine.

Note that, in our particular case, such POs and therefore such strengthening predicates relate to continuous states, and are expressed under the form

In our particular case, those predicates relate to continuous states, and can generally be expressed under the form $x_p \in \mathcal{H}$, where $\mathcal{H} \subseteq S$. Thus, strengthening proof obligations is of the form:

$$\Gamma \land x_p^A \underset{[0,t]}{\overset{\delta}{\in}} \mathcal{O} \circ x_p^C \land x_p^C \in \mathcal{H}^C \Rightarrow x_p^A \in \mathcal{H}^A \qquad (7.2)$$

An useful theorem is then proposed and proved that is used to handle this particular PO schema and simplify it.

Theorem 5 (Sufficient Condition for Predicate Strengthening). *Let $\delta \in \mathbb{R}^+$. Let $\mathcal{H}^A \subseteq S^A$ and $\mathcal{H}^C \subseteq S^C$. Given $x^C \in \mathcal{H}^C$ and $x^A \in^\delta \mathcal{O}[\{x^C\}]$. Then, a sufficient condition for $x^A \in \mathcal{H}^A$ is to have:*

$$\mathcal{O}[\mathcal{H}^C] \subseteq \mathcal{S}_\delta(\mathcal{H}^A)$$

Proof. $x^C \in \mathcal{H}^C$, i.e. $\{x^C\} \subseteq \mathcal{H}^C$, so it follows that $\mathcal{O}[\{x^C\}] \in \mathcal{O}[\mathcal{H}^C]$. By inclusion following the left-hand side of the implication, we have $\mathcal{O}[\{x^C\}] \subseteq \mathcal{S}_\delta(\mathcal{H}^A)$.
Besides, we have $x^A \in^\delta \mathcal{O}[\{x^C\}]$ so, by definition, $\exists y \in \mathcal{O}[\{x^C\}], d(x^A, y) \leq \delta$.
As $y \in \mathcal{O}[\{x^C\}]$ and using Theorem 2 of Section 4.2.2, we obtain $\overline{\mathcal{B}}(y, \delta) \subseteq \mathcal{H}^A$. Additionally, $d(x^A, y) \leq \delta$ is equivalent to $x^A \in \overline{\mathcal{B}}(y, \delta)$ (by definition of the closed ball).
It follows that $x^A \in \mathcal{H}^A$. □

The idea of this theorem is to guide the design of $\mathcal{H}^C$ so that the resulting predicate $x^C \in \mathcal{H}^C$ is stronger than $x^A \in \mathcal{H}^A$, taking into account the approximate gluing invariant $x^A \in^\delta \mathcal{O}[\{x^C\}]$. Concretely, it formalises the idea that, if the abstract system shall not cross a given border, then the concrete system, modulo observation, shall not cross this border plus or minus δ.

Note that this theorem is generalised to functions using universal quantification over t and substituting x^A (resp. x^C) with $x_p^A(t)$ (resp. $x_p^C(t)$).

7.3.3 Approximation Well-Definedness

The general problem of predicate strengthening raises an interesting remark. First of all, let us define some complementary notions related to metric spaces.

Definition 10 (Diameter of a Set). *Given (E, d) a metric space and $S \subseteq E$ a subset of E, the **diameter** of S, denoted $\text{diam}(S)$, is the greatest distance between any two points of S. Formally:*

$$\text{diam}(S) = \sup_{x,y \in S} d(x, y)$$

We note two particular properties of diameter, one associated with subsets and the other one with closed balls.

Property 5 (Diameter of a Subset). *Let $S \subseteq E$ and $T \subseteq S$ two sets. Then $\text{diam}(T) \leq \text{diam}(S)$.*

Property 6 (Diameter of a Closed Ball). *The diameter of the closed ball $\overline{\mathcal{B}}(x, \delta)$ of centre $x \in E$ and radius $\delta \in \mathbb{R}^+$ is equal to $\mathbf{2 \times \delta}$.*

These two properties are used to establish the following theorem:

Theorem 6 (Emptiness of δ-Shrinking). *Let $\delta \in \mathbb{R}^+$ and $S \subseteq E$ such that $\text{diam}(S) < 2 \times \delta$. Then $\mathcal{S}_\delta(S)$ is **empty**.*

Proof. We suppose $S \subseteq E$ with $\text{diam}(S) < 2 \times \delta$. Let $x \in \mathcal{S}_\delta(S)$; then by Theorem 2, we have $\overline{\mathcal{B}}(x, \delta) \subseteq S$.
By using Property 5, we deduce that $\text{diam}(\overline{\mathcal{B}}(x, \delta)) \leq \text{diam}(S)$. Using Property 6, we can replace the value of $\text{diam}(\overline{\mathcal{B}}(x, \delta))$ with $2 \times \delta$, and we write: $2 \times \delta \leq \text{diam}(S)$, which is incompatible with the hypothesis that $\text{diam}(S) < 2 \times \delta$. □

The consequence of this theorem is to highligh a particular case of approximate refinement. Let $\delta \in \mathbb{R}^+$ be a given approximation error. Let M^A be an abstract machine with a continuous state variable $x_p^A \in \mathbb{R} \nrightarrow S^A$ that features an event with a guard of the form $x_p^A(t) \in \mathcal{G}^A$, with $\mathrm{diam}(\mathcal{G}^A) < 2 \times \delta$. Let M^C be a concrete machine that approximately refines M^A, with a continuous state variable $x_p^C \in \mathbb{R} \nrightarrow S^C$, such that $x_p^A \in_{[0,t]}^{\delta} \mathcal{O} \circ x_p^C$.

The event with guard $\mathcal{G}^A$ has to be refined with a stronger guard $\mathcal{G}^C$, with $\mathcal{O}[\mathcal{G}^C] \subseteq \mathcal{S}_\delta(\mathcal{G}^A)$. However, by Theorem 6, we know that $\mathcal{S}_\delta(\mathcal{G}^A)$ is *empty*. Thus, we can only have $\mathcal{O}[\mathcal{G}^C] = \emptyset$.

Concretely, this means that the only correct approximate refinement of the given event is *never triggered*. Equivalently, given this broad margin of error δ, there is no guarantee that the event is triggered at a safe moment (i.e. at a moment the abstract event would trigger).

More generally speaking, this raises an interesting point: if δ is not *suitable* (here, small enough) with regard to guards and local invariants, then there does not exist a *correct approximate refinement* of the abstract machine that retains its features (events).

This is an example of a problem of *approximation well-definedness*: for any given machine, there exists values of δ for which a correct approximate refinement removes features from the system.

In practice, approximation is established for specific systems using domain-specific knowledge (i.e. ad hoc theorems). This knowledge gives properties of or conditions on δ that allow establishing that the proposed strengthened predicates are not empty.

7.4 Formalisation

Similar to other patterns, the actual use of approximation requires some Event-B formalisation effort. The operators and properties proposed in Section 4.2 have been encoded in Event-B theories (see Section 4.3.2.5). They are used throughout the models.

Limitation of Witnesses Using the approximation pattern, situations where multiple substituted variables are independent on each others may appear. Typically, if (x_1^A, x_2^A) is being glued to x^C with x_1^A and x_2^A linked by a particular relation, then in an event that refines an event where both (x_1^A, x_2^A) and x^C are modified, a witness is required for both $x_1^{A\prime}$ and $x_2^{A\prime}$. Unfortunately, Event-B does not allow witnesses to be defined "together" (something like $x_1^{A\prime}, x_2^{A\prime} : \ldots$), which means that we cannot express the particular relation links these variables in one single witness predicate. This may lead to invariant violation or prevent guard strengthening.

Note that this is a purely modelling limitation (and not a *formal* one). A common workaround for this Event-B limitation is to keep the variables and parameters of the abstract machine in the concrete one (sometimes referred to as *extension*).

This does not impact the proving process. For modelling, events become *extensions* of the abstract events. In this case, witnesses are not needed anymore, or are directly expressed in guards, invariants, or in the predicate of the event's action.

Witness Design Designing correct witnesses is required for being able to prove refinement correctness. When presenting the pattern, the witnesses that are given encompass implicit information. In particular when replacing continuous variables, in order to prove the simulation proof obligation, it is required to add the actual behaviour of the continuous variable (i.e. using a $CBAP*$ operator) to the witness, as well as typing information for well-definedness.

Therefore, the actuation given in Listing 7.3 is formalised as shown on Listing 7.5.

```
EVENT Actuate^C REFINES Actuate^A
ANY $\mathcal{P}^C$ , $s$ , $H^C$ , $tp$
WHERE . . .
WITH
    $x_p^{A\prime}$ :
        $x_p^{A\prime} \in \mathbb{R} \rightarrow\!\!\!\!\!+ S^A \wedge [0, t'] \subseteq \mathrm{dom}(x_p^{A\prime}) \wedge$
        $CBAP(t, t', x_p^A, x_p^{A\prime}, \mathcal{P}^A, H^A)$
. . .
```

Listings 7.5: Encoding of the Approximated Actuation

7.5 Application to Case Studies

To illustrate the use of the approximation pattern, we propose the developments of two case studies. The first, detailed in Section 7.5.1, is borrowed from the work of [FGP07; GJP08]. It relate to the control of a robot that must visit multiple targets while remaining in a set area. The second case study, presented in Section 7.5.2, addresses the problem of the inverted pendulum.

7.5.1 Planar Robot Case Study

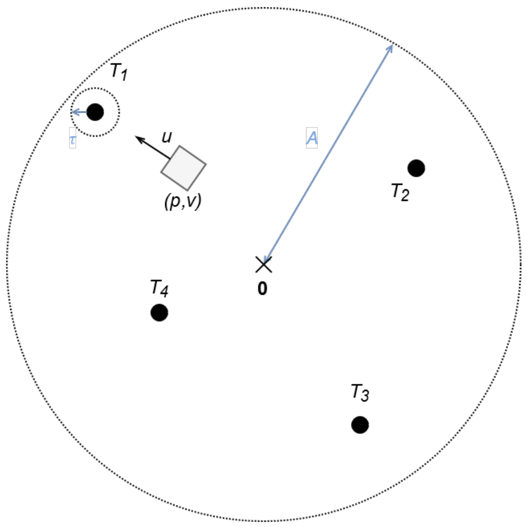


Figure 7.7: Planar Robot Typical Scenario

A robot evolves on a plane and must visit various targets, while remaining in a definite area. The robot's actual dynamics are rich and complex (e.g. taking into accounts friction, inertia, etc.), and require adequate control. However, these same dynamics are hard to handle during the proving process. The case study is summarised on Figure 7.7.

The idea is to model a version of the system with simpler dynamics that allows establishing the required properties of the system (i.e. safety). Approximate refinement is then used to safely concretise the system with the actual dynamics that approximate the simpler ones. The concrete model is closer to implementation, and retain the abstract system's properties.

This case study is borrowed from the work of [FGP07], and has been studied in [GJP08] where it has been addressed using approximate simulation and model-checking. In this work, we propose a proof-based approach, based on the results of Girard and Pappas, to propose a correct-by-construction version of the model.

7.5.1.1 Robot Theory

```
THEORY RobotTheory IMPORT Approximation
OPERATORS
    ...
    FO2DSystem expression (t_0: ℝ, y_0: ℝ × ℝ)
      direct definition caode(..., y_0, t_0)
    SO2DSystem expression (K: ℝ, t_0: ℝ, x_0: (ℝ × ℝ) × (ℝ × ℝ))
      direct definition caode(..., x_0, t_0)
    ...
    PointwiseControl expression (D: ℙ(ℝ), u: ℝ × ℝ, t_0: ℝ)
      well−definedness condition t_0 ∈ D
      direct definition (λt · t ∈ D ∧ t > t_0 | u)
    SlopedControl expression (D: ℙ(ℝ), v: ℝ × ℝ, t_0: ℝ)
      well−definedness condition t_0 ∈ D
      direct definition (λt · t ∈ D ∧ t > t_0 | (u, (t − t_0) · v)
    ...
AXIOMATIC DEFINITIONS
  OPERATORS
      planar_distance expression (x: ℝ × ℝ, y: ℝ × ℝ) : ℝ
  AXIOMS
      ...
END
```

Listings 7.6: Robot Theory

Before modelling the robots, we first define a theory to handle their specific properties. An extract of this theory is given in Listing 7.6. This theory mainly defines the controlled differential equations corresponding to the two models: *FO2DSystem* for the **f**irst-**o**rder robot system, and *SO2DSystem* for the **s**econd-**o**rder system. It also provides an adequate control, point-wise for the first order and "sloped" for the second order.

We also define an algebraic distance on $\mathbb{R} \times \mathbb{R}$, used throughout the model (typically to detect the distance between the robot and a target or the end of the area). The operator, *planar_distance*, is expressed axiomatically to avoid any assumption on the distance. It is associated with all the usual properties of distances (symmetry, separation, etc.). In the following, we will simply denote it d, for readability.

Finally, the theory provides an extensive number of axioms to qualify the defined controlled system and associated control: controllability, solvability with given control. It also gives the conditions under which the two differential equations are approximated, as well as the associated approximation bound, based on Girard and Pappas' work.

7.5.1.2 Abstract Robot

In this first model, we design a robot that is able to move on a 2D plane and visits targets, while staying in a designated area. The robot is modelled with simple dynamics that allows establishing required properties of the system, and in particular that the robot remains in the same areas (safety) and effectively visits the given targets (correction).

Note that this model is obtained by refining the generic model, presented in Chapter 5.

Preliminary Study

Plant Description. The controlled plant is the robot, modelled by its position $p^A \in \mathbb{R}^+ \to \mathbb{R} \times \mathbb{R}$ ($p^A(t) = (p^A_x(t), p^A_y(t))$). It evolves in a given area, a *closed ball* centered on $\mathbf{0} = (0, 0)$ and of fixed radius $A \in \mathbb{R}$, $A > 0$; formally:

$$\forall t, p^A(t) \in \overline{\mathcal{B}}(\mathbf{0}, A) \equiv d(p^A(t), \mathbf{0}) \leq A$$

The goal of the robot is to visit *targets* that are modelled as points of the plane $T_i \in \mathbb{R} \times \mathbb{R}$ ($i \in I$, where I is an arbitrary countable set). A target is considered to be "visited" when the robot is close enough to it by a fixed threshold $\tau > 0$:

$$T_i \text{ visited} \Leftrightarrow d(p^A, T_i) < \tau$$

Once a target is visited, a new target is given to the robot, and so on and so forth. There is no assumption on how and when those targets are given as this is out of scope of this case study. Note however that, to to ensure that the robot is always able to visit every target, we require that:

$$\forall i \in I, d(\mathbf{0}, T_i) < T - \tau$$

In other words, targets are always "far enough" from the allowed area's border so that, if the robot is close by τ from the target, it necessarily remains in this area.

To control the robot, the controller can issue a command $u^A \in \mathbb{R} \to \mathbb{R} \times \mathbb{R}$ that orients the robot and gives it a speed. The robot is thus modelled using the following controlled ODE:

$$\dot{p}^A = u^A \tag{7.3}$$

This is equivalently written using a (controlled) ODE function: $f^A(t, p^A, u^A) = u^A$. Speed of the robot, $\|u^A\| = d(\mathbf{0}, u^A)$, is bounded by a constant, denoted $\mu > 0$.

Controller Description. The role of the controller is to acknowledge the targets given to the robot and to issue commands to the robot in order to reach the target. At this point, targets and directions are given directly. Note that a later refinement may introduce the way by which the robot obtains each target and computes its direction, but this is not required for this particular part of the case study.

The controller stores two discrete states: the current target $T_i \in \mathbb{R}^2$ and the current general direction $Dir \in \mathbb{R}^2$. It senses when a target is visited ($d(p^A(t), T_i) \leq \tau$), and retrieves the next target to visit, T_j. Once the new target is known, the controller computes a new direction, used to control the robot.

The controller may also spuriously change its target and direction.

Note that the control for the plant (u^A) is "derived" from the direction ($Dir \in \mathbb{R}^2$) as so:

$$\forall t, u^A(t) = Dir$$

Requirements. The system's requirement can be summed up as so:

FUN1 The robot moves on a plane; it follows a direction *Dir* and its speed is bounded;
SAF1 The robot remains within the designated area;
FUN2 The robot visits a set of given targets, located in this area;

Event-B Development

```
CONTEXT Robot_0_Ctx
CONSTANTS $\mu$ , $\tau$ , $p_0$ , $A$
AXIOMS
    axm1 :  $\mu \in \mathbb{R} \land \mu > 0$
    axm2 :  $p_0 \in \mathbb{R} \times \mathbb{R}$
    axm3 :  $d(p_0, \mathbf{0}) \leq A$
    axm4 :  $\tau \in \mathbb{R}^+$
    axm5 :  $A \in \mathbb{R}^+$
    axm6 :  $\tau < A$
END
```

Listings 7.7: Abstract Robot – Context

Constants and Axioms. In addition to the robot theory defined in Section 7.5.1.1, a context is defined. An extract of this context is given in Listing 7.7. It contains all the necessary constants for the model: A, τ, μ, as well as an initial position p_0. Axioms are added to ensure the correct properties of these constants (e.g. $d(p_0, \mathbf{0}) \leq A$).

```
MACHINE Robot_0 REFINES Generic
SEES Robot_0_Ctx
VARIABLES $t$ , $p^A$ , $T$ , $Dir$
INVARIANTS
    inv1 :  $p^A \in \mathbb{R} \nrightarrow \mathbb{R} \times \mathbb{R}$
    inv2 :  $[0, t] \subseteq \mathrm{dom}(p^A)$
    inv3 :  $T \in \mathbb{R} \times \mathbb{R} \land d(T, \mathbf{0}) < A - \tau$
    inv4 :  $Dir \in \mathbb{R} \times \mathbb{R} \land \|Dir\| \leq \mu$
    inv5 :  $x_p = p^A$
    inv6 :  $x_s = T \mapsto Dir$
    inv7 :  $\forall t^* \cdot t^* \in [0, t] \Rightarrow d(p^A(t^*), \mathbf{0}) \leq A$
```

```
INITIALISATION
WITH
    $x'_p$ :  $x'_p = p^{A\prime}$
    $x'_s$ :  $x'_s = T' \mapsto Dir^{A\prime}$
THEN
    act1 :  $t := 0$
    act2 :  $T :| T' \in \mathbb{R} \times \mathbb{R} \land d(T, \mathbf{0}) \leq A - \tau$
    act3 :  $Dir = \mathbf{0}$
    act4 :  $p^A := \{0 \mapsto p_0\}$
END
```

Listings 7.8: Abstract Robot – Machine's Header and Initialisation

Machine Header. Listing 7.8 presents the machine's header and initialisation. The variables of the system are defined, and are associated to essential properties in the preliminary study. In particular, `inv5` and `inv6` glue the abstract (generic) and concrete state, and `inv7` formalises the system's safety invariant **SAF1**: the robot remains in the designated area.

Initialisation is straightforward: the robot starts with null speed and direction, at initial coordinates p_0 and with an arbitrary (valid) target.

```
EVENT sense_close_enough REFINES
    Sense
ANY $T_{next}$ , $Dir_{next}$
WHERE
    grd1 : $d(T, p^A(t)) \leq \tau$
    grd2 : $T_{next} \in \mathbb{R} \times \mathbb{R} \land d(\mathbf{0}, T_{next}) \leq A - \tau$
    grd3 : $Dir_{next} \in \mathbb{R} \times \mathbb{R} \land \|Dir_{next}\| \leq \mu$
WITH
    $s$ : $s = \{T_{next} \mapsto Dir_{next}\}$
    $p$ : $p = \{T \mapsto Dir\} \times \mathbb{R} \times \{\hat{p} \mid d(\hat{p}, T) \leq \tau\}$
    $x'_s$ : $x'_s = T_{next} \mapsto Dir_{next}$
THEN
    act1 : $T := T_{next}$
    act2 : $Dir := Dir_{next}$
END
```

```
EVENT transition_change_direction
    REFINES Transition
ANY $Dir_{new}$
WHERE
    grd1 : $Dir_{new} \in \mathbb{R} \times \mathbb{R} \land \|Dir_{new}\| \leq \mu$
WITH
    $s$ : $s = \{T \mapsto Dir_{new}\}$
    $x'_s$ : $x'_s = T \mapsto Dir_{new}$
THEN
    act1 : $Dir := Dir_{new}$
END
```

Listings 7.9: Abstract Robot – Machine's Discrete Events

Discrete Events. Listing 7.9 presents two discrete events of the machine. The `sense_close_enough` event senses when the robot is close enough from the target (thus considering the target as being visited). It assigns the state of the system a new target and a new direction. Guard `grd1` models the sensing, while the remaining guards ensure invariants are preserved. Note that there is no assumption on how the new target and direction are obtained.

The transition event `transition_change_direction` allows the controller to change its direction (with a new valid one). Note that there exists a `transition_change_target` event to change the current target, not given here as it is basically identical.

Witnesses are given in both events, following their guards.

```
EVENT actuate_movement REFINES Actuate
ANY $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathbf{Solvable}([t, t'], p^A, \mathbf{withControl}([t, t'],$
               $FO2DSystem(p^A(t), t), PointwiseControl([t, t'], Dir, t)),$
               $\{\hat{p} \mid d(\hat{p}, T) > \tau \land d(\hat{p}, \mathbf{0}) \leq A\})$
    grd2 : $d(p^A(t), T) > \tau$
WITH
    $eq$ : $eq = \mathbf{withControl}([t, t'], FO2DSystem(p^A(t), t), PointwiseControl([t, t'], Dir, t))$
    $H$ : $H = \{\hat{p} \mid d(\hat{p}, T) > \tau \land d(\hat{p}, \mathbf{0}) \leq A\}$
    $s$ : $s = (T, Dir)$
THEN
    act1 : $p^A :\sim_{t \to t'} \mathbf{withControl}([t, t'], FO2DSystem(p^A(t), t), PointwiseControl([t, t'], Dir, t))$
               $\&\{\hat{p} \mid d(\hat{p}, T) > \tau \land d(\hat{p}, \mathbf{0}) \leq A\}$
END
```

Listings 7.10: Abstract Robot – Machine's Actuation

Continuous Event. The system's actuation is presented in Listing 7.10. It mainly consists in instantiating the parameters of the generic model. The system is described using a controlled ODE defined in the theory, and we use the **withControl** operator to transform this controlled equation into a standard ODE with a fixed control command function, orienting the robot in the given direction *Dir* (function that also comes from the theory).

The evolution domain simply ensures the robot is not visiting a target (in which case it would need to change target and direction) and remains in the designated area.

Witnesses are provided, mirroring the guards, invariants and operated substitutions.

Proofs At this level, the model generates 110 POs. Excluding well-definedness (24%), most of those POs (50%) relate to refinement, and ensures the model is compatible with the generic model. These POs are discharge using the witnesses and the properties of the system.

The safety invariant is proven thanks to the evolution domain given for the continuous assignment. Note that witness feasibility (i.e. establish that there exist values for the witnesses) must be proven. The ODE is a simple one, making it straightforward establish witness feasibility.

7.5.1.3 Concrete Robot

The model developed in Section 7.5.1.2 allowed us to establish a number of fundamental properties in a fairly easy way, but it is quite far from the actual robot's implementation. For this reason, we propose a refinement of this model, closer to its real-world characteristics.

This refinement involves a more complex differential equation, and cannot be related in a direct, *exact* way to the abstract robot's differential equation. Therefore, our design relies on the approximation pattern, with approximate refinement to safely approximate the system's differential equation while retaining all of the properties we managed to establish at the abstract level.

Preliminary Study

Plant Description. The plant under control is the robot, characterised by its position $p^C \in \mathbb{R} \times \mathbb{R}$ and its speed $v^C \in \mathbb{R} \times \mathbb{R}$. Exactly like the abstract robot, it evolves in the area that is the closed ball of center $\mathbf{0}$ and radius $A > 0$, and must visit targets $T_{i,i \in I}$.

To control the robot, the controller issues a command $u^C \in \mathbb{R}^+ \to \mathbb{R} \times \mathbb{R}$ to give it a direction and a speed. Based on how the robot is built, the system follows the given controlled ODE:

$$\left\{ \begin{array}{lcl} \dot{v}^C & = & \frac{1}{2}u^C - K(p^C - w^C) - v^C \\ \dot{p}^C & = & v^C \\ \dot{w}^C & = & u^C \end{array} \right. \tag{7.4}$$

Where w^C is the derivative of the command and K is some correction coefficient. We note that the command u^C is bounded in norm by $\nu > 0$: $\|u^C\| = d(\mathbf{0}, u^C) \leq \nu$.

This system is equivalently written as a (controlled) ODE function:

$$f^C\left(t, \begin{bmatrix} v^C \\ p^C \end{bmatrix}, u^C\right) = \begin{pmatrix} \frac{1}{2}u^C - K(p^C - \int_{t_0}^{t} u^C) - v^C \\ v^C \end{pmatrix}$$

In [GJP08], the authors demonstrated that the system described by Equation 7.4 and the one described by Equation 7.3 are δ-bisimilar provided that:

1. the abstract command is bounded by μ: $\|u^A\| \leq \mu$
2. the concrete command is bounded by ν: $\|u^C\| \leq \nu$
3. the following relation holds:

$$\frac{\nu}{2}\left(1 + 4K + 2\sqrt{1+4K}\right) \leq \mu$$

In that case, $\delta = 2\nu$, and we have:

$$p^A \overset{\delta}{\approx} p^C$$

Controller Description. The controller does not change from the one of the abstract robot, with the exception that conditions require to be strengthened, following the approximation pattern (see Section 7.3.2). In the abstract system, a target is considered to be visited when $d(p^A(t), T_i) \leq \tau$; following the strengthening principle, the sensing becomes:

$$d(p^C(t), T_i) \leq \tau - \delta$$

Similarly, the area in which the robot evolves is strengthened as well:

$$\forall t, d(\mathbf{0}, p^C(t)) \leq A - \delta$$

Finally, the direction and speed stored by the controller are used to derive (u^C, w^C) (on a given interval $[t, t']$):

$$\forall t \in [t_0, t_1], \begin{cases} u^C(t) & = & \mathit{Dir} \\ w^C(t) & = & \int_{t_0}^{t} u^c(\tau)\mathrm{d}\tau = \mathit{Dir}(t - t_0) \end{cases}$$

This operation is performed by the operator *SlopedControl* defined in the robot theory (Section 7.5.1.1).

Requirements. The requirements of the concrete robot are the same as the abstract robot.

Event-B Development

```
CONTEXT Robot_1_Ctx EXTENDS Robot_0_Ctx
CONSTANTS δ , ν , K
AXIOMS
   axm1 : δ ∈ ℝ⁺
   axm2 : ν ∈ ℝ⁺
   axm3 : K ∈ ℝ⁺
    . . .
END
```

Listings 7.11: Abstract Robot – Context

Constants and Axioms. Listing 7.11 gives an extract of the defined context. This context extends the abstract one, and adds all the required constants (δ, ν, etc.). We recall at this point that a theory of robots is defined (Section 7.5.1.1), providing useful operators, and in particular a formalisation of the controlled ODE defined in Equation 7.4 (*SO2DSystem*) as well as an associated adequate control (*SlopedControl*).

Note that, to be able to prove certain properties later, we require that $\nu \leq \mu$ (which is not incompatible with the conditions of well-definedness of the approximation).

```
MACHINE Robot_1 REFINES Robot_0
SEES Robot_1_Ctx
VARIABLES t , p^C , v^C , T , Dir
INVARIANTS
    inv1−2 :  p^C ∈ ℝ ⇸ ℝ × ℝ ,
          v^C ∈ ℝ ⇸ ℝ × ℝ
    inv3−4 :  [0, t] ⊆ dom(p^C) ,
          [0, t] ⊆ dom(v^C)
    inv5 :  ‖Dir‖ ≤ ν
    inv6 :  p^A ≈^δ_[0,t] p^C
    inv7 :
          ∀t* · t* ∈ [0, t] ⇒ d(p^C(t*), 0) ≤ A − δ
```

```
INITIALISATION
WITH
    p^A′ :  p^A′ = p^C′
THEN
    act1 :  t := 0
    act2 :  T :| T′ ∈ ℝ × ℝ ∧ d(T, 0) ≤ A − τ
    act3 :  Dir = 0
    act4 :  p^C := {0 ↦ p_0}
    act5 :  v^C := {0 ↦ 0}
END
```

Listings 7.12: Concrete Robot – Machine's Header and Initialisation

Machine Header. Listing 7.12 present the machine's header and initialisation. The refinement replaces the abstract state variable p^A with the new continuous state variable, (p^C, v^C), defined and constrained in `inv1-4`. The other variables remain unchanged.

`inv5` constrains further variable *Dir* such that the resulting control is bounded in norm by ν (ensuring the required pre-condition for the approximation). `inv7` is a strengthened version of the safety invariant of the abstract machine (`inv7` in Listing 7.8), obtained by applying the δ-shrinking operator on the predicate.

`inv6` gives the approximate gluing invariant for the refinement. Compared to the approximation pattern, we note that $\mathcal{O} = \text{prj}_1$ (where prj_1 is the first canonical projector, i.e. the function that, given a pair, gives the first element of the pair). Note that, since prj_1 is a function, $\mathcal{O} \circ (p^C, v^C) = \text{prj}_1 \circ (p^C, v^C)$ is a function as well, and we can use $\approx^\delta$ instead of $\in^\delta$.

The initialisation is straightforward: variables are initialised in a similar way as for the abstract robot. The initial position is the same (p^0) and the initial speed of the robot is $\mathbf{0} = (0, 0)$, as to match $Dir = 0$.

Discrete Events. The refined events `sense_close_enough` and `transition_change_direction` are presented in Listing 7.13. Their refinements ensure that:

- invariant `inv5` is enforced (`grd3` in the sensing event and `grd1` in the transition); this is compatible with guard strengthening since $\nu \leq \mu$;

- guards are strengthened in accordance with the approximation pattern, using the shrinking operator:

$$\mathcal{S}_\delta(\{p \mid d(T, p) \leq \tau\}) = \{p \mid d(T, p) \leq \tau - \delta\}$$

  hence the expression of `grd1`;

```
EVENT sense__close__enough REFINES
      sense__close__enough
ANY $T_{next}$ , $Dir_{next}$
WHERE
    grd1 : $d(T, p^C(t)) \leq \tau - \delta$
    grd2 : $T_{next} \in \mathbb{R} \times \mathbb{R} \wedge d(\mathbf{0}, T_{next}) \leq A - \tau$
    grd3 : $Dir_{next} \in \mathbb{R} \times \mathbb{R} \wedge \|Dir_{next}\| \leq \nu$
THEN
    act1 : $T := T_{next}$
    act2 : $Dir := Dir_{next}$
END
```

```
EVENT transition__change__direction
      REFINES
      transition__change__direction
ANY $Dir_{new}$
WHERE
    grd1 : $Dir_{new} \in \mathbb{R} \times \mathbb{R} \wedge \|Dir_{new}\| \leq \nu$
THEN
    act1 : $Dir := Dir_{new}$
END
```

Listings 7.13: Concrete Robot – Machine's Discrete Events

```
EVENT actuate__movement REFINES actuate__movement
ANY $t'$
WHERE
    grd0 : $t' > t$
    grd1 : $\mathbf{Solvable}([t, t'], p^C \otimes v^C, \mathbf{withControl}([t, t'],$
               $\mathit{SO2DSystem}((p^C(t), v^C(t)), t), \mathit{SlopedControl}([t, t'], Dir, t)),$
               $\{\hat{p}, \hat{v} \mid d(\hat{p}, T) > \tau - \delta \wedge d(\hat{p}, \mathbf{0}) \leq A - \delta\})$
    grd2 : $d(p^C(t), T) > \tau - \delta$
WITH
    $p^{A\prime}$ : $p^{A\prime} \approx^{\delta}_{[0,t']} p^{C\prime}$
THEN
    act1 : $p^C, v^C :\sim_{t \to t'} \mathbf{withControl}([t, t'], \mathit{SO2DSystem}((p^C(t), v^C(t)), t), \mathit{SlopedControl}([t, t'], Dir, t))$
               $\& \{\hat{p}, \hat{v} \mid d(\hat{p}, T) > \tau - \delta \wedge d(\hat{p}, \mathbf{0}) \leq A - \delta\}$
END
```

Listings 7.14: Concrete Robot – Machine's Actuation

Continuous Events. Listing 7.14 gives the actuation for the concrete robot. The overall principle here is to substitute the abstract parameters (p^A, *FO2DSystem*, *PointwiseControl*) with the concrete ones ((p^C, v^C), *SO2DSystem*, *SlopedControl*). The substitution is possible thanks to the properties given in the robot theory. In particular, it allows enforcing correct approximation.

We note that the guards and the evolution domain are strengthened using predicate strengthening as proposed in Section 7.3.2. Last, a witness is provided for $p^{A\prime}$ that recalls the approximate gluing invariant.

Proofs The resulting model generated 91 proof obligations, about 19% of which relate to well-definedness and 40% to invariants. About 33% of the POs relate to refinement (guard strengthening, simulation, witness feasibility).

Invariant preservation proofs are straightforward since the invariants are simple. Invariant `inv7` is a bit more technical to maintain; it is proven using the various theorems and axioms available in the robot theory. The most difficult invariant preservation to prove from a purely technical point of view is the approximate continuous gluing invariant (`inv6`): it requires to enforce the approximation is well-defined, to instantiate theorems that establish the approximation of the two equation systems, and then to derive the approximation of their solution and hence of p^A and p^C using rewriting and substitution.

Guard strengthening and simulation are proven thanks to the work done in Section 7.3: it is a matter of establishing that the concrete predicates are included in the δ-shrinking of the abstract predicates and use related theorems to sort out simulation.

7.5.2 Inverted Pendulum Case Study

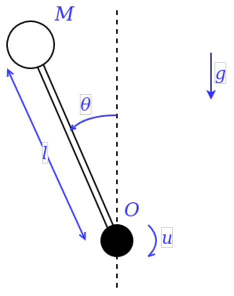


Figure 7.15: Inverted Pendulum Typical Scenario

A rod of length l is hooked to a step motor at point O and to a mass M at his other end. The motor is capable of delivering a *torque* of u. θ denotes the angle between the vertical axis and the rod.

The goal of the case study is to design a controller that issues commands to the motor (i.e. modifies u) in order to stabilise the rod around its centre position ($\theta = 0$).

The main problem of this case study is that realistic differential equations for θ are *non-linear*. However, for implementation purposes, and under certain conditions on θ, these differential equations may be *linearised*. A linear differential equation is obtained that approximates the non-linear one.

In the following, we formalise this linearisation step using approximate refinement.

7.5.2.1 Inverted Pendulum Theory

Inverted pendulums are associated with several physical and mathematical properties that are synthesised in a dedicated domain-specific theories of pendulum. An extract of said theory is given in Listing 7.15.

The inverted pendulum theory mainly defines controlled differential equations for the non-linear and linearised ($Pendulum_{NonLin}$ and $Pendulum_{Lin}$ resp.) pendulum plants, as well as possible control command functions for controlling these pendulums ($PendulumControl_{NonLin/Lin}$). It also formalises the linearisation step as an axiomatic well-definedness predicate ($PendulumApproxWD$) together with an axiom (`pend_approx`) that allows establishing an approximation relationship between the two controlled differential equations $Pendulum_{NonLin}$ and $Pendulum_{Lin}$.

The idea is to gather and abstract the mathematical background needed to handle approximation of inverted pendulum systems in an abstract operator that characterises the “good properties” of inverted pendulums, required for establishing approximation. The associated axiom allows establishing this approximation formally, using the well-definedness operator as hypothesis.

```
THEORY InvertedPendulum IMPORT Approximation
OPERATORS
  PendulumFun_NonLin expression (ω0 : ℝ)
    direct definition
      (λt ↦ (x1 ↦ x2) ↦ u · t ∈ ℝ+ ∧ (x1 ↦ x2) ∈ ℝ² ∧ u ∈ ℝ| x2 ↦ (u cos(x1) + ω0² sin(x1)) )
  Pendulum_NonLin expression (ω0 : ℝ , x0 : ℝ² , t0 : ℝ)
    direct definition code(PendulumFun_NonLin(ω0), x0, t0)
  PendulumFun_Lin expression (ω0 : ℝ)
    direct definition
      (λt ↦ (x1 ↦ x2) ↦ u · t ∈ ℝ+ ∧ (x1 ↦ x2) ∈ ℝ² ∧ u ∈ ℝ| x2 ↦ (u + ω0² x1) )
  Pendulum_Lin expression (ω0 : ℝ , x0 : ℝ² , t0 : ℝ)
    direct definition code(PendulumFun_Lin(ω0), x0, t0)
AXIOMATIC DEFINITIONS
  OPERATORS
    theta_max expression (ω0 : ℝ) : ℝ
    PendulumControl_NonLin expression (ω0 : ℝ , (θ0, θ̇0) : ℝ² , t0 : ℝ+) : ℝ ⇸ ℝ
    PendulumControl_Lin expression (ω0 : ℝ , (θ0, θ̇0) : ℝ² , t0 : ℝ+) : ℝ ⇸ ℝ
    ...
    PendulumApproxWD predicate (δ, ω0, θ_bound, M_NonLin, M_Lin, δ_C : ℝ+* , t0, t1 : ℝ+)
      well−definedness condition t0 < t1
  AXIOMS
    pend_approx : ∀t0, t1, δ, ω0, ... · t0 ∈ ℝ+ ∧ t1 ∈ ℝ+δ ∈ ℝ+ ∧ ω0 ∈ ℝ ∧ ... ∧
      PendulumApproxWD(δ, ω0, θ_bound, M_NonLin, M_Lin, δ_C, t0, t1)
        ⇒ withControl([t0, t1], Pendulum_Lin(ω0, θ⁰_Lin, t0), u_Lin)
          ≈^δ_[t0,t1] withControl([t0, t1], Pendulum_NonLin(ω0, θ⁰_NonLin, t0), u_NonLin)
```

Listings 7.15: Inverted Pendulum Theory

7.5.2.2 Non-Linear Inverted Pendulum

We first model the inverted pendulum accurately using physics and mathematics. The differential equation modelling the pendulum's behaviour is *non-linear*. This first model is used to establish safety properties (such as the fact the pendulum does not fall).

Note that this model is a refinement of the generic model (as presented in Chapter 5).

Preliminary Study

Plant Description. The controlled plant is the inverted pendulum, modelled by the angle between the rod and the vertical axis, denoted θ. This angle may be adjusted with the help of a step motor that serves as actuator for the system. This actuator is able to issue a control command function, denoted u, bounded in absolute value by the constant M_{NonLin}. Physics and trigonometry allows establishing the following differential equation for θ:

$$\ddot{\theta} - \frac{g}{l}\sin(\theta) = u\cos(\theta)$$

This differential equation may equivalently be expressed under the form of an ODE:

$$\begin{bmatrix}\dot{\theta}\\ \ddot{\theta}\end{bmatrix} = f_{NonLin}\left(\begin{bmatrix}\theta\\ \dot{\theta}\end{bmatrix}, u\right), \text{ with } f_{NonLin}\left(\begin{bmatrix}x_1\\ x_2\end{bmatrix}, u\right) = \begin{bmatrix}x_2\\ u\cos(x_1) + \frac{g}{l}x_1\end{bmatrix}$$

The factor $\frac{g}{l}$ is constant and usually denoted ω_0^2. It is the angular frequency or *pulsatance* of the system, and it is linked to the period of the pendulum's oscillations.

This ODE is non-linear because of the terms $\sin(\theta)$ and $\cos(\theta)$. In particular, it does not admit an analytical (explicit) solution, and reachability is undecidable.

The pendulum is controllable (i.e. can be balanced around $\theta = 0$) provided θ remains bounded by a constant, denoted θ_{max}, that depends on ω_0 and M_{NonLin}.

Controller Description. The role of the controller is to compute an adequate control command function to issue to the actuator. This is done in two steps:

1. The controller senses the value of θ and $\dot{\theta}$ and stores it in variables θ^{sense} and $\dot{\theta}^{sense}$ with the associated time point of the sensing, t^{sense} (`sense_angle`);

2. The controller uses the sensed values (θ^{sense} and $\dot{\theta}^{sense}$) and time (t^{sense}) to compute u (`calculate_control`);

This behaviour is summarised in Figure 7.17 as an automaton. Note that the controller has only one mode, named *control*.

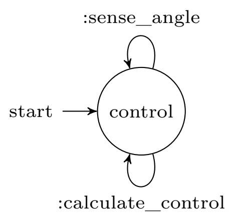


Figure 7.17: System Mode Automaton

Requirements. The system's requirements can be summarised as so:
FUN1 The controller senses the angle (θ) of the pendulum (`:sense_angle`)
FUN2 If the value of the sensed angle is not 0, the controller sends a command to stabilise the pendulum at $\theta = 0$ (`:calculate_control`)
SAF1 For $|\theta| < \theta_{max}$, the system is always controllable
ENV1 The system is subject to perturbations that may cause its angle to vary

Event-B Development

Constants and Axioms. We first define a context for the model that defines any needed constants and parameters. This context is given in Listing 7.16. It defines ω_0 and θ_{max}, as well as the system's initial conditions, θ^0. It also specifies the set of modes for the system (STATES), consisting of one element (control).

Machine Header. Listing 7.17 presents the machine's header and initialisation. The machine refines the generic model, substituting the abstract continuous state x_p with $[\theta\ \dot{\theta}]^\top$ using gluing invariant `inv5` and the abstract discrete state x_s with the unique mode of the controller (control) using gluing invariant `inv7`. In addition to these variables, the machine defines t^{sense}, θ^{sense} and $\dot{\theta}^{sense}$ to store

```
CONTEXT PendulumCtx
CONSTANTS $\omega_0$ , $\theta_{max}$ , $\theta^0$ , control
AXIOMS
    axm1 : $\omega_0 \in \mathbb{R}$
    axm2 : $\theta_{max} = \dots$
    axm3 : $\theta^0 \in \mathbb{R}$
    axm4 : $|\theta^0| < \theta_{max}$
    axm5 : $\text{partition}(\text{STATES}, \{\text{control}\})$
END
```

Listings 7.16: (Non-Linear) Pendulum Context

```
MACHINE Pendulum REFINES Generic
SEES PendulumCtx
VARIABLES
    $t$ , $\theta$ , $\dot\theta$ , $t^{sense}$ , $\theta^{sense}$ , $\dot\theta^{sense}$ , $control\_fun$
INVARIANTS
    inv1-4 : $\theta \in \mathbb{R} \nrightarrow \mathbb{R}$ , $\dot\theta \in \mathbb{R} \nrightarrow \mathbb{R}$ , $[0,t] \subseteq \text{dom}(\theta)$
           , $[0,t] \subseteq \text{dom}(\dot\theta)$
    inv5 : $x_p = [\theta\,\dot\theta]^\top$
    inv6 : $\forall t^* \cdot t^* \in [0,t] \Rightarrow |\theta(t^*)| < \theta_{max}$
    inv7 : $x_s = \text{control}$
    inv8-10 : $t^{sense} \in \mathbb{R}^+$ , $\theta^{sense} \in \mathbb{R}$ , $\dot\theta^{sense} \in \mathbb{R}$
    inv11-12 : $u \in \mathbb{R} \nrightarrow \mathbb{R}$ , $[t^{sense}, +\infty[ \subseteq \text{dom}(u)$
    inv13 : $|\theta^{sense}| \leq \theta_{max}$
```

```
INITIALISATION
WITH
    $x_p'$ : $x_p' = \{0 \mapsto (\theta_0 \mapsto 0)\}$
    $x_s'$ : $x_s' = control$
THEN
    act1 : $t := 0$
    act2 : $\theta := \{0 \mapsto \theta_0\}$
    act3 : $\dot\theta := \{0 \mapsto 0\}$
    act4 : $t^{sense} := 0$
    act5 : $\theta^{sense}, \dot\theta^{sense} := \theta_0, 0$
    act6 : $u :=$
      $PendulumControl_{NonLin}(\omega_0, (\theta_0 \mapsto 0), 0)$
END
```

Listings 7.17: Machine Header and Initialisation

the values sensed by the controller, as well as u, the control command function computed by the controller and used during actuation.

Invariants `inv1-4` and `inv8-10` precise the basic type and properties of each variables. Finally, invariants `inv6` and `inv13` model the system's safety, i.e. the condition under which the system is controllable.

Initialisation is written following the generic model. Each variables of the model is given an initial value; in particular $\theta(0) = \theta^0$ and $\dot\theta(0) = 0$. Witnesses are given for the substituted variables (x_s and x_p) in order to enforce the correctness of the refinement.

Discrete Events. Listing 7.18 shows the sensing and transition events of the system, following the system's mode automaton (Figure 7.17). When $\theta(t)$ does not equal to 0, `sense_angle` is triggered. It senses the value of $\theta(t)$ and $\dot\theta(t)$, and stores these two values, in addition to the current time t, in dedicated variables (θ^{sense}, $\dot\theta^{sense}$ and t^{sense} resp.).

Event `transition_calculate_control` uses the sensed values to compute an adequate control command function (u) to be issued to the pendulum's motor during actuation. The function's computation is done using pendulum's theory operators (see Section 7.5.2.1).

Continuous Event. The system's actuation is given in Listing 7.19. It revolves around the use of the $:\sim_{t \to t'}$ operator to update the behaviour of the continuous state variables (θ, $\dot\theta$) using the controlled ordinary differential equation $Pendulum_{NonLin}$ (defined in the inverted pendulum theory,

```
**sense_angle** REFINES *Sense*
WHERE
    grd1 : $|\theta(t)| > 0$
WITH
    $x'_s$ : $x'_s = control$
    $s$ : $s = \{\text{control}\}$
    $p$ : $p = \{\text{control}\} \times \mathbb{R} \times$
        $\{\theta^*, \dot{\theta}^* \mid |\theta^*| \geq \theta_{max}\}$
THEN
    act1 : $t^{sense}, \theta^{sense}, \dot{\theta}^{sense} := t, \theta(t), \dot{\theta}(t)$
END
```

```
**transition_calculate_control**
  REFINES *Transition*
WITH
    $x'_s$ : $x'_s = control$
    $s$ : $s = \{\text{control}\}$
THEN
    act1 : $u :=$
  $PendulumControl_{NonLin}(\omega_0, \theta^{sense}, \dot{\theta}^{sense}, t^{sense})$
END
```

Listings 7.18: Sensing and Transition Events

```
**actuate_balance** REFINES *Actuate*
ANY   $t'$
WHERE
    grd1 : $t' \in \mathbb{R}^+ \wedge t < t'$
    grd2 : $\textbf{SolvableWith}([t, t'], Pendulum_{NonLin}(\omega_0, (\theta(t) \mapsto \dot{\theta}(t)), t), u)$
    grd3 : $\theta(t) < \theta_{max}$
WITH
    $e$ : $e = \textbf{withControl}([t, t'], Pendulum_{NonLin}(\omega_0, (\theta(t) \mapsto \dot{\theta}(t)), t), u)$
    $H$ : $H = \{\theta^*, \dot{\theta}^* \mid |\theta^*| < \theta_{max}|\}$
    $x'_p$ : $x'_p = \begin{bmatrix} \theta' \; \dot{\theta}' \end{bmatrix}^\top$
    $s$ : $s = \{\text{control}\}$
THEN
    act1 : $t, \theta, \dot{\theta} :\sim_{t \to t'}$
               $\textbf{withControl}([t, t'], Pendulum_{NonLin}(\omega_0, (\theta(t) \mapsto \dot{\theta}(t)), t), u)$
               $\& \{\theta^*, \dot{\theta}^* \mid |\theta^*| < \theta_{max}\}$
END
```

Listings 7.19: System Actuation

see Section 7.5.2.1) and the computed control command function, u.

Witnesses are provided for the generic model's parameters to reflect the model's constraints.

Proofs A number of 100 proof obligations is generated for this first refinement. 34% of these POs relate to refinement, that is to say to the instantiation of the generic model. Well-definedness makes for 29% of these POs, and is mostly due to the use of complex operators, defined in theories. Finally, the other POs (37%) are for invariant preservation. These are mostly related to typing and basic properties of the variables, and also include the proof of the system's safety properties, carried out using properties written in the pendulum theory.

7.5.2.3 Linearised Inverted Pendulum

The non-linear version of the inverted pendulum is derived directly from physics, but cannot be implemented in its current form due to its non-linear differential equation. We propose a linearised version of the inverted pendulum, expressed as an approximate refinement of the non-linear one.

The correctness of the approximation is abstracted in the inverted pendulum domain theory (see Section 7.5.2.1).

Preliminary Study

Plant Description. The controlled plant is the same as for the non-linear inverted pendulum. We use θ_{Lin} to denote the continuous state of the linearised inverted pendulum.

The idea is that, when θ is *small enough*, $\sin(\theta)$ may be approximated with θ_{Lin}, and $\cos(\theta)$ with 1. The control command function used to control the system is modified, and denoted u_{Lin}. It is bounded by the constant M_{Lin}. The system is associated with the following differential equation for θ_{Lin}:

$$\ddot{\theta}_{Lin} - \omega_0^2 \theta_{Lin} = u_{Lin}$$

This differential equation may equivalently be expressed under the form of an ODE:

$$\begin{bmatrix} \dot{\theta}_{Lin} \\ \ddot{\theta}_{Lin} \end{bmatrix} = f_{Lin}\left(\begin{bmatrix} \theta_{Lin} \\ \dot{\theta}_{Lin} \end{bmatrix}, u \right), \text{ with } f_{Lin}\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, u \right) = \begin{bmatrix} x_2 \\ u_{Lin} + \omega_0^2 x_1 \end{bmatrix}$$

Note that this differential equation is linear.

This new differential equation approximated the non-linear one, provided controls are approximated ($u \approx^{\delta_C} u_{Lin}$, δ_C constant) and θ is bounded ($|\theta| \leq \theta_{bound}$, with θ_{bound} constant). This bound depends on the required approximation bound of the system δ, the control functions bounds (M_{NonLin} and M_{Lin}) and approximation bound (δ_C).

Controller Description. The controller remains unchanged compared to the non-linear inverted pendulum. The only difference is that guards and local invariants shall be *strengthened* in order to establish refinement correctness.

Requirements. The requirements are the same as for the non-linear inverted pendulum, substituting θ by θ_{Lin}. To ensure approximation is possible, the system is required to observe an additional property:

FUN3 The value of θ must remain bounded by θ_{bound};

It is to be noted that this requirement relate to the non-linear version of θ. Using predicate strengthening, it can be transformed into an equivalent requirement on θ_{Lin}:

FUN3' The value of θ_{Lin} must remain bounded by $\theta_{bound} - \delta$;

Event-B Development

Constants and Axioms. A context is defined for this development, given in Listing 7.20. This context defines the approximation bounds δ and δ_C, as well as θ_{bound}.

Machine Header. Listing 7.21 shows the machine's header and initialisation. The abstract continuous state variables (θ and $\dot{\theta}$) are substituted with θ_{Lin} and $\dot{\theta}_{Lin}$ using the approximate continuous gluing invariant `inv5`. Similarly, sensed variables θ^{sense} and $\dot{\theta}^{sense}$ are substituted with θ_{Lin}^{sense} and θ_{Lin}^{sense} respectively, using the approximate equality operator ($\approx$).

Invariants `inv6`, `inv8` and `inv11` give additional constraints on the system's variables in order to enforce approximation well-definedness (in particular, it ensures δ_C-approximation of the control command function u and u_{Lin}).

```
CONTEXT PendulumLinCtx EXTENDS PendulumCtx
CONSTANTS $\delta$ , $\delta_C$ , $\theta_{bound}$
AXIOMS
   axm1 : $\delta \in \mathbb{R}$
   axm2 : $\delta > 0$
   axm3 : $\delta_C \in \mathbb{R}$
    . . .
END
```

Listings 7.20: (Non-Linear) Pendulum Context

```
MACHINE PendulumLin REFINES Pendulum
SEES PendulumLinCtx
VARIABLES  $t$ , $t^{sense}$ , $u_{Lin}$ ,
      $\theta_{Lin}$ , $\dot{\theta}_{Lin}$ , $\theta^{sense}_{Lin}$ , $\dot{\theta}^{sense}_{Lin}$
INVARIANTS
   inv1−4 : $\theta_{Lin} \in \mathbb{R} \nrightarrow \mathbb{R}$ , $\dot{\theta}_{Lin} \in \mathbb{R} \nrightarrow \mathbb{R}$ ,
         $[0, t] \subseteq \mathrm{dom}(\theta_{Lin})$ , $[0, t] \subseteq \mathrm{dom}(\dot{\theta}_{Lin})$
   inv5 : $[\theta_{Lin}\ \dot{\theta}_{Lin}]^\top \approx^{\delta}_{[0,t]} [\theta\ \dot{\theta}]^\top$
   inv6 : $\forall t^* \cdot t^* \in [0, t] \Rightarrow$
      $|\theta(t^*)| < \theta_{bound} \wedge |\theta_{Lin}(t^*)| < \theta_{bound} - \delta$
   inv7 : $u_{Lin} \in \mathbb{R} \nrightarrow \mathbb{R}$
   inv8 : $u_{Lin} \approx^{\delta_{ctrl}}_{[t^{sense},+\infty[} control\_fun$
   inv9−10 : $\theta^{sense}_{Lin} \in \mathbb{R}$ , $\dot{\theta}^{sense}_{Lin} \in \mathbb{R}$
   inv11 : $|\theta^{sense}_{Lin}| \leq \theta_{bound} - \delta$
   inv12 : $[\theta^{sense}\ \dot{\theta}^{sense}]^\top \approx^{\delta} [\theta^{sense}_{Lin}\ \dot{\theta}^{sense}_{Lin}]^\top$
```

```
INITIALISATION
WITH
   $\theta'$ ; $\theta' = \theta_0$ , $\dot{\theta}'$ : $\dot{\theta}' = 0$
   $\theta^{sense\prime}$ : $\theta^{sense\prime} = \theta_0$ , $\dot{\theta}^{sense\prime}$ : $\dot{\theta}^{sense\prime} = 0$
   $control\_fun'$ : $control\_fun' =$
  $PendulumControl_{NonLin}(\omega_0, (\theta_0 \mapsto 0), 0)$
THEN
   act1 : $t := 0$
   act2 : $t^{sense} := 0$
   act3 : $\theta_{Lin} := \{0 \mapsto \theta_0\}$
   act4 : $\dot{\theta}_{Lin} := \{0 \mapsto 0\}$
   act5 : $u_{Lin} :=$
      $PendulumControl_{Lin}(\omega_0, (\theta_0 \mapsto 0), 0)$
   act6 : $\theta^{sense}_{Lin}$ , $\dot{\theta}^{sense}_{Lin} := \theta_0, 0$
END
```

Listings 7.21: Machine Header and Initialisation

Finally, invariants `inv1-4`, `inv7` and `inv9-10` give the basic properties of the machine's variables.

The machine's initialisation is derived from the abstract machine's initialisation by substituting the abstract variables with the concrete ones using their respective gluing invariants. The concrete system is assumed to start from the exact same position as the abstract one (i.e. $\theta_{Lin}(0) = \theta(0) = \theta^0$ and $\dot{\theta}_{Lin}(0) = \dot{\theta}(0) = 0$).

```
sense_angle REFINES sense_angle
WHERE
   grd1 : $|\theta_{Lin}(t)| > 0$
WITH
   $\theta^{sense\prime}$ , $\dot{\theta}^{sense\prime}$ :
      $[\theta^{sense\prime}\ \dot{\theta}^{sense\prime}]^\top \approx^{\delta} [\theta^{sense\prime}_{Lin}\ \dot{\theta}^{sense\prime}_{Lin}]^\top$
THEN
   act1 : $t^{sense}, \theta^{sense}_{Lin}, \dot{\theta}^{sense}_{Lin} :=$
      $t, \theta_{Lin}(t), \dot{\theta}_{Lin}(t)$
END
```

```
transition_calculate_control REFINES
      transition_calculate_control
WITH
   $u'$ : $u' \approx^{\delta}_{[t^{sense},+\infty[} u'_{Lin}$
THEN
   act1 : $u_{Lin} :=$
   $PendulumControl_{Lin}(\omega_0, \theta^{sense}_{Lin}, \dot{\theta}^{sense}_{Lin}, t^{sense})$
END
```

Listings 7.22: Sensing and Transition Events

Discrete Events. Listing 7.22 presents the refining sensing and transition events for the system. They follow the abstract machine's sensing and transition events, substituting the abstract variables with the concrete ones using the gluing invariants as witnesses. The control command function for the system is provided by the inverted pendulum theory ($PendulumControl_{Lin}$). This function is expected to be a δ_C-approximation of $PendulumControl_{NonLin}$, used in the abstract model, in order for the approximation to be correct.

```
**actuate_balance** REFINES *actuate_balance*
ANY  $t'$
WHERE
    grd1 :  $t' \in \mathbb{R}^+ \wedge t < t'$
    grd2 :  $\mathbf{SolvableWith}([t,t'], Pendulum_{Lin}(\omega_0, (\theta_{Lin}(t) \mapsto \dot{\theta}_{Lin}(t)), t), u_{Lin})$
    grd3 :  $|\theta_{Lin}(t)| < \theta_{bound} - \delta$
    grd4 :  $|\theta_{Lin}(t)| < \theta_{max} - \delta$
WITH
    $\theta, \dot{\theta}$ :  $[\theta' \; \dot{\theta}'] \approx^{\delta}_{[0,t']} [\theta'_{Lin} \; \dot{\theta}'_{Lin}]$
THEN
    act1 :  $t, \theta_{Lin}, \dot{\theta}_{Lin} :\sim_{t \to t'}$
        $\mathbf{withControl}([t,t'], Pendulum_{Lin}(\omega_0, (\theta_{Lin}(t) \mapsto \dot{\theta}_{Lin}(t)), t), u_{Lin})$
        $\&\{\theta^*, \dot{\theta}^* \mid |theta^*| < \theta_{max} \wedge |\theta^*| < \theta_{bound} - \delta\}$
END
```

Listings 7.23: System Actuation

Continuous Event. The system's actuation is given in Listing 7.19. It is built from the abstract system's actuation by substituting the abstract continuous variables, θ and $\dot{\theta}$, with the concrete ones, θ_{Lin} and $\dot{\theta}_{Lin}$, using the approximate gluing invariant as a witness.

Proofs The model generated 63 proof obligations. Well-definedness makes for 37% of them, and again is due to our extensive use of operators defined in the inverted pendulum and approximation theories. Refinement-related POs (19%) consist of simulation and guard strengthening. The model is designed following the principle of predicate strengthening as presented in Section 7.3.2, which helps in carrying out the proof of such POs. The properties associated to both systems, defined in the inverted pendulum domain theory, are also fundamental for discharging the POs.

The other POs relate to invariant preservation (44%). Most invariants serve to type and constraint variables. The proof of the safety invariant also benefits from the use of predicate strengthening.

Finally, preservation of the approximate gluing invariant is the least trivial PO to discharge. It heavily relies on the particular properties of the system, synthesised in the inverted pendulum theory.

7.6 Discussion

Approximation is a fundamental technique used by controller designers. It takes many forms (e.g. linearisation, numeric integration, etc.). We formalised this operation in the context of our framework, as a generic refinement that can be instantiated for any hybrid system. The general schema of hybrid systems in our framework and of approximate refinement itself allows its use for two purposes:

- start with an approximated abstraction of the system for which proofs are eased, and refine it to an exact, more complex concrete system that preserves the property of the abstract system;
- start with a complex system, closer to the actual physical phenomena involved in the system, and refine it to an approximated, simpler system that can be implemented;

The work presented in this chapter has been published in [Dup+20b; Dup+20a]. where approximation has been successfully used to address the two case studies presented in this chapter. Note that the first use case for approximation corresponds to the planar robot case study (Section 7.5.1), while the second use case corresponds to the inverted pendulum case study (Section 7.5.2).

The control of a planar robot [Dup+20b], borrowed from the work of Fainekos [FGP07] and studied by Girard and Pappas [GJP08], has been addressed in the context of our framework and using approximate refinement. The idea is to first model a simpler system, on which proving properties is eased, and then to safely approximately refine it with a more complex system, closer to reality and to the controller's implementation. The correctness of the simplified system and of the approximate refinement operation entail the correctness of the complex system, without further proofs.

Approximate refinement is also used in [Dup+20a] in order to tackle the inverted pendulum case study. The inverted pendulum is associated with a non-linear differential equation that is linearised so it can be implemented. The equation's linearisation is formalised as an approximate refinement step to ensure the resulting model retains its properties (and especially safety).

The complete models for both cases studies are available in Appendix B. The robot case study is given in Section B.5, and the inverted pendulum case study is given in Section B.6.

Following our general approach, approximation is made generic. It may be used on any system that is based on the generic model, but also with any particular kind of approximation technique (i.e. linearisation, polynomial approximation, etc.) without having to alter it.

Approximation correctness highly depends on the system's dynamics, and is generally hard to establish. The use of domain theories to gather domain-specific knowledge is particularly relevant in this context. In practice, specific results on approximation well-definedness is established *outside* Event-B, and is used directly during development.

Conclusion

Hybrid systems gained an important place in our surrounding, including in safety-critical applications, from autonomous vehicles to computer-assisted tools. Addressing the formal design and verification of such systems is a crucial issue, with numerous challenges.

Due to their hybrid nature, standard "discrete" formal methods techniques do not to handle hybrid systems properly. Conversely, control theory alone cannot be used to formalise and study them. The major challenge in this context is to offer the capability to model and prove properties of such systems *as a whole*. In particular, handling, at the same level, both their discrete and continuous behaviours.

To address this challenge, several formal methods have been proposed. We can split these approaches in two categories:

- *ad hoc* formal methods that propose completely new languages, semantics and verification systems, specialised for hybrid systems

- *extension* of existing formal methods with new features, exploiting their underlying verification system (and tooling support)

Our work follows this latter category. Our contribution extends the Event-B formal method with specific features for handling hybrid systems, both in modelling phases and during the proof process. We exploit Event-B's semantics and proof system to address correctness of the developed hybrid systems. Refinement is used as the basis of generic development operations.

The use of Event-B also makes our work *tool-supported*: the various patterns presented throughout this manuscript have been encoded in Rodin, using the theory plug-in.

Contributions. We provide a generic, reusable and extensible framework, based on a well-tried method that supports the design of controller-plant loop hybrid systems, in both a formal and incremental way. This formal framework, relying on a generic hybrid system model and on the refinement operation, proposes several important formally verified patterns, inspired by common design techniques, handling the comprehensive design of hybrid systems. It is the support of a general *methodology* for hybrid system design.

Event-B Extension We extended Event-B using theories to incorporate continuous features in models, to model the continuous behaviours of hybrid systems at the same level as the discrete behaviours, and to handle them during the verification process.

The general theories defined in the context of this work are available in Appendix A.

Generic Model The foundation of the framework is a generic model that abstracts controller-plant loop hybrid systems. It relies on generic variables and parameters that are instantiated by providing witnesses to derive any specific hybrid system, making it reusable.

This model is the entry point of specific hybrid system developments. It is proven once and for all and, when instanciating it, instanciation proofs are carried out.

Generic Patterns The framework provides several generic *patterns*, formalised as refinements of the generic model. These patterns ease the development and verification process of hybrid systems. They can be applied on any model refining the generic model. The patterns defined in this work consists of:

- architectural patterns, for decomposing a hybrid system into multiple components:
  - one controller with one plant (single-to-single)
  - one controller with several plants (single-to-many)
  - several controllers with several plants (many-to-many)
- behavioural patterns, for handling the hybrid system's behaviour at a high level:
  - approximation, for substituting a continuous behaviour with another one close enough but not exactly the same, while retaining its properties (safety)

Having the generic model as a basis for these patterns makes the framework extensible: adding a new pattern to the framework consists of defining a refinement of the generic model, possibly with the addition of new general theories used by the pattern.

Refinement and Instantiation Patterns are applied using the *instantiation* operation: the model is refined following the pattern, and the generic features (variables and parameters) of the pattern are provided using well-defined witnesses. Pattern application may be performed at any point during development and in any order, making these patterns composable.

Methodology The framework is the support of a general methodology for hybrid system design. It is built around:

- *Domain Theories*: a specific hybrid system is accompanied by *domain specific knowledge*, gathered and formalised in a domain theory.
- *Refinement Strategy*: the development process is designed as a sequence of pattern applications, with the generic model as entry point.
  - Architecture patterns are applied first to decompose the system in simpler components while preserving high-level properties;
  - Behavioural patterns are used to handle each component's behaviour;

The theories and patterns defined in the framework as well as its associated methodology are summarised in Figure 7.25.

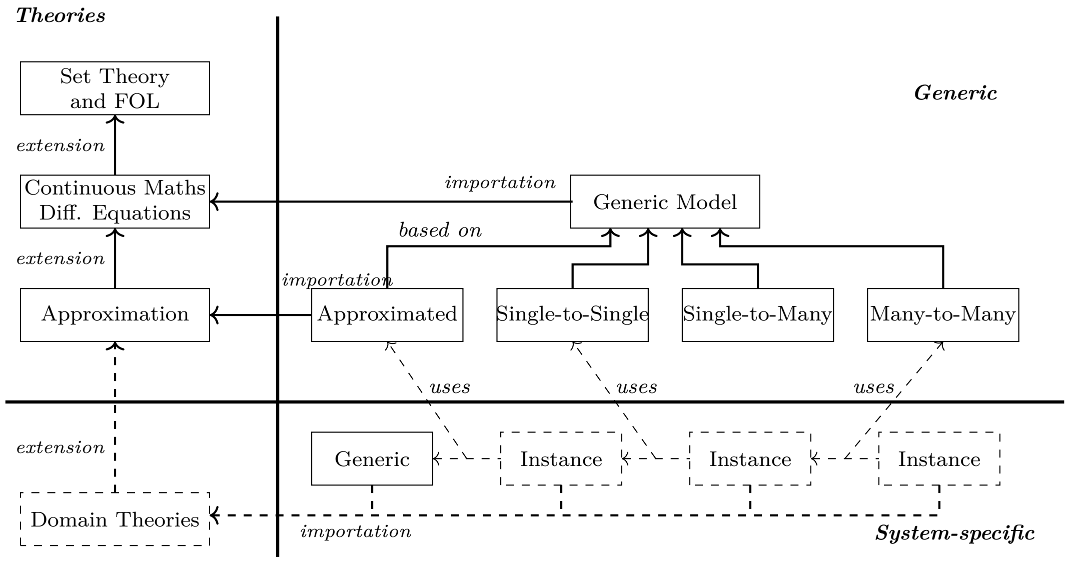


Figure 7.25: Framework Components

Case Studies The use of the framework is illustrated with the development of multiple case studies:

- control of a car for automatic braking and signalised left-turn
- control of a volume of liquid in one or several tank(s) with multiple different tanks configuration
- control of a robot evolving in a designated area and visiting a set of given targets
- control of an inverted pendulum, i.e. balancing a mass on a rod

Several domain theories have been designed to support these case studies development with domain specific knowledge: a theory for liquid tanks, a theory for planar robots and a theory for inverted pendulum.

These cases studies have been developed in Event-B using Rodin. Complete models for them can be found in Appendix B.

Perspectives and Future Work. The proposed framework is the entry point for designing hybrid systems, but it can be extended and improved furthermore in a number of directions.

Extension of the Framework Additional patterns may be defined in the framework, to handle other development operations, common in the design of hybrid systems. In particular, *discretisation* operations would allow formally transforming a continuous model of a hybrid system into a discrete one, closer to implementation, while ensuring the preservation of its properties. This operation could be further extended to handle *floating-point numbers*, as an approximated implementation of real numbers on real-world machines.

Finally, other types of approximations may be handled, for instance *proportional-integration-derivation* (PID) that addresses specific types of hybrid systems and are widely used in controller design.

Additional Theories To be able to handle a larger panel of hybrid systems, other theories should be defined. For instance, in our work, we only handle ordinary differential equations (ODEs), which is the most common type of continuous behaviour models in hybrid systems; but other types of differential equations, together with their relevant properties, may be formalised in theories (e.g. partial differential equations).

Moreover, general domain theories for physics may be defined, e.g. for kinematics, thermodynamics, fluid mechanics, etc. Then, these theories may be used to design more specific domain theories; for instance, theories for self-driving cars, trains, planes, and so on may be proposed to address several case studies in these respective domains.

Proving Process Due to the limitations of Rodin and the theory plug-in and its poor integration with Event-B's provers, proof is often cumbersome. Its ergonomics needs to be improved. Note that this is slightly mitigated by the possibility to define proving rules in theories, although these rules are not applied automatically.

The handling of continuous behaviours in proofs can be improved by the use of external dedicated tools such as analytical and numerical differential equation solvers (Mathematica, MatLab, etc.) and specialised SMT for ODEs (e.g. iSAT-ODE [Egg+11]) to establish useful properties on the handled behaviours, in particular for reachability properties (i.e. properties of the form $x(t) \in H$). Specialised SMT solvers for real numbers such as dReal [GKC13] may also be used to deal with real arithmetic, common in hybrid system models (particularly in axioms, guards and invariants).

Model-Checking and Animation Classical Event-B models are model-checked and animated using the ProB tool [LB03] that supports symbolic execution, invariant checking and temporal logic properties checking. ProB could be extended to handle hybrid features such as continuous dynamics, and used to model-check/animate hybrid systems, although that would considerably impact its core functions.

Hybrid model checkers may also be used, but they require to transform hybrid system Event-B models into hybrid automata.

Another lead on this topic is to interface ProB with dedicated tools for numerical differential equation integration (e.g. Simulink): ProB would handle the discrete controller part of hybrid systems and would interact with numerical solvers that handle continuous plant behaviours. This co-simulation technique may be used for animation and reachability analysis.

Theories Consistency The Event-B theories written in the context of this work heavily rely on numerous axioms that are formalisation of mathematical properties. Although these properties are perfectly sound, the transcription process may be flawed, resulting in imprecise or even inconsistent axioms.

Event-B theories do not provide a way to check consistency of the theories. Ideally, the theory may be improved by reducing the number of axioms they rely on, and prove everything else. However, finding the minimal set of axioms at the foundation of a theory is a difficult problem. This type of problem has already been studied in other contexts (e.g. with Dedukti [Sai13]).

Autonomous Systems A broader perspective for this framework is to address the problem of fully autonomous composite systems (e.g. a drone fleet or a car platoon), which are advanced cases of distributed system with no global state, but a set of properties, local to a restrained number of components. Establishing global invariants on such system is particularly challenging; it usually requires advanced models of the environment and communicating network.

Bibliography

[Abr+09] Jean-Raymond Abrial et al. *Proposals for Mathematical Extensions for Event-B*. Tech. rep. 2009.

[Abr+10] Jean-Raymond Abrial et al. “Rodin: an open toolset for modelling and reasoning in Event-B”. English. In: *International Journal STTT* 12.6 (2010), pp. 447–466. ISSN: 1433-2779.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 9781139637794.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996. ISBN: 978-0-521-02175-3.

[ADM02] Eugene Asarin, Thao Dang, and Oded Maler. “The d/dt Tool for Verification of Hybrid Systems”. In: *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 365–370. ISBN: 978-3-540-45657-5.

[AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. “Automatic symbolic verification of embedded systems”. In: *IEEE Transactions on Software Engineering* 22.3 (1996), pp. 181–201.

[Alu+95] Rajeev Alur et al. “The algorithmic analysis of hybrid systems”. In: *Theoretical Computer Science* 138.1 (1995). Hybrid Systems, pp. 3–34. ISSN: 0304-3975.

[Aré+12] Nikos Aréchiga et al. “Using theorem provers to guarantee closed-loop system properties”. In: *2012 American Control Conference (ACC)*. June 2012, pp. 3573–3580. DOI: `10.1109/ACC.2012.6315388`.

[BAB16] Michael Butler, Jean-Raymond Abrial, and Richard Banach. “From Action Systems to Distributed Systems: The Refinement Approach”. In: ed. by Luigia Petre and Emil Sekerinski. Computer and Information Science Series. Chapman and Hall/CRC, Apr. 2016. Chap. Modelling and Refining Hybrid Systems in Event-B and Rodin, pp. 29–42. ISBN: 978-1-49-870158-7.

[Ban+11] Richard Banach et al. “Formalising the Continuous/Discrete Modeling Step”. In: *Proceedings 15th International Refinement Workshop, Refine 2011, Limerick, Ireland, 20th June 2011*. Ed. by John Derrick, Eerke A. Boiten, and Steve Reeves. Vol. 55. EPTCS. 2011, pp. 121–138.

[Ban+12] Richard Banach et al. "ASM and Controller Synthesis". English. In: *Abstract State Machines, Alloy, B, VDM, and Z*. Ed. by John Derrick et al. Vol. 7316. LNCS. Springer Berlin Heidelberg, 2012, pp. 51–64. ISBN: 978-3-642-30884-0.

[Ban+14] Richard Banach et al. "A Continuous ASM Modelling Approach to Pacemaker Sensing". In: *ACM Trans. Softw. Eng. Methodol.* 24.1 (Oct. 2014), 2:1–2:40. ISSN: 1049-331X.

[Ban+15] Richard Banach et al. "Core Hybrid Event-B I: Single Hybrid Event-B machines". In: *Science of Computer Programming* (2015). ISSN: 0167-6423.

[Ban+17] Richard Banach et al. "Core Hybrid Event-B II: Multiple cooperating Hybrid Event-B machines". In: *Science of Computer Programming* 139 (2017), pp. 1–35. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2016.12.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0167642317300229`.

[Ban13] Richard Banach. "Pliant Modalities in Hybrid Event-B". In: *Theories of Programming and Formal Methods*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8051. LNCS. Springer Berlin Heidelberg, 2013, pp. 37–53. ISBN: 978-3-642-39697-7.

[Ban16] Richard Banach. "Formal Refinement and Partitioning of a Fuel Pump System for Small Aircraft in Hybrid Event-B". In: *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. July 2016, pp. 65–72.

[BC04] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004, p. 470. ISBN: 3-540-20854-2.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. "Combining Coq and Gappa for Certifying Floating-Point Programs". In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 59–74.

[BK89] Ralph-Joan Back and Reino Kurki-Suonio. "Decentralization of process nets with centralized control". In: *Distributed Computing* 3 (June 1989). DOI: `10.1007/BF01558665`.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. "Coquelicot: A User-Friendly Library of Real Analysis for Coq". English. In: *Mathematics in Computer Science* 9.1 (2015), pp. 41–62. ISSN: 1661-8270.

[BM13] Michael Butler and Issam Maamria. "Practical Theory Extension in Event-B". In: *Theories of Programming and Formal Methods*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8051. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 67–81. ISBN: 978-3-642-39697-7.

[Bol+14] Sylvie Boldo et al. "Trusting computations: A mechanized proof from partial differential equations to actual program". In: *Computers & Mathematics with Applications* 68.3 (2014), pp. 325–352. ISSN: 0898-1221.

[BP13] Timothy Bourke and Marc Pouzet. "Zélus: A Synchronous Language with ODEs". In: *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*. Philadelphia, USA, Mar. 2013, pp. 113–118. URL: `http://www.di.ens.fr/~pouzet/bib/hscc13.pdf`.

[BPP00] Ralph-Johan Back, Luigia Petre, and Ivan Porres. “Generalizing Action Systems to Hybrid Systems”. In: *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. FTRTFT ’00. London, UK, UK: Springer-Verlag, 2000, pp. 202–213. ISBN: 3-540-41055-4.

[CÁS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. “Flow*: An Analyzer for Non-linear Hybrid Systems”. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263. ISBN: 978-3-642-39799-8.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. The MIT Press, Dec. 1999. ISBN: 9780262032704.

[CJR95] Zhou Chaochen, Wang Ji, and Anders P. Ravn. “A Formal Description of Hybrid Systems”. In: *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*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Vol. 1066. Lecture Notes in Computer Science. Springer, 1995, pp. 511–530.

[Cuo+12] Pascal Cuoq et al. “Frama-C”. In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247. ISBN: 978-3-642-33826-7.

[Dup+18a] Guillaume Dupont et al. “Hybrid Systems and Event-B: A Formal Approach to Signalised Left-Turn Assist”. In: *New Trends in Model and Data Engineering*. Springer International Publishing, 2018, pp. 153–158.

[Dup+18b] Guillaume Dupont et al. “Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Michael Butler et al. Cham: Springer International Publishing, 2018, pp. 155–170. DOI: `10.1007/978-3-319-91271-4_11`.

[Dup+19] Guillaume Dupont et al. “Handling Refinement of Continuous Behaviors: A Refinement and Proof Based Approach with Event-B”. In: *13th International Symposium TASE*. IEEE Computer Society Press, 2019, pp. 9–16.

[Dup+20a] Guillaume Dupont et al. “An Event-B Based Generic Framework for Hybrid Systems Formal Modelling”. In: *16th International Conference on integrated Formal Methods, iFM 2020*. Ed. by Carlo A. Furia, Brijesh Dongol, and Elena Troubitsyna. Vol. 12546. LNCS. Springer, 2020, pp. 82–102.

[Dup+20b] Guillaume Dupont et al. “Embedding Approximation in Event-B: Safe Hybrid System Design using Proof and Refinement”. In: *22nd International Conference on Formal Engineering Methods, ICFEM 2020*. Ed. by Jin Song Dong and Jim McCarthy. Vol. 12531. LNCS. Springer, 2020, pp. 251–267. DOI: `10.1007/978-3-030-63406-3\_15`.

[Dup+20c] Guillaume Dupont et al. “Formally Verified Architecture Patterns of Hybrid Systems Using Proof and Refinement with Event-B”. In: *7th International Conference, ABZ 2020, Proceedings*. Ed. by Alexander Rashke and Dominique Méry. Vol. 12071. LNCS. Springer, 2020, pp. 155–170.

[Egg+11] Andreas Eggers et al. "Improving SAT Modulo ODE for Hybrid Systems Analysis by Combining Different Enclosure Methods". In: *Software Engineering and Formal Methods: 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Berlin, Heidelberg: Springer, 2011, pp. 172–187. ISBN: 978-3-642-24690-6.

[FGP07] Georgios E. Fainekos, Antoine Girard, and George J. Pappas. "Hierarchical Synthesis of Hybrid Controllers from Temporal Logic Specifications". In: *Hybrid Systems: Computation and Control, 10th International Workshop, HSCC 2007, Proceedings*. 2007, pp. 203–216.

[FM07] Jean-Christophe Filliâtre and Claude Marché. "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification". In: *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*. Ed. by Werner Damm and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 173–177. ISBN: 978-3-540-73368-3.

[Frä+07] Martin Fränzle et al. "Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure". In: *Journal on Satisfiability, Boolean Modeling and Computation* 1.3-4 (2007), pp. 209–236.

[Fre+11] Goran Frehse et al. "SpaceEx: Scalable Verification of Hybrid Systems". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 379–395. ISBN: 978-3-642-22110-1.

[Fre08] Goran Frehse. "PHAVer: algorithmic verification of hybrid systems past HyTech". In: *International Journal STTT* 10.3 (2008), pp. 263–279. ISSN: 1433-2787.

[Ful+15] Nathan Fulton et al. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *25th CADE Conference*. Vol. 9195. LNCS. Springer, 2015, pp. 527–538.

[GJP08] Antoine Girard, A. Agung Julius, and George J. Pappas. "Approximate Simulation Relations for Hybrid Systems". In: *Discrete Event Dynamic Systems* 18.2 (2008), pp. 163–179.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". In: *Automated Deduction – CADE-24*. Ed. by Maria Paola Bonacina. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–214. ISBN: 978-3-642-38574-2.

[GP07] Antoine Girard and George J. Pappas. "Approximation Metrics for Discrete and Continuous Systems". In: *IEEE Trans. Automat. Contr.* 52.5 (2007), pp. 782–798.

[Hal+91] Nicolas Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[Hen+98] Thomas A. Henzinger et al. "What's Decidable about Hybrid Automata?" In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94–124. ISSN: 0022-0000. DOI: `https://doi.org/10.1006/jcss.1998.1581`. URL: `http://www.sciencedirect.com/science/article/pii/S0022000098915811`.

[Hen00] Thomas A. Henzinger. "The Theory of Hybrid Automata". English. In: *Verification of Digital and Hybrid Systems*. Ed. by M. Kemal Inan and Robert P. Kurshan. Vol. 170. NATO ASI Series. Springer Berlin Heidelberg, 2000, pp. 265–292. ISBN: 978-3-642-64052-0.

[Her+12] Heber Herencia-Zapana et al. "PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL". In: *NASA Formal Methods*. Ed. by Alwyn E. Goodloe and Suzette Person. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 147–161. ISBN: 978-3-642-28891-3.

[HHW97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. "HyTech: A Model Checker for Hybrid Systems". English. In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (1997), pp. 110–122. ISSN: 1433-2779.

[Hoa69] Charles A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[Imm+18] Fabian Immler et al. "ARCH-COMP18 Category Report: Continuous and Hybrid Systems with Nonlinear Dynamics". In: *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems, ARCH@ADHS 2018, Oxford, UK, July 13, 2018*. Ed. by Goran Frehse et al. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 53–70.

[Imm+19] Fabian Immler et al. "ARCH-COMP19 Category Report: Continuous and Hybrid Systems with Nonlinear Dynamics". In: *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systemsi, part of CPS-IoT Week 2019, Montreal, QC, Canada, April 15, 2019*. Ed. by Goran Frehse and Matthias Althoff. Vol. 61. EPiC Series in Computing. EasyChair, 2019, pp. 41–61.

[Imm18] Fabian Immler. "A Verified ODE Solver and the Lorenz Attractor". In: *J. Autom. Reasoning* 61.1-4 (2018), pp. 73–111.

[IT19] Fabian Immler and Christoph Traut. "The Flow of ODEs: Formalization of Variational Equation and Poincaré Map". In: *J. Autom. Reasoning* 62.2 (2019), pp. 215–236.

[Jif94] He Jifeng. "A Classical Mind". In: ed. by A. W. Roscoe. Prentice Hall International (UK) Ltd., 1994. Chap. From CSP to Hybrid Systems, pp. 171–189. ISBN: 0-13-294844-3.

[Kou+13] Yanni Kouskoulas et al. "Certifying the Safe Design of a Virtual Fixture Control Algorithm for a Surgical Robot". In: *Hybrid Systems: Computation and Control (part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013*. Ed. by Calin Belta and Franjo Ivancic. ACM, 2013, pp. 263–272.

[LB03] Michael Leuschel and Michael Butler. "ProB: A Model Checker for B". In: *FME 2003: Formal Methods*. Ed. by Keijiro Araki, Stefania Gnesi, and Dino Mandrioli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 855–874. ISBN: 978-3-540-45236-2.

[LBT17] Simon Lunel, Benoît Boyer, and Jean-Pierre Talpin. "Compositional Proofs in Differential Dynamic Logic dL". In: *17th International Conference on Application of Concurrency to System Design, ACSD*. IEEE Computer Society, 2017, pp. 19–28.

[Lee08] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369.

[Liu+10a] Jiang Liu et al. "A Calculus for Hybrid CSP". In: *Programming Languages and Systems*. Ed. by Kazunori Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–15. ISBN: 978-3-642-17164-2.

[Liu+10b] Jiang Liu et al. "A Calculus for Hybrid CSP". In: *Programming Languages and Systems - 8th APLAS Symposium*. Ed. by Kazunori Ueda. Vol. 6461. LNCS. Springer, 2010, pp. 1–15.

[LP16] Sarah M. Loos and André Platzer. "Differential Refinement Logic". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16. New York, NY, USA: ACM, 2016, pp. 505–514. ISBN: 978-1-4503-4391-6.

[LS14] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. 1.5. LeeSeshia.org, 2014. ISBN: 978-1-312-42740-2. URL: `http://leeseshia.org/`.

[Lun+19] Simon Lunel et al. "Parallel Composition and Modular Verification of Computer Controlled Systems in Differential Dynamic Logic". In: *Formal Methods – The Next 30 Years*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Cham: Springer International Publishing, 2019, pp. 354–370. ISBN: 978-3-030-30942-8.

[Mar07] Claude Marché. "Jessie: An Intermediate Language for Java and C Verification". In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. Freiburg, Germany: ACM, 2007, pp. 1–2. ISBN: 978-1-59593-677-6.

[Max68] James Clerk Maxwell. "On governors". In: *Proceedings of the Royal Society of London*. Jan. 1868. DOI: `10.1098/rspl.1867.0055`.

[MH06] Larissa Meinicke and Ian J. Hayes. "Continuous Action System Refinement". In: *Proceedings of the 8th International Conference on Mathematics of Program Construction*. MPC'06. Kuressaare, Estonia: Springer-Verlag, 2006, pp. 316–337.

[Mil89] Robin Milner. "Communication and Concurrency". In: Prentice Hall, 1989. ISBN: 0131149849.

[Muñ+15] C. Muñoz et al. "DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems". In: *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*. 2015, 5A1-1-5A1–12. DOI: `10.1109/DASC.2015.7311421`.

[Muñ+16] César A. Muñoz et al. "Unmanned Aircraft Systems in the National Airspace System: A Formal Methods Perspective". In: *ACM SIGLOG News* 3.3 (Aug. 2016), pp. 67–76. DOI: `10.1145/2984450.2984459`. URL: `https://doi.org/10.1145/2984450.2984459`.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002. ISBN: 3-540-43376-7.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752. ISBN: 978-3-540-47252-0.

[OS01] Sam Owre and N. Shankar. *Theory Interpretations in PVS*. Tech. rep. 2001.

[PC09a] André Platzer and Edmund M. Clarke. “Computing Differential Invariants of Hybrid Systems as Fixedpoints”. In: *Form. Methods Syst. Des.* 35.1 (2009), pp. 98–120. DOI: `10.1007/s10703-009-0079-8`.

[PC09b] André Platzer and Edmund M. Clarke. “Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study”. In: *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings.* Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 547–562. ISBN: 978-3-642-05089-3.

[Pla08] André Platzer. “Differential Dynamic Logic for Hybrid Systems”. In: *Journal of Automated Reasoning* 41.2 (2008), pp. 143–189. ISSN: 1573-0670.

[Pla15] André Platzer. “A Uniform Substitution Calculus for Differential Dynamic Logic”. In: *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings.* Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 467–481. ISBN: 978-3-319-21401-6.

[PQ09] André Platzer and Jan-David Quesel. “European Train Control System: A Case Study in Formal Verification”. In: *Formal Methods and Software Engineering: 11th International Conference on Formal Engineering Methods ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings.* Ed. by Karin Breitman and Ana Cavalcanti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 246–265. ISBN: 978-3-642-10373-5.

[Que+16] Jan-David Quesel et al. “How to model and prove hybrid systems with KeYmaera: a tutorial on safety”. In: *International Journal STTT* 18.1 (2016), pp. 67–91. ISSN: 1433-2787.

[SA14] Wen Su and Jean-Raymond Abrial. “Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System”. In: *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings.* Ed. by Frédéric Boniol et al. Springer International Publishing, 2014, pp. 19–35. ISBN: 978-3-319-07512-9.

[Sai13] Ronan Saillard. “Dedukti : a Universal Proof Checker”. In: *Foundation of Mathematics for Computer-Aided Formalization Workshop.* Padova, Italy, Jan. 2013.

[SAZ14] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. “Formalizing hybrid systems with Event-B and the Rodin Platform”. In: *Science of Computer Programming* 94, Part 2 (2014). Abstract State Machines, Alloy, B, VDM, and Z Selected and extended papers from ABZ 2012, pp. 164–202. ISSN: 0167-6423.

[Sta+19] Paulius Stankaitis et al. “Modelling Hybrid Train Speed Controller using Proof and Refinement”. In: *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS).* 2019, pp. 107–113.

[STW14] Steve Schneider, Helen Treharne, and Heike Wehrheim. “The behavioural semantics of Event-B refinement”. In: *Formal aspects of computing* 26.2 (2014), pp. 251–280.

[SWD21] S. T. Slagel, L. White, and A. Dutle. “Formal Verification of Semi-algebraic Sets and Real Analytic Functions”. In: *Certified Programs and Proofs (CPP) 2021.* 2021.

[WZZ15] Shuling Wang, Naijun Zhan, and Liang Zou. "An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems". In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM*. Ed. by Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 382–399.

[Zou+14] Liang Zou et al. "Verifying Chinese Train Control System under a Combined Scenario by Theorem Proving". In: *Verified Software: Theories, Tools, Experiments*. Ed. by Ernie Cohen and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 262–280. ISBN: 978-3-642-54108-7.

Part III

Appendices

Appendix A

Theories

This first appendix shows the complete Event-B model of the general theories presented in this manuscript (Chapter 4).

Note that domain-specific theories are presented in the sections of their associated cases studies, in Appendix B.

A.1 Theories of General Algebra

This section details the theories for general algebra (monoids, groups, etc.). These theories have been formally presented in Section 4.3.2.1.

Theory of Monoids

Listings A.1: Monoid theory

```
THEORY
  TYPE PARAMETERS M
  OPERATORS
    associative predicate (op: (M × M) → M)
      direct definition
        ∀ x, y, z · x ∈ M ∧ y ∈ M ∧ z ∈ M ⇒ op(x ↦ op(y ↦ z)) = op(op(x ↦
            y) ↦ z)
    neutral predicate (op: (M × M) → M, e: M)
      well−definedness M ≠ ∅
      direct definition
        ∀ x · x ∈ M ⇒ (op(x ↦ e) = x ∧ op(e ↦ x) = x)
    Monoid predicate (op: (M × M) → M, e: M)
      well−definedness M ≠ ∅
      direct definition
        associative(op) ∧ neutral(op, e)
  THEOREMS
    neutralUnicity:
      ∀ op, e · op ∈ ((M × M) → M) ∧ e ∈ M ∧ Monoid(op, e) ⇒ (
        ∀ x · x ∈ M ∧ neutral(op, x) ⇒ x = e
      )
END
```

Theory of Groups

Listings A.2: Group theory

```
THEORY
  IMPORT THEORY Monoid
  TYPE PARAMETERS G
  OPERATORS
    invertible predicate (op: (G × G) → G, e: G)
      well−definedness G ≠ ∅
      direct definition
        ∀ x · x ∈ G ⇒ (∃ y · y ∈ G ∧ op(x ↦ y) = e ∧ op(y ↦ x) = e)
    commutative predicate (op: (G × G) → G)
      well−definedness G ≠ ∅
      direct definition
        ∀ x, y · x ∈ G ∧ y ∈ G ⇒ op(x ↦ y) = op(y ↦ x)
    Group predicate (op: (G × G) → G, e: G)
      well−definedness G ≠ ∅
      direct definition
        Monoid(op, e) ∧ invertible(op, e)
    AbelianGroup predicate (op: (G × G) → G, e: G)
      well−definedness G ≠ ∅
      direct definition
        Group(op, e) ∧ commutative(op)
    inverses predicate (op: (G × G) → G, e: G, x: G, y: G)
      well−definedness G ≠ ∅
      direct definition
        invertible(op, e) ⇒ (op(x ↦ y) = e ∧ op(y ↦ x) = e)
  THEOREMS
    inversesCommutative:
      ∀ op, e, x, y · op ∈ ((G × G) → G) ∧ e ∈ G ∧ x ∈ G ∧ y ∈ G
        ∧ Group(op, e) ⇒ (inverses(op, e, x, y) ⇔ inverses(op, e, y, x))
    latinSquare:
      ∀ op, e, x, y · op ∈ ((G × G) → G) ∧ e ∈ G ∧ x ∈ G ∧ y ∈ G
        ∧ Group(op, e) ⇒ (∃ g · g ∈ G ∧ (op(x ↦ g) = y
          ∧ (∀ g2 · g2 ∈ G ∧ op(x ↦ g2) = y ⇒ g = g2))
        )
    leftCancellation:
      ∀ op, e· op ∈ ((G × G) → G) ∧ e ∈ G ∧ Group(op, e) ⇒ (
        ∀ a, b, c · a ∈ G ∧ b ∈ G ∧ c ∈ G ⇒
          ((op(a ↦ b) = op(a ↦ c)) ⇔ (b = c))
      )
    rightCancellation:
      ∀ op, e· op ∈ ((G × G) → G) ∧ e ∈ G ∧ Group(op, e) ⇒ (
        ∀ a, b, c · a ∈ G ∧ b ∈ G ∧ c ∈ G ⇒
          ((op(b ↦ a) = op(c ↦ a)) ⇔ (b = c))
      )
    inverseEqn:
      ∀ op, e· op ∈ ((G × G) → G) ∧ e ∈ G ∧ Group(op, e) ⇒ (
        ∀ x, y · x ∈ G ∧ y ∈ G ⇒ (
          op(x ↦ y) = e ⇔ inverses(op, e, x, y)
        )
      )
    zeroInverse:
      ∀ op, e · op ∈ ((G × G) → G) ∧ e ∈ G ∧ invertible(op, e)
        ∧ neutral(op, e) ⇒ inverses(op, e, e, e)
END
```

Theory of Rings

Listings A.3: Ring theory

```
THEORY
  IMPORT THEORY Group
  TYPE PARAMETERS A
  OPERATORS
    distributive predicate (oplus: (A × A) → A, otimes: (A × A) → A)
      well−definedness A ≠ ∅
      direct definition
        ∀ x, y, z · x ∈ A ∧ y ∈ A ∧ z ∈ A ⇒ (
          otimes(x ↦ oplus(y ↦ z)) = oplus(otimes(x ↦ y) ↦ otimes(x ↦ z))
              ∧
          otimes(oplus(y ↦ z) ↦ x) = oplus(otimes(y ↦ x) ↦ otimes(z ↦ x))
        )
    integral predicate (otimes: (A × A) → A, azero: A)
      well−definedness A ≠ ∅ ∧ A ≠ {azero}
      direct definition
        ∀ x, y · x ∈ A ∧ y ∈ A ⇒
          (otimes(x ↦ y) = azero ⇒ (x = azero ∨ y = azero))
    Ring predicate (oplus: (A × A) → A, otimes: (A × A) → A,
        azero: A, aunit: A)
      well−definedness A ≠ ∅
      direct definition
        AbelianGroup(oplus, azero) ∧ Monoid(otimes, aunit)
          ∧ distributive(oplus, otimes)
    CommutativeRing predicate (oplus: (A × A) → A, otimes: (A × A) → A,
        azero: A, aunit: A)
      well−definedness A ≠ ∅
      direct definition
        Ring(oplus, otimes, azero, aunit) ∧ commutative(otimes)
    nonZeroInvertible predicate (op: (A × A) → A,
        azero: A, aunit: A)
      well−definedness A ≠ ∅ ∧ (azero ≠ aunit)
      direct definition
        ∀ x · x ∈ A ∧ x ≠ azero
```

```
          ⇒ (∃ y · y ∈ A ∧ y ≠ azero ⇒ (op(x ↦ y) = aunit ∧ op(y ↦ x) =
            aunit))
    nonZeroInverses predicate (op: (A × A) → A, azero: A, aunit: A, x: A, y: A)
      well−definedness A ≠ ∅ ∧ (azero ≠ aunit)
      direct definition
        nonZeroInvertible(op, azero, aunit) ∧ x ≠ azero
          ⇒ ((op(x ↦ y) = aunit) ∧ (op(y ↦ x) = aunit))
    DivisionRing predicate (oplus: (A × A) → A, otimes: (A × A) → A,
         azero: A, aunit: A)
      well−definedness A ≠ ∅
      direct definition
        (azero ≠ aunit) ∧ Ring(oplus, otimes, azero, aunit)
          ∧ nonZeroInvertible(otimes, azero, aunit)
    Field predicate (oplus: (A × A) → A, otimes: (A × A) → A,
         azero: A, aunit: A)
      well−definedness A ≠ ∅
      direct definition
        (azero ≠ aunit) ∧
        Ring(oplus, otimes, azero, aunit) ∧
        nonZeroInvertible(otimes, azero, aunit) ∧
        integral(otimes, azero) ∧
        commutative(otimes)
    absorbing predicate (op: (A × A) → A, azero: A)
      direct definition
        ∀ x · x ∈ A ⇒ (op(x ↦ azero) = azero ∧ op(azero ↦ x) = azero)
  THEOREMS
    zeroAbsorbing:
      ∀ oplus, otimes, azero, aunit · oplus ∈ ((A × A) → A) ∧
        otimes ∈ ((A × A) → A) ∧ azero ∈ A ∧ aunit ∈ A ∧
          Ring(oplus, otimes, azero, aunit) ⇒ absorbing(otimes, azero)
    plusInverseLeftDistribution:
      ∀ oplus, otimes, azero, aunit · oplus ∈ ((A × A) → A) ∧
        otimes ∈ ((A × A) → A) ∧ azero ∈ A ∧ aunit ∈ A ∧ Ring(oplus, otimes,
            azero, aunit)
          ⇒ (∀ a, b, a1 · a ∈ A ∧ b ∈ A ∧ a1 ∈ A ∧ inverses(oplus, azero, a, a1
              )
              ⇒ inverses(oplus, azero, otimes(a ↦ b), otimes(a1 ↦ b))
          )
    plusInverseRightDistribution:
      ∀ oplus, otimes, azero, aunit · oplus ∈ ((A × A) → A) ∧
        otimes ∈ ((A × A) → A) ∧ azero ∈ A ∧ aunit ∈ A ∧
        Ring(oplus, otimes, azero, aunit) ⇒ (
          ∀ a, b, b1 · a ∈ A ∧ b ∈ A ∧ b1 ∈ A ∧ inverses(oplus, azero, b, b1) ⇒
              (
            inverses(oplus, azero, otimes(a ↦ b), otimes(a ↦ b1)))
        )
    ringLeftCancellation:
      ∀ oplus, otimes, azero, aunit · oplus ∈ ((A × A) → A) ∧ otimes ∈ ((A × A)
          → A) ∧ azero ∈ A ∧ aunit ∈ A ∧ azero ≠ aunit ∧ Ring(oplus, otimes,
         azero, aunit) ∧ integral(otimes, azero) ⇒ (
        ∀ a, b, c · a ∈ A ∧ b ∈ A ∧ c ∈ A ∧ a ≠ azero ⇒
          ((otimes(a ↦ b) = otimes(a ↦ c)) ⇔ (b = c))
      )
    ringRightCancellation:
      ∀ oplus,otimes,azero,aunit · oplus ∈ ((A × A) → A) ∧
        otimes ∈ ((A × A) → A) ∧ azero ∈ A ∧ aunit ∈ A ∧ azero ≠ aunit ∧
        Ring(oplus,otimes,azero,aunit) ∧ integral(otimes,azero) ⇒ (
          ∀ a, b, c · a ∈ A ∧ b ∈ A ∧ c ∈ A ∧ a ≠ azero ⇒
            ((otimes(b ↦ a) = otimes(c ↦ a)) ⇔ (b = c))
        )
    nonZeroInverseNotZero:
      ∀ oplus,otimes,azero,aunit,x,y · oplus ∈ ((A × A) → A) ∧
        otimes ∈ ((A × A) → A) ∧ azero ∈ A ∧ aunit ∈ A ∧
        Ring(oplus,otimes,azero,aunit) ∧
        nonZeroInvertible(otimes,azero,aunit) ∧
        x ∈ A ∧ y ∈ A ∧ x ≠ azero ∧ nonZeroInverses(otimes,azero,aunit,x,y)
          ⇒ y ≠ azero
END
```

Theory of Relations

Listings A.4: Relation theory

```
THEORY
  IMPORT THEORY Ring
  TYPE PARAMETERS S
  OPERATORS
    reflexive predicate (rel: S ↔ S)
      well—definedness S ≠ ∅
      direct definition
        ∀ x · x ∈ S ⇒ ((x ↦ x) ∈ rel)
    antireflexive predicate (rel: S ↔ S)
      well—definedness S ≠ ∅
      direct definition
        ∀ x · x ∈ S ⇒ (x ↦ x ∉ rel)
    symetrical predicate (rel: S ↔ S)
      well—definedness S ≠ ∅
      direct definition
        ∀ x,y · x ∈ S ∧ y ∈ S ⇒ (
          (x ↦ y ∈ rel) ⇒ (y ↦ x ∈ rel)
        )
    asymetrical predicate (rel: S ↔ S)
      well—definedness S ≠ ∅
      direct definition
        ∀ x,y · x ∈ S ∧ y ∈ S ⇒ (
          (x ↦ y ∈ rel) ⇒ (y ↦ x ∉ rel)
        )
    antisymetrical predicate (rel: S ↔ S)
      well—definedness S ≠ ∅
      direct definition
    ∀ x,y · x ∈ S ∧ y ∈ S ⇒ (
      (x ↦ y ∈ rel) ∧ (y ↦ x ∈ rel) ⇒ x = y
    )
transitive predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    ∀ x, y, z · x ∈ S ∧ y ∈ S ∧ z ∈ S ⇒ (
      (x ↦ y ∈ rel) ∧ (y ↦ z ∈ rel) ⇒ (x ↦ z ∈ rel)
    )
total predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    ∀ x, y · x ∈ S ∧ y ∈ S ⇒ ((x ↦ y ∈ rel) ∨ (y ↦ x ∈ rel))
equivalence predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    reflexive(rel) ∧ symetrical(rel) ∧ transitive(rel)
order predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    reflexive(rel) ∧ transitive(rel) ∧ antisymetrical(rel)
strict expression (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { x ↦ y | x ↦ y ∈ rel ∧ x ≠ y }
wellFounded predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    ∀ X · X ⊆ S ∧ X ≠ ∅ ⇒ (
      ∃ m · m ∈ X ∧ (∀ x · x ∈ X ⇒ (x ↦ m ∉ rel))
    )
wellPartialOrder predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    order(rel) ∧ wellFounded(strict(rel))
wellOrder predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    order(rel) ∧ total(rel) ∧ wellFounded(strict(rel))
covers predicate (rel: S ↔ S,a: S,b: S)
  well−definedness S ≠ ∅ ∧ order(rel)
  direct definition
    (a ↦ b ∈ rel) ∧ a ≠ b ∧ (∀ c · c ∈ S ∧ (a ↦ c ∈ rel) ∧
      (c ↦ b ∈ rel) ⇒ ((c = a) ∨ (b = a)))
compose expression (rel1: S ↔ S,rel2: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { x, z · x ∈ S ∧ z ∈ S ∧ (∃ y · y ∈ S ⇒
      ((x ↦ y ∈ rel1) ∧ (y ↦ z ∈ rel2)))
    | x ↦ z }
```

```
converse expression (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { x, y · x ∈ S ∧ y ∈ S ∧ y ↦ x ∈ rel
      | x ↦ y }
complement expression (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { x,y · x ∈ S ∧ y ∈ S ∧ x ↦ y ∉ rel
      | x ↦ y }
equality expression ()
  well−definedness S ≠ ∅
  direct definition
    { x ↦ y | x ∈ S ∧ y ∈ S ∧ x = y }
preorder predicate (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    reflexive(rel) ∧ transitive(rel)
equivalenceClass expression (rel: S ↔ S,x: S)
  well−definedness S ≠ ∅ ∧ equivalence(rel)
  direct definition
    { y · y ∈ S ∧ (x ↦ y) ∈ rel | y }
leftGeneralized expression (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { x,P · x ∈ S ∧ P ∈ ℙ(S) ∧ P ≠ ∅ ∧ (∀ y · y ∈ P ⇒ (x ↦ y ∈ rel)) | x
        ↦ P }
rightGeneralized expression (rel: S ↔ S)
  well−definedness S ≠ ∅
  direct definition
    { P,x · P ∈ ℙ(S) ∧ P ≠ ∅ ∧ x ∈ S ∧ (∀ y · y ∈ P ⇒ (y ↦ x ∈ rel)) | P
        ↦ x }
upperBound predicate (ord: S ↔ S,T: ℙ(S),B: S)
  direct definition
    ∀ t · t ∈ T ⇒ t ↦ B ∈ ord
lowerBound predicate (ord: S ↔ S,T: ℙ(S),B: S)
  direct definition
    ∀ t · t ∈ T ⇒ B ↦ t ∈ ord
bounds predicate (ord: S ↔ S,T: ℙ(S),m: S,M: S)
  direct definition
    lowerBound(ord,T,m) ∧ upperBound(ord,T,M)
upperBounded predicate (ord: S ↔ S,T: ℙ(S))
  direct definition
    ∃ M · M ∈ S ∧ upperBound(ord,T,M)
lowerBounded predicate (ord: S ↔ S,T: ℙ(S))
  direct definition
    ∃ m · m ∈ S ∧ lowerBound(ord,T,m)
bounded predicate (ord: S ↔ S,T: ℙ(S))
  direct definition
    ∃ m, M · m ∈ S ∧ M ∈ S ∧ bounds(ord,T,m,M)
    supremum predicate (ord: S ↔ S,T: ℙ(S),M: S)
      direct definition
        upperBound(ord,T,M) ∧
        (∀ m · m ∈ S ∧ upperBound(ord,T,m) ⇒ M ↦ m ∈ ord)
    infimum predicate (ord: S ↔ S,T: ℙ(S),m: S)
      direct definition
        lowerBound(ord,T,m) ∧
        (∀ M · M ∈ S ∧ lowerBound(ord,T,M) ⇒ M ↦ m ∈ ord)
    maximal predicate (ord: S ↔ S,T: ℙ(S),M: T)
      direct definition
        ∀ x · x ∈ T ∧ M ↦ x ∈ ord ⇒ x = M
    minimal predicate (ord: S ↔ S,T: ℙ(S),M: T)
      direct definition
        ∀ x · x ∈ T ∧ x ↦ M ∈ ord ⇒ x = M
    maximum predicate (ord: S ↔ S,T: ℙ(S),M: S)
      direct definition
        (M ∈ T) ∧ (∀ x · x ∈ T ⇒ x ↦ M ∈ ord)
    minimum predicate (ord: S ↔ S,T: ℙ(S),M: S)
      direct definition
        (M ∈ T) ∧ (∀ x · x ∈ T ⇒ M ↦ x ∈ ord)
    hasMaximum predicate (ord: S ↔ S,T: ℙ(S))
      direct definition
        ∃ M · M ∈ T ∧ maximum(ord,T,M)
    hasMinimum predicate (ord: S ↔ S,T: ℙ(S))
      direct definition
        ∃ m · m ∈ T ∧ minimum(ord,T,m)
    monoidCompatible predicate (op: (S × S) → S,e: S,rel: S ↔ S)
      well−definedness S ≠ ∅ ∧ Monoid(op,e) ∧ order(rel)
      direct definition
        ∀ x, y, z · x ∈ S ∧ y ∈ S ∧ z ∈ S ∧ (x ↦ y ∈ rel) ⇒ (
          ((op(x ↦ z) ↦ op(y ↦ z)) ∈ rel) ∧
          ((op(z ↦ x) ↦ op(z ↦ y)) ∈ rel)
        )
    ringCompatible predicate (oplus: (S × S) → S,otimes: (S × S) → S,
         azero: S,unit: S,rel: S ↔ S)
      well−definedness S ≠ ∅ ∧ Ring(oplus,otimes,azero,unit) ∧ order(rel)
      direct definition
        monoidCompatible(oplus,azero,rel) ∧ (
         ∀ x, y · x ∈ S ∧ y ∈ S ∧ (azero ↦ x ∈ rel) ∧ (azero ↦ y ∈ rel) ⇒
          ((azero ↦ otimes(x ↦ y) ∈ rel) ∧ (azero ↦ otimes(y ↦ x) ∈ rel))
        )
  AXIOMATIC DEFINITIONS   operations:
    OPERATORS
      Gmax expression (ord: S ↔ S,T: ℙ(S)) : S
      Gmin expression (ord: S ↔ S,T: ℙ(S)) : S
      Gsup expression (ord: S ↔ S,T: ℙ(S)) : S
      Ginf expression (ord: S ↔ S,T: ℙ(S)) : S
    AXIOMS
      GmaxDef:
        ∀ ord,T · ord ∈ S ↔ S ∧ T ⊆ S ∧ hasMaximum(ord,T)
        ⇒ (maximum(ord ,T,Gmax(ord ,T)))
    GminDef :
      ∀ ord ,T · ord ∈ S ↔ S ∧ T ⊆ S ∧ hasMinimum(ord ,T)
        ⇒ (minimum(ord ,T,Gmin(ord ,T)))
    GsupDef :
      ∀ ord ,T · ord ∈ S ↔ S ∧ T ⊆ S ∧ upperBounded(ord ,T)
        ⇒ (supremum(ord ,T,Gsup(ord ,T)))
    GinfDef :
      ∀ ord ,T · ord ∈ S ↔ S ∧ T ⊆ S ∧ lowerBounded(ord ,T)
        ⇒ (infimum(ord ,T,Ginf(ord ,T)))
THEOREMS
  converseDomain :
    ∀ rel · rel ∈ (S ↔ S) ⇒ (dom(converse(rel)) = ran(rel))
  converseRange :
    ∀ rel · rel ∈ (S ↔ S) ⇒ (ran(converse(rel)) = dom(rel))
  converseInvolutive :
    ∀ rel · rel ∈ (S ↔ S) ⇒ (converse(converse(rel)) = rel)
  converseSymetry :
    ∀ rel · rel ∈ (S ↔ S) ∧ symetrical(rel) ⇒ symetrical(converse(rel))
  converseReflexivity :
    ∀ rel · rel ∈ (S ↔ S) ∧ reflexive(rel) ⇒ reflexive(converse(rel))
  converseAntireflexivity :
    ∀ rel · rel ∈ (S ↔ S) ∧ antireflexive(rel) ⇒ antireflexive(converse(
        rel))
  complementInvolutive :
    ∀ rel · rel ∈ (S ↔ S) ⇒ (complement(complement(rel)) = rel)
  complementConverse :
    ∀ rel · rel ∈ (S ↔ S) ⇒ (complement(converse(rel)) = converse(
        complement(rel)))
  complementSymetry :
    ∀ rel · rel ∈ (S ↔ S) ∧ symetrical(rel) ⇒ symetrical(complement(rel))
  complementReflexivity :
    ∀ rel · rel ∈ (S ↔ S) ∧ reflexive(rel) ⇒ antireflexive(complement(rel)
        )
  complementAntireflexivity :
    ∀ rel · rel ∈ (S ↔ S) ∧ antireflexive(rel) ⇒ reflexive(complement(rel)
        )
  totalRelationsReflexivity :
    ∀ rel · rel ∈ (S ↔ S) ∧ total(rel) ⇒ reflexive(rel)
  equivalenceClassEquity :
    ∀ rel ,x  ,y · rel ∈ (S ↔ S) ∧ equivalence(rel) ∧ x ∈ S ∧ y ∈ S ⇒ (
      ((x ↦ y) ∈ rel) ⇔ (equivalenceClass(rel , x) = equivalenceClass(rel ,
          y))
    )
  equivalenceClassNotEmpty :
    ∀ rel , x · rel ∈ (S ↔ S) ∧ equivalence(rel) ∧ x ∈ S ⇒
      (equivalenceClass(rel ,x) ≠ ∅)
  equivalenceClassCover :
    ∀ rel · rel ∈ (S ↔ S) ∧ equivalence(rel) ⇒ (
      (⋃ equivalenceClass(rel ,x) | x ∈ S) = S
      )
    equivalenceClassDisjoint:
      ∀ rel, x, y · rel ∈ (S ↔ S) ∧ equivalence(rel) ∧ x ∈ S ∧ y ∈ S ∧ (x ↦
           y ∉ rel) ⇒ (
        (equivalenceClass(rel,x) ∩ equivalenceClass(rel,y)) = ∅
      )
    supremumMaximal:
      ∀ ord, T, M · ord ∈ (S ↔ S) ∧ order(ord) ∧ T ⊆ S ∧ M ∈ S ⇒
        (supremum(ord,T,M) ∧ M ∈ T ⇒ maximal(ord,T,M))
    infimumMinimal:
      ∀ ord, T, M · ord ∈ (S ↔ S) ∧ order(ord) ∧ T ⊆ S ∧ M ∈ S ⇒
        (infimum(ord,T,M) ∧ M ∈ T ⇒ minimal(ord,T,M))
END
```

A.2 Theories of Real Numbers and Invervals

This section gives the theories of real numbers and intervals, as presented in Section 4.3.2.2. These theories define the important *RReal* type, that represent real numbers, with its associated operators (addition, subtraction, etc.) and relations (lower than, greater than, etc.). A number of properties are encoded as axioms and theorems.

The theory for intervals allow to define closed/open/unbounded intervals and handle them in models.

Theory of Reals

Listings A.5: Reals theory

```
THEORY
  IMPORT THEORY Relation
  AXIOMATIC DEFINITIONS
   Rdef:
     TYPES RReal
     OPERATORS
        Rzero expression () : RReal
        Rone expression () : RReal
        Rtwo expression () : RReal
     AXIOMS
        oneIsNotZero:
          Rone ≠ Rzero
        twoIsNotOne:
          Rtwo ≠ Rone
        twoIsNotZero:
          Rtwo ≠ Rzero
   Rorder:
     OPERATORS
        leq expression () : RReal ↔ RReal
        lt expression () : RReal ↔ RReal
        geq expression () : RReal ↔ RReal
        gt expression () : RReal ↔ RReal
```

```
      Rmax expression (A: ℙ(RReal)) : RReal
      Rmin expression (A: ℙ(RReal)) : RReal
      Rsup expression (A: ℙ(RReal)) : RReal
      Rinf expression (A: ℙ(RReal)) : RReal
    AXIOMS
      leqDef:
        order(leq)
      leqTotal:
        total(leq)
      ltDef:
        lt = strict(leq)
      geqDef:
        geq = converse(leq)
      gtDef:
        gt = strict(geq)
      supremumProperty:
        ∀ P · P ∈ ℙ(RReal) ∧ P ≠ ∅ ⇒ (
          (∃ m · m ∈ RReal ∧ upperBound(leq,P,m)) ⇒
          (∃ B · B ∈ RReal ∧ supremum(leq,P,B))
        )
      zeroLtOne:
        (Rzero ↦ Rone) ∈ lt
      oneLtTwo:
        (Rone ↦ Rtwo) ∈ lt
      RmaxDef:
        ∀ S · S ⊆ RReal ∧ hasMaximum(leq,S) ⇒ (Rmax(S) = Gmax(leq,S))
      RminDef:
        ∀ S · S ⊆ RReal ∧ hasMinimum(leq,S) ⇒ (Rmin(S) = Gmin(leq,S))
      RsupDef:
        ∀ S · S ⊆ RReal ∧ upperBounded(leq,S) ⇒ (Rsup(S) = Gsup(leq,S))
      RinfDef:
        ∀ S · S ⊆ RReal ∧ lowerBounded(leq,S) ⇒ (Rinf(S) = Ginf(leq,S))
  Rparts:
    OPERATORS
      RRealStar expression () : ℙ(RReal)
      RRealPlus expression () : ℙ(RReal)
      RRealMinus expression () : ℙ(RReal)
      RRealPlusStar expression () : ℙ(RReal)
      RRealMinusStar expression () : ℙ(RReal)
    AXIOMS
      realStarDef:
        RRealStar = { x | x ∈ RReal ∧ x ≠ Rzero }
      realPlusDef:
        RRealPlus = { x | Rzero ↦ x ∈ leq }
      realMinusDef:
        RRealMinus = { x | x ↦ Rzero ∈ leq }
      realPlusStarDef:
        RRealPlusStar = { x | Rzero ↦ x ∈ lt }
      realMinusStarDef:
        RRealMinusStar = { x | x ↦ Rzero ∈ lt }
```

```
Roperators:
  OPERATORS
    plus expression () : ℙ(RReal×RReal×RReal)
    times expression () : ℙ(RReal×RReal×RReal)
    uminus expression () : ℙ(RReal×RReal)
    inverse expression () : ℙ(RReal×RReal)
    minus expression () : ℙ(RReal×RReal×RReal)
    divide expression () : ℙ(RReal×RReal×RReal)
    abs expression () : ℙ(RReal×RReal)
    sqrt expression () : ℙ(RReal×RReal)
  AXIOMS
    plusType:
      plus ∈ ((RReal×RReal) → RReal)
    timesType:
      times ∈ ((RReal×RReal) → RReal)
    uminusType:
      uminus ∈ (RReal ⤖ RReal)
    inverseType:
      inverse ∈ (RRealStar ⤖ RRealStar)
    minusType:
      minus ∈ ((RReal×RReal) → RReal)
    divideType:
      divide ∈ ((RReal×RRealStar) → RReal)
    uminusDef:
      ∀ x · x ∈ RReal ⇒ (inverses(plus,Rzero,x,uminus(x)))
    minusDef:
      ∀ x, y · x ∈ RReal ∧ y ∈ RReal ⇒ (
        minus(x ↦ y) = plus(x ↦ uminus(y))
      )
    inverseDef:
      ∀ x · x ∈ RReal ∧ x ≠ Rzero ⇒ (
        times(x ↦ inverse(x)) = Rone
      )
    divideDef:
      ∀ x, y · x ∈ RReal ∧ y ∈ RReal ∧ y ≠ Rzero ⇒ (
        divide(x ↦ y) = times(x ↦ inverse(y))
      )
    absType:
      abs ∈ (RReal → RRealPlus)
    absPos:
      ∀ x · x ∈ RReal ⇒ (Rzero ↦ abs(x) ∈ leq)
    absZero:
      ∀ x · x ∈ RReal ⇒ ((x = Rzero) ⇔ (abs(x) = Rzero))
    absDef:
      ∀ x · x ∈ RReal ⇒ (
        ((x ↦ Rzero ∈ lt) ⇒ (abs(x) = uminus(x))) ∧
        ((Rzero ↦ x ∈ lt) ⇒ (abs(x) = x))
      )
    absTriangular:
      ∀ x, y · x ∈ RReal ∧ y ∈ RReal ⇒ (
          (abs(plus(x ↦ y)) ↦ plus(abs(x) ↦ abs(y)) ∈ leq)
        )
      absMinus:
        ∀ x · x ∈ RReal ⇒ (abs(uminus(x)) = abs(x))
      absMult:
        ∀ x, y · x ∈ RReal ∧ y ∈ RReal ⇒ (
          abs(times(x ↦ y)) = times(abs(x) ↦ abs(y))
        )
      sqrtType:
        sqrt ∈ (RRealPlus ↠ RRealPlus)
      sqrtDef:
        ∀ x · x ∈ RRealPlus ⇒ (times(sqrt(x) ↦ sqrt(x)) = x)
      twoDef:
        Rtwo = plus(Rone ↦ Rone)
 Rstructure:
   AXIOMS
      realField:
        Field(plus, times, Rzero, Rone)
      realIntegrity:
        integral(times, Rzero)
      ringCompatibility:
        ringCompatible(plus, times, Rzero, Rone, leq)
THEOREMS
   RzeroLtRtwo:
     Rzero ↦ Rtwo ∈ lt
   uminusOrderInversion:
     ∀ x · x ∈ RReal ∧ (Rzero ↦ x ∈ leq) ⇒ (uminus(x) ↦ Rzero ∈ leq)
   uminusInvolutive:
     ∀ x · x ∈ RReal ⇒ (uminus(uminus(x)) = x)
   uminusNeutrality:
     uminus(Rzero)=Rzero
   inverseInvolutive:
     ∀ x · x ∈ RReal ∧ x ≠ Rzero ⇒ (inverse(inverse(x)) = x)
   sqrtTimesDistrib:
     ∀ x, y · x ∈ RRealPlus ∧ y ∈ RRealPlus ⇒ (
       sqrt(times(x ↦ y)) = times(sqrt(x) ↦ sqrt(y))
     )
   sqrtOne:
     sqrt(Rone) = Rone
   sqrtZero:
     sqrt(Rzero) = Rzero
   realNotUpperBounded:
     ∀ x · x ∈ RReal ⇒ (∃ x2 · x2 ∈ RReal ∧ x ↦ x2 ∈ lt)
   realNotLowerBounded:
     ∀ x · x ∈ RReal ⇒ (∃ x2 · x2 ∈ RReal ∧ x2 ↦ x ∈ lt)
   realPlusNotBounded:
     ∀ x · x ∈ RRealPlus ⇒ (∃ x2 · x2 ∈ RRealPlus ∧ x ↦ x2 ∈ lt)
   realMinusNotBounded:
     ∀ x · x ∈ RRealMinus ⇒ (∃ x2 · x2 ∈ RRealMinus ∧ x2 ↦ x ∈ lt)
   realPlusOrder:
```

```
  ∀ a, b · a ∈ RRealPlus ∧ b ∈ RReal ∧ (a ↦ b ∈ leq) ⇒ b ∈ RRealPlus
realMinusOrder:
  ∀ a, b · a ∈ RRealMinus ∧ b ∈ RReal ∧ (b ↦ a ∈ leq) ⇒ b ∈ RRealMinus
plusCompatibility:
  ∀ a, b, c · a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧ (a ↦ b ∈ leq) ∧ (a ↦
      c ∈ leq) ⇒
    (a ↦ plus(b ↦ c) ∈ leq)
plusCompatibility2:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    plus(a ↦ b) ↦ c ∈ leq ∧ Rzero ↦ b ∈ leq ⇒
      a ↦ c ∈ leq
extendedTransitivity:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      a ↦ c ∈ lt
extendedTransitivity2:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      a ↦ c ∈ lt
extendedTransitivity3:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      a ↦ c ∈ lt
timesCompatibiliyy:
  ⊤
PROOF RULES
  plusAbelianGroup:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
    plus_commutativity: plus(a ↦ b)
      rhs1: ⊤ ⇒ plus(b ↦ a)
    plus_neutralityL: plus(a ↦ Rzero)
      rhs1: ⊤ ⇒ a
    plus_neutralityR: plus(Rzero ↦ a)
      rhs1: ⊤ ⇒ a
    plus_associativityL: plus(a ↦ plus(b ↦ c))
      rhs1: ⊤ ⇒ plus(plus(a ↦ b) ↦ c)
    plus_associativityR: plus(plus(a ↦ b) ↦ c)
      rhs1: ⊤ ⇒ plus(a ↦ plus(b ↦ c))
    minus_rewrite: minus(a ↦ b)
      rhs1: ⊤ ⇒ plus(a ↦ uminus(b))
    minus_uminus: uminus(minus(a ↦ b))
      rhs1: ⊤ ⇒ minus(b ↦ a)
    minus_uminus_reverse: minus(a ↦ b)
      rhs1: ⊤ ⇒ uminus(minus(b ↦ a))
    uminus_involutive: uminus(uminus(a))
      rhs1: ⊤ ⇒ a
    uminus_reductionL: plus(a ↦ uminus(a))
      rhs1: ⊤ ⇒ Rzero
    uminus_reductionR: plus(uminus(a) ↦ a)
      rhs1: ⊤ ⇒ Rzero
    minus_reduction: minus(a ↦ a)
      rhs1: ⊤ ⇒ Rzero
    zero_minus_left: minus(Rzero ↦ a)
      rhs1: ⊤ ⇒ uminus(a)
    zero_minus_right: minus(a ↦ Rzero)
      rhs1: ⊤ ⇒ a
    uminus_neutral: uminus(Rzero)
      rhs1: ⊤ ⇒ Rzero
  timesCommutativeMonoid:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
    times_commutativity: times(a ↦ b)
      rhs1: ⊤ ⇒ times(b ↦ a)
    times_neutralityL: times(a ↦ Rone)
      rhs1: ⊤ ⇒ a
    times_neutralityR: times(Rone ↦ a)
      rhs1: ⊤ ⇒ a
    times_associativityL: times(a ↦ times(b ↦ c))
      rhs1: ⊤ ⇒ times(times(a ↦ b) ↦ c)
    times_associativityR: times(times(a ↦ b) ↦ c)
      rhs1: ⊤ ⇒ times(a ↦ times(b ↦ c))
    times_absorbingL: times(a ↦ Rzero)
      rhs1: ⊤ ⇒ Rzero
    times_absorbingR: times(Rzero ↦ a)
      rhs1: ⊤ ⇒ Rzero
    division_rewrite: divide(a ↦ b)
      rhs1: ⊤ ⇒ times(a ↦ inverse(b))
    divide_inverse: inverse(divide(a ↦ b))
      rhs1: a ≠ Rzero ⇒ divide(b ↦ a)
    divide_inverse_reverse: divide(a ↦ b)
      rhs1: a ≠ Rzero ⇒ inverse(divide(b ↦ a))
    inverse_involutive: inverse(inverse(a))
      rhs1: ⊤ ⇒ a
    inverse_reductionL: times(a ↦ inverse(a))
      rhs1: ⊤ ⇒ Rone
    inverse_reductionR: times(inverse(a) ↦ a)
      rhs1: ⊤ ⇒ Rone
    divide_reduction: divide(a ↦ a)
      rhs1: ⊤ ⇒ Rone
    one_divide_left: divide(Rone ↦ a)
      rhs1: ⊤ ⇒ inverse(a)
    one_divide_right: divide(a ↦ Rone)
      rhs1: ⊤ ⇒ a
    inverse_neutral: inverse(Rone)
      rhs1: ⊤ ⇒ Rone
  plusTimesDistributive:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
    left_distribute: times(a ↦ plus(b ↦ c))
      rhs1: ⊤ ⇒ plus(times(a ↦ b) ↦ times(a ↦ c))
    right_distribute: times(plus(b ↦ c) ↦ a)
      rhs1: ⊤ ⇒ plus(times(b ↦ a) ↦ times(c ↦ a))
    left_factorize: plus(times(a ↦ b) ↦ times(a ↦ c))
      rhs1: ⊤ ⇒ times(a ↦ plus(b ↦ c))
    right_factorize: plus(times(b ↦ a) ↦ times(c ↦ a))
      rhs1: ⊤ ⇒ times(plus(b ↦ c) ↦ a)
    left_minus_distribute: times(a ↦ minus(b ↦ c))
      rhs1: ⊤ ⇒ minus(times(a ↦ b) ↦ times(a ↦ c))
    right_minus_distribute: times(minus(b ↦ c) ↦ a)
      rhs1: ⊤ ⇒ minus(times(b ↦ a) ↦ times(c ↦ a))
    left_minus_factorize: minus(times(a ↦ b) ↦ times(a ↦ c))
      rhs1: ⊤ ⇒ times(a ↦ minus(b ↦ c))
    right_minus_factorize: minus(times(b ↦ a) ↦ times(c ↦ a))
      rhs1: ⊤ ⇒ times(minus(b ↦ c) ↦ a)
    minus_distribute: uminus(plus(a ↦ b))
      rhs1: ⊤ ⇒ plus(uminus(a) ↦ uminus(b))
    minus_factorize: plus(uminus(a) ↦ uminus(b))
      rhs1: ⊤ ⇒ uminus(plus(a ↦ b))
    divide_distribute: divide(plus(b ↦ c) ↦ a)
      rhs1: ⊤ ⇒ plus(divide(b ↦ a) ↦ divide(c ↦ a))
    divide_factorize: plus(divide(b ↦ a) ↦ divide(c ↦ a))
      rhs1: ⊤ ⇒ divide(plus(b ↦ c) ↦ a)
    inverse_uminus: inverse(uminus(a))
      rhs1: a ≠ Rzero ⇒ uminus(inverse(a))
    uminus_inverse: uminus(inverse(a))
      rhs1: a ≠ Rzero ⇒ inverse(uminus(a))
    divide_left_absorbing: divide(Rzero ↦ a)
      rhs1: ⊤ ⇒ Rzero
  equations:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
    plus_left_simplify: plus(a ↦ b) = plus(a ↦ c)
      rhs1: ⊤ ⇒ b = c
    plus_right_simplify: plus(b ↦ a) = plus(c ↦ a)
      rhs1: ⊤ ⇒ b = c
    times_left_simplify: times(a ↦ b) = times(a ↦ c)
      rhs1: a ≠ Rzero ⇒ b = c
    times_right_simplify: times(b ↦ a) = times(c ↦ a)
      rhs1: a ≠ Rzero ⇒ b = c
    minus_left_simplify: minus(a ↦ b) = minus(a ↦ c)
      rhs1: ⊤ ⇒ b = c
    minus_right_simplify: minus(b ↦ a) = minus(c ↦ a)
      rhs1: ⊤ ⇒ b = c
    divide_left_simplify: divide(a ↦ b) = divide(a ↦ c)
      rhs1: a ≠ Rzero ⇒ b = c
    divide_right_simplify: divide(b ↦ a) = divide(c ↦ a)
      rhs1: ⊤ ⇒ b = c
    uminus_simplify: uminus(a) = uminus(b)
      rhs1: ⊤ ⇒ a = b
    inverse_simplify: inverse(a) = inverse(b)
      rhs1: ⊤ ⇒ a = b
    uminus_unsimplify: a = b
      rhs1: ⊤ ⇒ uminus(a) = uminus(b)
    inverse_unsimplify: a = b
      rhs1: a ≠ Rzero ∧ b ≠ Rzero ⇒ inverse(a) = inverse(b)
    equal_to_minus: a = b
      rhs1: ⊤ ⇒ minus(a ↦ b) = Rzero
    equal_to_divide: a = b
      rhs1: b ≠ Rzero ⇒ divide(a ↦ b) = Rone
    plus_eq_zero: plus(a ↦ b) = Rzero
      rhs1: ⊤ ⇒ a = uminus(b)
    minus_eq_zero: minus(a ↦ b) = Rzero
      rhs1: ⊤ ⇒ a = b
    times_eq_one: times(a ↦ b) = Rone
      rhs1: b ≠ Rzero ⇒ a = inverse(b)
    divide_eq_one: divide(a ↦ b) = Rone
      rhs1: ⊤ ⇒ a = b
    times_integral: times(a ↦ b) = Rzero
      rhs1: a ≠ Rzero ⇒ b = Rzero
      rhs2: b ≠ Rzero ⇒ a = Rzero
      rhs3: ⊤ ⇒ a = Rzero ∨ b = Rzero
    divide_integral: divide(a ↦ b) = Rzero
      rhs1: ⊤ ⇒ a = Rzero
    uminus_eq_rewrite: uminus(a) = b
      rhs1: ⊤ ⇒ a = uminus(b)
    inverse_eq_rewrite: inverse(a) = b
      rhs1: b ≠ Rzero ⇒ a = inverse(b)
  inequationsGeneral:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
```

```
    leq_inversion: a ↦ b ∈ leq
      rhs1: ⊤ ⇒ b ↦ a ∈ geq
    geq_inversion: a ↦ b ∈ geq
      rhs1: ⊤ ⇒ b ↦ a ∈ leq
    lt_inversion: a ↦ b ∈ lt
      rhs1: ⊤ ⇒ b ↦ a ∈ gt
    gt_inversion: a ↦ b ∈ gt
      rhs1: ⊤ ⇒ b ↦ a ∈ lt
    leq_total: ¬(a ↦ b ∈ leq)
      rhs1: ⊤ ⇒ a ↦ b ∈ gt
    geq_total: ¬(a ↦ b ∈ geq)
      rhs1: ⊤ ⇒ a ↦ b ∈ lt
    geq_total_reverse: ¬(a ↦ b ∈ lt)
      rhs1: ⊤ ⇒ a ↦ b ∈ geq
    leq_total_reverse: ¬(a ↦ b ∈ gt)
      rhs1: ⊤ ⇒ a ↦ b ∈ leq
    eq_to_leq_weakening: a = b
      rhs1: ⊤ ⇒ a ↦ b ∈ leq
    eq_to_geq_weakening: a = b
      rhs1: ⊤ ⇒ a ↦ b ∈ geq
    noteq_rewrite: a ≠ b
      rhs1: ⊤ ⇒ (a ↦ b ∈ lt) ∨ (b ↦ a ∈ lt)
    lt_to_leq_weakening: a ↦ b ∈ lt
      rhs1: ⊤ ⇒ (a ↦ b ∈ leq) ∧ a ≠ b
    gt_to_geq_weakening: a ↦ b ∈ gt
      rhs1: ⊤ ⇒ (a ↦ b ∈ geq) ∧ a ≠ b
    leq_reflexive: a ↦ a ∈ leq
      rhs1: ⊤ ⇒ ⊤
    geq_reflexive: a ↦ a ∈ geq
      rhs1: ⊤ ⇒ ⊤
  Inference Rules
    leq_transitivity: a ↦ b ∈ leq,b ↦ c ∈ leq ⊢ a ↦ c ∈ leq
    lt_transitivity: a ↦ b ∈ lt,b ↦ c ∈ lt ⊢ a ↦ c ∈ lt
    geq_transitivity: a ↦ b ∈ geq,b ↦ c ∈ geq ⊢ a ↦ c ∈ geq
    gt_transitivity: a ↦ b ∈ gt,b ↦ c ∈ gt ⊢ a ↦ c ∈ gt
    leq_antismetry: a ↦ b ∈ leq,b ↦ a ∈ leq ⊢ a = b
    geq_antisymetry: a ↦ b ∈ geq,b ↦ a ∈ geq ⊢ a = b
    leq_lt_weakening: a ↦ b ∈ leq,a ≠ b ⊢ a ↦ b ∈ lt
    geq_gt_weakening: a ↦ b ∈ geq,a ≠ b ⊢ a ↦ b ∈ gt
    lt_leq_reduction: (a ↦ b ∈ lt) ∨ a = b ⊢ a ↦ b ∈ leq
    gt_geq_reduction: (a ↦ b ∈ gt) ∨ a = b ⊢ a ↦ b ∈ geq
  inequationsCompatibility:
  Metavariables
    a: RReal
    b: RReal
    c: RReal
  Rewrite Rules
    leq_plus_compatibility: plus(a ↦ b) ↦ plus(a ↦ c) ∈ leq
      rhs1: ⊤ ⇒ b ↦ c ∈ leq
    lt_plus_compatibility: plus(a ↦ b) ↦ plus(a ↦ c) ∈ lt
    rhs1: ⊤ ⇒ b ↦ c ∈ lt
  uminus_leq_ineq: a ↦ b ∈ leq
    rhs1: ⊤ ⇒ uminus(b) ↦ uminus(a) ∈ leq
  uminus_lt_ineq: a ↦ b ∈ lt
    rhs1: ⊤ ⇒ uminus(b) ↦ uminus(a) ∈ lt
  leq_times_compatibility: times(a ↦ b) ↦ times(a ↦ c) ∈ leq
    rhs1: Rzero ↦ a ∈ lt ⇒ b ↦ c ∈ leq
    rhs2: a ↦ Rzero ∈ lt ⇒ c ↦ b ∈ leq
  lt_times_compatibility: times(a ↦ b) ↦ times(a ↦ c) ∈ lt
    rhs1: Rzero ↦ a ∈ lt ⇒ b ↦ c ∈ lt
    rhs2: a ↦ Rzero ∈ lt ⇒ c ↦ b ∈ lt
  inverse_leq_ineq: a ↦ b ∈ leq
    rhs1: a ≠ Rzero ∧ b ≠ Rzero ⇒ inverse(b) ↦ inverse(a) ∈ leq
  inverse_lt_ineq: a ↦ b ∈ lt
    rhs1: a ≠ Rzero ∧ b ≠ Rzero ⇒ inverse(b) ↦ inverse(a) ∈ lt
  leq_balance_left: a ↦ b ∈ leq
    rhs1: ⊤ ⇒ minus(a ↦ b) ↦ Rzero ∈ leq
  leq_balance_right: a ↦ b ∈ leq
    rhs1: ⊤ ⇒ Rzero ↦ minus(b ↦ a) ∈ leq
  lt_balance_left: a ↦ b ∈ lt
    rhs1: ⊤ ⇒ minus(a ↦ b) ↦ Rzero ∈ lt
  lt_balance_right: a ↦ b ∈ lt
    rhs1: ⊤ ⇒ Rzero ↦ minus(b ↦ a) ∈ lt
  leq_div_balance_left: a ↦ b ∈ leq
    rhs1: b ≠ Rzero ⇒ divide(a ↦ b) ↦ Rone ∈ leq
  leq_div_balance_right: a ↦ b ∈ leq
    rhs1: a ≠ Rzero ⇒ Rone ↦ divide(b ↦ a) ∈ leq
  lt_div_balance_left: a ↦ b ∈ lt
    rhs1: b ≠ Rzero ⇒ divide(a ↦ b) ↦ Rone ∈ lt
  lt_div_balance_right: a ↦ b ∈ lt
    rhs1: a ≠ Rzero ⇒ Rone ↦ divide(b ↦ a) ∈ lt
  leq_plus_balance_left: a ↦ plus(b ↦ c) ∈ leq
    rhs1: ⊤ ⇒ minus(a ↦ c) ↦ b ∈ leq
  leq_plus_balance_right: plus(a ↦ b) ↦ c ∈ leq
    rhs1: ⊤ ⇒ a ↦ minus(c ↦ b) ∈ leq
  leq_times_balance_left: a ↦ times(b ↦ c) ∈ leq
    rhs1: Rzero ↦ c ∈ lt ⇒ divide(a ↦ c) ↦ b ∈ leq
    rhs2: c ↦ Rzero ∈ lt ⇒ b ↦ divide(a ↦ c) ∈ leq
  leq_times_balance_right: times(a ↦ b) ↦ c ∈ leq
    rhs1: Rzero ↦ b ∈ lt ⇒ a ↦ divide(c ↦ b) ∈ leq
    rhs2: b ↦ Rzero ∈ lt ⇒ divide(c ↦ b) ↦ a ∈ leq
  lt_plus_balance_left: a ↦ plus(b ↦ c) ∈ lt
    rhs1: ⊤ ⇒ minus(a ↦ c) ↦ b ∈ lt
  lt_plus_balance_right: plus(a ↦ b) ↦ c ∈ lt
    rhs1: ⊤ ⇒ a ↦ minus(b ↦ c) ∈ lt
  lt_times_balance_left: a ↦ times(b ↦ c) ∈ lt
    rhs1: Rzero ↦ c ∈ lt ⇒ divide(a ↦ c) ↦ b ∈ lt
    rhs2: c ↦ Rzero ∈ lt ⇒ b ↦ divide(a ↦ c) ∈ lt
  lt_times_balance_right: times(a ↦ b) ↦ c ∈ lt
    rhs1: Rzero ↦ b ∈ lt ⇒ a ↦ divide(c ↦ b) ∈ lt
```

```
      rhs2: b ↦ Rzero ∈ lt ⇒ divide(c ↦ b) ↦ a ∈ lt
    inverse_positive_leq: Rzero ↦ a ∈ leq
      rhs1: a ≠ Rzero ⇒ Rzero ↦ inverse(a) ∈ lt
    inverse_negative_leq: a ↦ Rzero ∈ leq
      rhs1: a ≠ Rzero ⇒ inverse(a) ↦ Rzero ∈ lt
    inverse_positive_lt: Rzero ↦ a ∈ lt
      rhs1: ⊤ ⇒ Rzero ↦ inverse(a) ∈ lt
    inverse_negative_lt: a ↦ Rzero ∈ lt
      rhs1: ⊤ ⇒ inverse(a) ↦ Rzero ∈ lt
    times_compatibility_leq: Rzero ↦ times(a ↦ b) ∈ leq
      rhs5: Rzero ↦ a ∈ leq ⇒ Rzero ↦ b ∈ leq
      rhs6: Rzero ↦ b ∈ leq ⇒ Rzero ↦ a ∈ leq
      rhs7: a ↦ Rzero ∈ leq ⇒ b ↦ Rzero ∈ leq
      rhs8: b ↦ Rzero ∈ leq ⇒ a ↦ Rzero ∈ leq
    times_compatibility_geq: Rzero ↦ times(a ↦ b) ∈ geq
      rhs5: ⊤ ⇒ (a ↦ Rzero ∈ leq ∧ Rzero ↦ b ∈ leq) ∨ (Rzero ↦ a ∈ leq
          ∧ b ↦ Rzero ∈ leq)
  absPredicates:
  Metavariables
    x: RReal
    y: RReal
  Rewrite Rules
    abs_eqn: abs(x) = y
      rhs1: ⊤ ⇒ x = y ∨ uminus(x) = y
    abs_contradict: abs(x) = y
      rhs1: y ↦ Rzero ∈ lt ⇒ ⊥
    abs_zero: abs(x) = Rzero
      rhs1: ⊤ ⇒ x = Rzero
  realSubSetsUtility:
  Metavariables
    a: RReal
  Rewrite Rules
    realPlus_to_leq: a ∈ RRealPlus
      rhs1: ⊤ ⇒ Rzero ↦ a ∈ leq
    realMinus_to_leq: a ∈ RRealMinus
      rhs1: ⊤ ⇒ a ↦ Rzero ∈ leq
    realStar_to_neq: a ∈ RRealStar
      rhs1: ⊤ ⇒ a ≠ Rzero
    realPlusStar_to_leq: a ∈ RRealPlusStar
      rhs1: ⊤ ⇒ Rzero ↦ a ∈ lt
    realMinusStar_to_leq: a ∈ RRealMinusStar
      rhs1: ⊤ ⇒ a ↦ Rzero ∈ lt
    zero_in_RRealPlus: Rzero ∈ RRealPlus
      rhs1: ⊤ ⇒ ⊤
  typing:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    plusTypeRew: a ↦ b ∈ dom(plus)
```

```
      rhs1: ⊤ ⇒ a ∈ RReal ∧ b ∈ RReal
    minusTypeRew: a ↦ b ∈ dom(minus)
      rhs1: ⊤ ⇒ a ∈ RReal ∧ b ∈ RReal
    timesTypeRew: a ↦ b ∈ dom(times)
      rhs1: ⊤ ⇒ a ∈ RReal ∧ b ∈ RReal
    divideTypeRew: a ↦ b ∈ dom(divide)
      rhs1: ⊤ ⇒ a ∈ RReal ∧ b ∈ RRealStar
    uminusTypeRew: a ∈ dom(uminus)
      rhs1: ⊤ ⇒ a ∈ RReal
    inverseTypeRew: a ∈ dom(inverse)
      rhs1: ⊤ ⇒ a ∈ RRealStar
    absTypeRew: a ∈ dom(abs)
      rhs1: ⊤ ⇒ a ∈ RReal
    sqrtTypeRew: a ∈ dom(sqrt)
      rhs1: ⊤ ⇒ a ∈ RRealPlus
    plusBasicTypeRew: plus ∈ RReal×RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    minusBasicTypeRew: minus ∈ RReal×RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    timesBasicTypeRew: times ∈ RReal×RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    divideBasicTypeRew: divide ∈ RReal×RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    uminusBasicTypeRew: uminus ∈ RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    inverseBasicTypeRew: inverse ∈ RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    absBasicTypeRew: abs ∈ RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
    sqrtBasicTypeRew: sqrt ∈ RReal ⇸ RReal
      rhs1: ⊤ ⇒ ⊤
  Inference Rules
    plusType: a ∈ RReal,b ∈ RReal ⊢ plus(a ↦ b) ∈ RReal
    minusType: a ∈ RReal,b ∈ RReal ⊢ minus(a ↦ b) ∈ RReal
    timesType: a ∈ RReal,b ∈ RReal ⊢ times(a ↦ b) ∈ RReal
    divideType: a ∈ RReal,b ∈ RReal,b ≠ Rzero ⊢ divide(a ↦ b) ∈ RRealStar
    uminusType: a ∈ RReal ⊢ uminus(a) ∈ RReal
    inverseType: a ∈ RRealStar ⊢ inverse(a) ∈ RRealStar
    sqrtType: a ∈ RRealPlus ⊢ sqrt(a) ∈ RRealPlus
    absType: a ∈ RReal ⊢ abs(a) ∈ RRealPlus
    plusTypeArg: a ∈ RReal,b ∈ RReal ⊢ a ↦ b ∈ dom(plus)
    minusTypeArg: a ∈ RReal,b ∈ RReal ⊢ a ↦ b ∈ dom(minus)
    timesTypeArg: a ∈ RReal,b ∈ RReal ⊢ a ↦ b ∈ dom(times)
    divideTypeArg: a ∈ RReal,b ∈ RReal,b ≠ Rzero ⊢ a ↦ b ∈ dom(divide)
    uminusTypeArg: a ∈ RReal ⊢ a ∈ dom(uminus)
    inverseTypeArg: a ∈ RRealStar ⊢ a ∈ dom(inverse)
    sqrtTypeArg: a ∈ RRealPlus ⊢ a ∈ dom(sqrt)
    absTypeArg: a ∈ RReal ⊢ a ∈ dom(abs)
END
```

Theory of Intervals

Listings A.6: Intervals theory

```
THEORY
  IMPORT THEORY Reals
  OPERATORS
    Infinity2Open expression (b: RReal)
      direct definition
        { t | t ↦ b ∈ lt }
    Infinity2Closed expression (b: RReal)
      direct definition
        { t | t ↦ b ∈ leq }
    Open2Infinity expression (a: RReal)
      direct definition
        { t | a ↦ t ∈ lt }
    Closed2Infinity expression (a: RReal)
      direct definition
        { t | a ↦ t ∈ leq }
    Open2Open expression (a: RReal,b: RReal)
      direct definition
        { t | a ↦ t ∈ lt ∧ t ↦ b ∈ lt }
    Open2Closed expression (a: RReal,b: RReal)
      direct definition
        { t | a ↦ t ∈ lt ∧ t ↦ b ∈ leq }
    Closed2Open expression (a: RReal,b: RReal)
      direct definition
        { t | a ↦ t ∈ leq ∧ t ↦ b ∈ lt }
    Closed2Closed expression (a: RReal,b: RReal)
      direct definition
        { t | a ↦ t ∈ leq ∧ t ↦ b ∈ leq }
  THEOREMS
    realPlusIsZero2Infinity:
      RRealPlus = Closed2Infinity(Rzero)
    realMinusIsInfinity2Zero:
      RRealMinus = Infinity2Closed(Rzero)
    c2c_existence:
      ∀ a,b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒ (∃ x · x ∈ Closed2Closed
          (a,b))
    c2o_existence:
      ∀ a,b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒ (∃ x · x ∈ Closed2Open(a,
          b))
    o2c_existence:
      ∀ a,b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒ (∃ x · x ∈ Open2Closed(a,
          b))
    o2o_existence:
      ∀ a,b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒ (∃ x · x ∈ Open2Open(a,b)
          )
    boundaryInClosed2Closed:
```

```
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ⇒ (a ∈ Closed2Closed(a,b) ∧ b ∈
      Closed2Closed(a,b))
boundaryInClosed2Open:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ⇒ (a ∈ Closed2Open(a,b))
boundaryInOpen2Closed:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ⇒ (b ∈ Open2Closed(a,b))
boundaryInInfinity2Closed:
  ∀ b · b ∈ RReal ⇒ (b ∈ Infinity2Closed(b))
boundaryInClosed2Infinity:
  ∀ a · a ∈ RReal ⇒ (a ∈ Closed2Infinity(a))
closed2ClosedLowerBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    lowerBound(leq, Closed2Closed(a,b), a)
closed2OpenLowerBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    lowerBound(leq, Closed2Open(a,b), a)
open2ClosedLowerBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    lowerBound(leq, Open2Closed(a,b), a)
open2OpenLowerBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    lowerBound(leq, Open2Open(a,b), a)
closed2InfinityLowerBound:
  ∀ a · a ∈ RReal ⇒
    lowerBound(leq, Closed2Infinity(a), a)
open2InfinityLowerBound:
  ∀ a · a ∈ RReal ⇒
    lowerBound(leq, Open2Infinity(a), a)
infinity2ClosedLowerBound:
  ∀ b · b ∈ RReal ⇒
    ¬ lowerBounded(leq, Infinity2Closed(b))
infinity2OpenLowerBound:
  ∀ b · b ∈ RReal ⇒
    ¬ lowerBounded(leq, Infinity2Open(b))
closed2ClosedInfimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    infimum(leq, Closed2Closed(a,b), a)
closed2OpenInfimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    infimum(leq, Closed2Open(a,b), a)
open2ClosedInfimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    infimum(leq, Open2Closed(a,b), a)
open2OpenInfimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    infimum(leq, Open2Open(a,b), a)
closed2InfinityInfimum:
  ∀ a · a ∈ RReal ⇒
    infimum(leq, Closed2Infinity(a), a)
open2InfinityInfimum:
  ∀ a · a ∈ RReal ⇒
    infimum(leq, Open2Infinity(a), a)
closed2ClosedMinimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    minimum(leq, Closed2Closed(a,b), a)
closed2OpenMinimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    minimum(leq, Closed2Open(a,b), a)
open2ClosedMinimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    ¬ hasMinimum(leq, Open2Closed(a,b))
open2OpenMinimum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    ¬ hasMinimum(leq, Open2Open(a,b))
closed2InfinityMinimum:
  ∀ a · a ∈ RReal ⇒
    minimum(leq, Closed2Infinity(a), a)
open2InfinityMinimum:
  ∀ a · a ∈ RReal ⇒
    ¬ hasMinimum(leq, Open2Infinity(a))
infinity2ClosedMinimum:
  ∀ b · b ∈ RReal ⇒
    ¬ hasMinimum(leq, Infinity2Closed(b))
infinity2OpenMinimum:
  ∀ b · b ∈ RReal ⇒
    ¬ hasMinimum(leq, Infinity2Open(b))
closed2ClosedUpperBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    upperBound(leq, Closed2Closed(a,b), b)
closed2OpenUpperBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    upperBound(leq, Closed2Open(a,b), b)
open2ClosedUpperBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    upperBound(leq, Open2Closed(a,b), b)
open2OpenUpperBound:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    upperBound(leq, Open2Open(a,b), b)
closed2InfinityUpperBound:
  ∀ a · a ∈ RReal ⇒
    ¬ upperBounded(leq, Closed2Infinity(a))
open2InfinityUpperBound:
  ∀ a · a ∈ RReal ⇒
    ¬ upperBounded(leq, Open2Infinity(a))
infinity2ClosedUpperBound:
  ∀ b · b ∈ RReal ⇒
    upperBound(leq, Infinity2Closed(b), b)
infinity2OpenUpperBound:
  ∀ b · b ∈ RReal ⇒
    upperBound(leq, Infinity2Open(b), b)
```

```
closed2ClosedSupremum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    supremum(leq, Closed2Closed(a,b), b)
closed2OpenSupremum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    supremum(leq, Closed2Open(a,b), b)
open2ClosedSupremum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    supremum(leq, Open2Closed(a,b), b)
open2OpenSupremum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    supremum(leq, Open2Open(a,b), b)
infinity2ClosedSupremum:
  ∀ b · b ∈ RReal ⇒
    supremum(leq, Infinity2Closed(b), b)
infinity2OpenSupremum:
  ∀ b · b ∈ RReal ⇒
    supremum(leq, Open2Infinity(b), b)
closed2ClosedMaximum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ leq ⇒
    maximum(leq, Closed2Closed(a,b), b)
closed2OpenMaximum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    ¬ hasMaximum(leq, Closed2Open(a,b))
open2ClosedMaximum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    maximum(leq, Open2Closed(a,b), b)
open2OpenMaximum:
  ∀ a, b · a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ⇒
    ¬ hasMaximum(leq, Open2Open(a,b))
closed2InfinityMaximum:
  ∀ a · a ∈ RReal ⇒
    ¬ hasMaximum(leq, Closed2Infinity(a))
open2InfinityMaximum:
  ∀ a · a ∈ RReal ⇒
    ¬ hasMaximum(leq, Open2Infinity(a))
infinity2ClosedMaximum:
  ∀ b · b ∈ RReal ⇒
    maximum(leq, Infinity2Closed(b), b)
infinity2OpenMaximum:
  ∀ b · b ∈ RReal ⇒
    ¬ hasMaximum(leq, Infinity2Open(b))
c2cUc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ leq ⇒
      Closed2Closed(a,b) ∪ Closed2Closed(b,c) = Closed2Closed(a,c)
c2cUc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∪ Closed2Open(b,c) = Closed2Open(a,c)
o2cUc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Open2Closed(a,b) ∪ Closed2Closed(b,c) = Open2Closed(a,c)
o2cUc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Closed(a,b) ∪ Closed2Open(b,c) = Open2Open(a,c)
inf2cUc2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ leq ⇒
      Infinity2Closed(b) ∪ Closed2Closed(b,c) = Infinity2Closed(c)
inf2cUc2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∪ Closed2Open(b,c) = Infinity2Open(c)
c2cUc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ leq ⇒
      Closed2Closed(a,b) ∪ Closed2Infinity(b) = Closed2Infinity(a)
o2cUc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Open2Closed(a,b) ∪ Closed2Infinity(b) = Open2Infinity(a)
inf2cUc2inf:
  ∀ a ·
    a ∈ RReal ⇒
      Infinity2Closed(a) ∪ Closed2Infinity(a) = RReal
c2cUo2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∪ Open2Closed(b,c) = Closed2Closed(a,c)
c2cUo2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∪ Open2Open(b,c) = Closed2Open(a,c)
o2cUo2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
```

```
      Open2Closed(a,b) ∪ Open2Closed(b,c) = Open2Closed(a,c)
o2cUo2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Closed(a,b) ∪ Open2Open(b,c) = Open2Open(a,c)
inf2cUo2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∪ Open2Closed(b,c) = Infinity2Closed(c)
inf2cUo2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∪ Open2Open(b,c) = Infinity2Open(c)
c2cUo2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ leq ⇒
      Closed2Closed(a,b) ∪ Open2Infinity(b) = Closed2Infinity(a)
o2cUo2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Open2Closed(a,b) ∪ Open2Infinity(b) = Open2Infinity(a)
inf2cUo2inf:
  ∀ a ·
    a ∈ RReal ⇒
      Infinity2Closed(a) ∪ Open2Infinity(a) = RReal
c2oUc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Closed2Open(a,b) ∪ Closed2Closed(b,c) = Closed2Closed(a,c)
c2oUc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Closed2Open(a,b) ∪ Closed2Open(b,c) = Closed2Open(a,c)
o2oUc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Open2Open(a,b) ∪ Closed2Closed(b,c) = Open2Closed(a,c)
o2oUc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Open(a,b) ∪ Closed2Open(b,c) = Open2Open(a,c)
```

```
inf2oUc2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ leq ⇒
      Infinity2Open(b) ∪ Closed2Closed(b,c) = Infinity2Closed(c)
inf2oUc2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Open(b) ∪ Closed2Open(b,c) = Infinity2Open(c)
c2oUc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Closed2Open(a,b) ∪ Closed2Infinity(b) = Closed2Infinity(a)
o2oUc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Open2Open(a,b) ∪ Closed2Infinity(b) = Open2Infinity(a)
inf2oUc2inf:
  ∀ a ·
    a ∈ RReal ⇒
      Infinity2Open(a) ∪ Closed2Infinity(a) = RReal
c2cCc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ leq ⇒
      Closed2Closed(a,b) ∩ Closed2Closed(b,c) = {b}
o2cCc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Open2Closed(a,b) ∩ Closed2Closed(b,c) = {b}
c2cCc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∩ Closed2Open(b,c) = {b}
o2cCc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Closed(a,b) ∩ Closed2Open(b,c) = {b}
inf2cCc2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ leq ⇒
      Infinity2Closed(b) ∩ Closed2Closed(b,c) = {b}
inf2cCc2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∩ Closed2Open(b,c) = {b}
c2cCc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ leq ⇒
      Closed2Closed(a,b) ∩ Closed2Infinity(b) = {b}
o2cCc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Open2Closed(a,b) ∩ Closed2Infinity(b) = {b}
inf2cCc2inf:
  ∀ a ·
    a ∈ RReal ⇒
      Infinity2Closed(a) ∩ Closed2Infinity(a) = {a}
c2oCc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Closed2Open(a,b) ∩ Closed2Closed(b,c) = ∅
o2oCc2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ leq ⇒
      Open2Open(a,b) ∩ Closed2Closed(b,c) = ∅
c2oCc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Closed2Open(a,b) ∩ Closed2Open(b,c) = ∅
o2oCc2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Open(a,b) ∩ Closed2Open(b,c) = ∅
inf2oCc2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ leq ⇒
      Infinity2Open(b) ∩ Closed2Closed(b,c) = ∅
inf2oCc2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Open(b) ∩ Closed2Open(b,c) = ∅
c2oCc2inf:
  ∀ a, b ·
```

```
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Closed2Open(a,b) ∩ Closed2Infinity(b) = ∅
o2oCc2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ lt ⇒
      Open2Open(a,b) ∩ Closed2Infinity(b) = ∅
inf2oCc2inf:
  ∀ a ·
    a ∈ RReal ⇒
      Infinity2Closed(a) ∩ Closed2Infinity(a) = ∅
c2cCo2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∩ Open2Closed(b,c) = ∅
o2cCo2c:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Closed(a,b) ∩ Open2Closed(b,c) = ∅
c2cCo2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ leq ∧ b ↦ c ∈ lt ⇒
      Closed2Closed(a,b) ∩ Open2Open(b,c) = ∅
o2cCo2o:
  ∀ a, b, c ·
    a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
    a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
      Open2Closed(a,b) ∩ Open2Open(b,c) = ∅
inf2cCo2c:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∩ Open2Closed(b,c) = ∅
inf2cCo2o:
  ∀ b, c ·
    b ∈ RReal ∧ c ∈ RReal ∧
    b ↦ c ∈ lt ⇒
      Infinity2Closed(b) ∩ Open2Open(b,c) = ∅
c2cCo2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
    a ↦ b ∈ leq ⇒
      Closed2Closed(a,b) ∩ Open2Infinity(b) = ∅
o2cCo2inf:
  ∀ a, b ·
    a ∈ RReal ∧ b ∈ RReal ∧
```

```
      a ↦ b ∈ lt ⇒
        Open2Closed(a,b) ∩ Open2Infinity(b) = ∅
  inf2cCo2inf:
    ∀ a ·
      a ∈ RReal ⇒
        Infinity2Closed(a) ∩ Open2Infinity(a) = ∅
  c2oCo2c:
    ∀ a, b, c ·
      a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
      a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
        Closed2Open(a,b) ∩ Open2Closed(b,c) = ∅
  o2oCo2c:
    ∀ a, b, c ·
      a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
      a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
        Open2Open(a,b) ∩ Open2Closed(b,c) = ∅
  c2oCo2o:
    ∀ a, b, c ·
      a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
      a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
        Closed2Open(a,b) ∩ Open2Open(b,c) = ∅
  o2oCo2o:
    ∀ a, b, c ·
      a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧
      a ↦ b ∈ lt ∧ b ↦ c ∈ lt ⇒
        Open2Open(a,b) ∩ Open2Open(b,c) = ∅
  inf2oCo2c:
    ∀ b, c ·
      b ∈ RReal ∧ c ∈ RReal ∧
      b ↦ c ∈ lt ⇒
        Infinity2Open(b) ∩ Open2Closed(b,c) = ∅
  inf2oCo2o:
    ∀ b, c ·
      b ∈ RReal ∧ c ∈ RReal ∧
      b ↦ c ∈ lt ⇒
        Infinity2Open(b) ∩ Open2Open(b,c) = ∅
  c2oCo2inf:
    ∀ a, b ·
      a ∈ RReal ∧ b ∈ RReal ∧
      a ↦ b ∈ lt ⇒
        Closed2Open(a,b) ∩ Open2Infinity(b) = ∅
  o2oCo2inf:
    ∀ a, b ·
      a ∈ RReal ∧ b ∈ RReal ∧
      a ↦ b ∈ lt ⇒
        Open2Open(a,b) ∩ Open2Infinity(b) = ∅
  inf2oCo2inf:
    ∀ a ·
      a ∈ RReal ⇒
        Infinity2Open(a) ∩ Open2Infinity(a) = ∅
```

```
PROOF RULES
  minimumRew:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    minClosed2Closed: Rmin(Closed2Closed(a,b))
      rhs1: a ↦ b ∈ leq ⇒ a
    minClosed2Open: Rmin(Closed2Open(a,b))
      rhs1: a ↦ b ∈ lt ⇒ a
    minClosed2Infinity: Rmin(Closed2Infinity(a))
      rhs1: ⊤ ⇒ a
  maximumRew:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    maxClosed2Closed: Rmax(Closed2Closed(a,b))
      rhs1: a ↦ b ∈ leq ⇒ b
    maxOpen2Closed: Rmax(Open2Closed(a,b))
      rhs1: a ↦ b ∈ lt ⇒ b
    maxInfinity2Closed: Rmax(Infinity2Closed(b))
      rhs1: ⊤ ⇒ b
  infimumRew:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    infClosed2Closed: Rinf(Closed2Closed(a,b))
      rhs1: a ↦ b ∈ leq ⇒ a
    infClosed2Open: Rinf(Closed2Open(a,b))
      rhs1: a ↦ b ∈ lt ⇒ a
    infOpen2Closed: Rinf(Open2Closed(a,b))
      rhs1: a ↦ b ∈ lt ⇒ a
    infOpen2Open: Rinf(Open2Open(a,b))
      rhs1: a ↦ b ∈ lt ⇒ a
    infClosed2Infinity: Rinf(Closed2Infinity(a))
      rhs1: ⊤ ⇒ a
    infOpen2Infinity: Rinf(Open2Infinity(a))
      rhs1: ⊤ ⇒ a
  supremumRew:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    supClosed2Closed: Rsup(Closed2Closed(a,b))
      rhs1: a ↦ b ∈ leq ⇒ b
    supClosed2Open: Rsup(Closed2Open(a,b))
      rhs1: a ↦ b ∈ lt ⇒ b
    supOpen2Closed: Rsup(Open2Closed(a,b))
    rhs1: a ↦ b ∈ lt ⇒ b
  supOpen2Open: Rsup(Open2Open(a,b))
    rhs1: a ↦ b ∈ lt ⇒ b
  supInfinity2Closed: Rsup(Infinity2Closed(b))
    rhs1: ⊤ ⇒ b
  supInfinity2Open: Rsup(Infinity2Open(b))
    rhs1: ⊤ ⇒ b
intervalURew:
Metavariables
  a: RReal
  b: RReal
  c: RReal
Rewrite Rules
  c2cUc2c_rew: Closed2Closed(a,b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Closed2Closed(a,c)
  o2cUc2c_rew: Open2Closed(a,b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Open2Closed(a,c)
  c2cUc2o_rew: Closed2Closed(a,b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Closed2Open(a,c)
  o2cUc2o_rew: Open2Closed(a,b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Open2Open(a,c)
  inf2cUc2c_rew: Infinity2Closed(b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Infinity2Closed(c)
  inf2cUc2o_rew: Infinity2Closed(b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Infinity2Open(c)
  c2cUc2inf_rew: Closed2Closed(a,b) ∪ Closed2Infinity(b)
    rhs1: ⊤ ⇒ Closed2Infinity(a)
  o2cUc2inf_rew: Open2Closed(a,b) ∪ Closed2Infinity(b)
    rhs1: ⊤ ⇒ Open2Infinity(a)
  c2oUc2c_rew: Closed2Open(a,b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Closed2Closed(a,c)
  o2oUc2c_rew: Open2Open(a,b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Open2Closed(a,c)
  c2oUc2o_rew: Closed2Open(a,b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Closed2Open(a,c)
  o2oUc2o_rew: Open2Open(a,b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Open2Open(a,c)
  inf2oUc2c_rew: Infinity2Open(b) ∪ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ Infinity2Closed(c)
  inf2oUc2o_rew: Infinity2Open(b) ∪ Closed2Open(b,c)
    rhs1: ⊤ ⇒ Infinity2Open(c)
  c2oUc2inf_rew: Closed2Open(a,b) ∪ Closed2Infinity(b)
    rhs1: ⊤ ⇒ Closed2Infinity(a)
  o2oUc2inf_rew: Open2Open(a,b) ∪ Closed2Infinity(b)
    rhs1: ⊤ ⇒ Open2Infinity(a)
  c2cUo2c_rew: Closed2Closed(a,b) ∪ Open2Closed(b,c)
    rhs1: ⊤ ⇒ Closed2Closed(a,c)
  o2cUo2c_rew: Open2Closed(a,b) ∪ Open2Closed(b,c)
    rhs1: ⊤ ⇒ Open2Closed(a,c)
  c2cUo2o_rew: Closed2Closed(a,b) ∪ Open2Open(b,c)
```

```
    rhs1: ⊤ ⇒ Closed2Open(a,c)
  o2cUo2o_rew: Open2Closed(a,b) ∪ Open2Open(b,c)
    rhs1: ⊤ ⇒ Open2Open(a,c)
  inf2cUo2c_rew: Infinity2Closed(b) ∪ Open2Closed(b,c)
    rhs1: ⊤ ⇒ Infinity2Closed(c)
  inf2cUo2o_rew: Infinity2Closed(b) ∪ Open2Open(b,c)
    rhs1: ⊤ ⇒ Infinity2Open(c)
  c2cUo2inf_rew: Closed2Closed(a,b) ∪ Open2Infinity(b)
    rhs1: ⊤ ⇒ Closed2Infinity(a)
  o2cUo2inf_rew: Open2Closed(a,b) ∪ Open2Infinity(b)
    rhs1: ⊤ ⇒ Open2Infinity(a)
intervalCRew:
Metavariables
  a: RReal
  b: RReal
  c: RReal
Rewrite Rules
  c2cCc2c_rew: Closed2Closed(a,b) ∩ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ {b}
  o2cCc2c_rew: Open2Closed(a,b) ∩ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ {b}
  c2cCc2o_rew: Closed2Closed(a,b) ∩ Closed2Open(b,c)
    rhs1: ⊤ ⇒ {b}
  o2cCc2o_rew: Open2Closed(a,b) ∩ Closed2Open(b,c)
    rhs1: ⊤ ⇒ {b}
  inf2cCc2c_rew: Infinity2Closed(b) ∩ Closed2Closed(b,c)
    rhs1: ⊤ ⇒ {b}
  inf2cCc2o_rew: Infinity2Closed(b) ∩ Closed2Open(b,c)
    rhs1: ⊤ ⇒ {b}
  c2cCc2inf_rew: Closed2Closed(a,b) ∩ Closed2Infinity(b)
    rhs1: ⊤ ⇒ {b}
  o2cCc2inf_rew: Open2Closed(a,b) ∩ Closed2Infinity(b)
    rhs1: ⊤ ⇒ {b}
intervalRealParts:
Rewrite Rules
  RPlus2Int: RRealPlus
    rhs1: ⊤ ⇒ Closed2Infinity(Rzero)
  Int2RPlus: Closed2Infinity(Rzero)
    rhs1: ⊤ ⇒ RRealPlus
  RMinus2Int: RRealMinus
    rhs1: ⊤ ⇒ Infinity2Closed(Rzero)
  Int2RMinus: Infinity2Closed(Rzero)
    rhs1: ⊤ ⇒ RRealMinus
  RPlusStar2Int: RRealPlusStar
    rhs1: ⊤ ⇒ Open2Infinity(Rzero)
  Int2RPlusStar: Open2Infinity(Rzero)
    rhs1: ⊤ ⇒ RRealPlusStar
  RMinusStar2Int: RRealMinusStar
    rhs1: ⊤ ⇒ Infinity2Open(Rzero)
  Int2RMinusStar: Infinity2Open(Rzero)
```

```
    rhs1: ⊤ ⇒ RRealMinusStar
intervalInclusion:
Metavariables
  a1: RReal
  b1: RReal
  a2: RReal
  b2: RReal
Rewrite Rules
  c2InfIncc2Inf: Closed2Infinity(a1) ⊆ Closed2Infinity(a2)
    rhs1: ⊤ ⇒ a2 ↦ a1 ∈ leq
  o2InfIncc2Inf: Open2Infinity(a1) ⊆ Closed2Infinity(a2)
    rhs1: ⊤ ⇒ a2 ↦ a1 ∈ leq
  c2InfInco2Inf: Closed2Infinity(a1) ⊆ Open2Infinity(a2)
    rhs1: ⊤ ⇒ a2 ↦ a1 ∈ lt
  o2InfInco2Inf: Open2Infinity(a1) ⊆ Open2Infinity(a2)
    rhs1: ⊤ ⇒ a2 ↦ a1 ∈ leq
  inf2cIncinf2c: Infinity2Closed(b1) ⊆ Infinity2Closed(b2)
    rhs1: ⊤ ⇒ b1 ↦ b2 ∈ leq
  inf2oIncinf2c: Infinity2Open(b1) ⊆ Infinity2Closed(b2)
    rhs1: ⊤ ⇒ b1 ↦ b2 ∈ leq
  inf2cIncinf2o: Infinity2Closed(b1) ⊆ Infinity2Open(b2)
    rhs1: ⊤ ⇒ b1 ↦ b2 ∈ lt
  inf2oIncinf2o: Infinity2Open(b1) ⊆ Infinity2Open(b2)
    rhs1: ⊤ ⇒ b1 ↦ b2 ∈ leq
  o2oIno2o: Open2Open(a1,b1) ⊆ Open2Open(a2,b2)
    rhs1: ⊤ ⇒ a1 ↦ a2 ∈ lt ∧ b1 ↦ b2 ∈ gt
  c2oInc2c: Closed2Open(a1,b1) ⊆ Closed2Closed(a2,b2)
    rhs1: ⊤ ⇒ a1 ↦ a2 ∈ leq ∧ b1 ↦ b2 ∈ geq
intervalInclusion2:
Metavariables
  a: RReal
  b: RReal
Rewrite Rules
  s_o2oIno2Inf: Open2Open(a,b) ⊆ Open2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_o2cIno2Inf: Open2Closed(a,b) ⊆ Open2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_o2oInc2Inf: Open2Open(a,b) ⊆ Closed2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_o2cInc2Inf: Open2Closed(a,b) ⊆ Closed2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_c2oInc2Inf: Closed2Open(a,b) ⊆ Closed2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_c2cInc2Inf: Closed2Closed(a,b) ⊆ Closed2Infinity(a)
    rhs1: ⊤ ⇒ ⊤
  s_o2oIno2c: Open2Open(a,b) ⊆ Open2Closed(a,b)
    rhs1: ⊤ ⇒ ⊤
  s_o2oInc2o: Open2Open(a,b) ⊆ Closed2Open(a,b)
    rhs1: ⊤ ⇒ ⊤
  s_o2oInc2c: Open2Open(a,b) ⊆ Closed2Closed(a,b)
```

```
      rhs1: ⊤ ⇒ ⊤
    s_c2oInc2c: Closed2Open(a,b) ⊆ Closed2Closed(a,b)
      rhs1: ⊤ ⇒ ⊤
    s_o2cInc2c: Open2Closed(a,b) ⊆ Closed2Closed(a,b)
      rhs1: ⊤ ⇒ ⊤
  degenerated_intervals:
  Metavariables
    a: RReal
  Rewrite Rules
    c2c_single: Closed2Closed(a,a)
      rhs1: ⊤ ⇒ {a}
    o2c_empty: Open2Closed(a,a)
      rhs1: ⊤ ⇒ ∅ : ℙ(RReal)
    c2o_empty: Closed2Open(a,a)
      rhs1: ⊤ ⇒ ∅ : ℙ(RReal)
    o2o_empty: Open2Open(a,a)
      rhs1: ⊤ ⇒ ∅ : ℙ(RReal)
  intervalElements:
  Metavariables
    a: RReal
    b: RReal
  Rewrite Rules
    a_in_c2c: a ∈ Closed2Closed(a,b)
      rhs1: ⊤ ⇒ a ↦ b ∈ leq
    a_in_c2o: a ∈ Closed2Open(a,b)
      rhs1: ⊤ ⇒ a ↦ b ∈ lt
    b_in_c2c: b ∈ Closed2Closed(a,b)
      rhs1: ⊤ ⇒ a ↦ b ∈ leq
    b_in_o2c: b ∈ Open2Closed(a,b)
      rhs1: ⊤ ⇒ a ↦ b ∈ lt
    a_in_c2inf: a ∈ Closed2Infinity(a)
      rhs1: ⊤ ⇒ ⊤
    b_in_inf2c: b ∈ Infinity2Closed(b)
      rhs1: ⊤ ⇒ ⊤
END
```

A.3 Theories of Functions and Piecewise Functions

This section show the theories of functions and piecewise functions presented in Section 4.3.2.3. The theory of functions, in particular, defines the general principles of continuity, differentiability, Lipschitz-continuity, etc. The theory of piecewise functions extends these principles to piecewise-defined functions.

Theory of Functions

Listings A.7: Functions theory

```
THEORY
  IMPORT THEORY Intervals
```

```
TYPE PARAMETERS E,F,G,H,I
OPERATORS
  bind expression (fab: E ⇸ F, gac: E ⇸ G)
    direct definition
      (λ x · x ∈ dom(fab) ∩ dom(gac) | (fab(x) ↦ gac(x)))
  fproj1 expression (fa_bc: E ⇸ F×G)
    direct definition
      (λ x↦y · x ∈ F ∧ y ∈ G | x) ∘ fa_bc
  fproj2 expression (fa_bc: E ⇸ F×G)
    direct definition
      (λ x↦y · x ∈ F ∧ y ∈ G | y) ∘ fa_bc
  Rfplus expression ()
    direct definition
      (λ rf ↦ rg · rf ∈ (RReal → RReal) ∧ rg ∈ (RReal → RReal) |
        (λ x · x ∈ RReal | plus(rf(x) ↦ rg(x)))
      )
  Rftimes expression ()
    direct definition
      (λ rf ↦ rg · rf ∈ (RReal → RReal) ∧ rg ∈ (RReal → RReal) |
        (λ x · x ∈ RReal | times(rf(x) ↦ rg(x)))
      )
  Rfscal expression ()
    direct definition
      (λ l ↦ rf · l ∈ RReal ∧ rf ∈ (RReal → RReal) |
        (λ x · x ∈ RReal | times(l ↦ rf(x)))
      )
  Rfcste expression (l: RReal)
    direct definition
      (λ x · x ∈ RReal | l)
  partial1 expression (fab_c: E×F⇸G, y: F)
    direct definition
      (λ x · (x ↦ y) ∈ dom(fab_c) | fab_c(x↦y))
  partial2 expression (fab_c: E×F⇸G, x: E)
    direct definition
      (λ y · (x ↦ y) ∈ dom(fab_c) | fab_c(x↦y))
  partialComp expression (fabb: E×F⇸G, gab: E⇸F)
    direct definition
      (λ t · t ∈ dom(gab) ∧ (t ↦ gab(t)) ∈ dom(fabb) | fabb(t ↦ gab(t)))
  unpartialize1 expression (A: ℙ(E), fbc: F⇸G)
    direct definition
      (λ x↦y · x ∈ A ∧ y ∈ dom(fbc) | fbc(y))
  unpartialize2 expression (B: ℙ(F), fac: E ⇸ G)
    direct definition
      (λ x↦y · x ∈ dom(fac) ∧ y ∈ B | fac(x))
  increasing predicate (AR: ℙ(RReal), f: RReal ⇸ RReal)
    well−definedness AR ⊆ dom(f)
    direct definition
      ∀ x, y · x ∈ AR ∧ y ∈ AR ⇒ (
        (x ↦ y ∈ leq) ⇔ (f(x) ↦ f(y) ∈ leq)
      )
```

```
strictlyIncreasing predicate (AR: ℙ(RReal),f: RReal ⇸ RReal)
  well−definedness AR ⊆ dom(f)
  direct definition
    ∀ x, y · x ∈ AR ∧ y ∈ AR ⇒ (
      (x ↦ y ∈ lt) ⇔ (f(x) ↦ f(y) ∈ lt)
    )
decreasing predicate (AR: ℙ(RReal),f: AR ⇸ RReal)
  well−definedness AR ⊆ dom(f)
  direct definition
    ∀ x, y · x ∈ AR ∧ y ∈ AR ⇒ (
      (x ↦ y ∈ leq) ⇔ (f(x) ↦ f(y) ∈ geq)
    )
strictlyDecreasing predicate (AR: ℙ(RReal),f: AR ⇸ RReal)
  well−definedness AR ⊆ dom(f)
  direct definition
    ∀ x, y · x ∈ AR ∧ y ∈ AR ⇒ (
      (x ↦ y ∈ lt) ⇔ (f(x) ↦ f(y) ∈ gt)
    )
constant predicate (A: ℙ(E),fa: E ⇸ F)
  well−definedness A ⊆ dom(fa)
  direct definition
    ∀ x · x ∈ A ⇒ (∀ y · y ∈ A ⇒ fa(x) = fa(y))
positive predicate (A: ℙ(E),far: E ⇸ RReal)
  well−definedness A ⊆ dom(far)
  direct definition
    ∀ x · x ∈ A ⇒ Rzero ↦ far(x) ∈ leq
strictlyPositive predicate (A: ℙ(E),far: E ⇸ RReal)
  well−definedness A ⊆ dom(far)
  direct definition
    ∀ x · x ∈ A ⇒ Rzero ↦ far(x) ∈ lt
negative predicate (A: ℙ(E),far: E ⇸ RReal)
  well−definedness A ⊆ dom(far)
  direct definition
    ∀ x · x ∈ A ⇒ Rzero ↦ far(x) ∈ geq
strictlyNegative predicate (A: ℙ(E),far: E ⇸ RReal)
  well−definedness A ⊆ dom(far)
  direct definition
    ∀ x · x ∈ A ⇒ Rzero ↦ far(x) ∈ gt
application expression ()
  direct definition
    (λ f_ ↦ e_ · f_ ∈ E ⇸ F ∧ e_ ∈ dom(f_) | f_(e_))
until expression (start: RReal,fre: RReal ⇸ E,t0: RReal,gre: RReal ⇸ E)
  well−definedness start ↦ t0 ∈ leq,Closed2Open(start,t0) ⊆ dom(fre),
      Closed2Infinity(t0) ⊆ dom(gre)
  direct definition
    (Closed2Open(start,t0) ◁ fre) ∪ (Closed2Infinity(t0) ◁ gre)
untilF expression (start: RReal,fref: RReal×E ⇸ F,t0: RReal,gref: RReal×E
    ⇸ F)
  well−definedness start ↦ t0 ∈ leq,Closed2Open(start,t0)×∅ ⊆ dom(fref),
      Closed2Infinity(t0)×∅ ⊆ dom(gref)
    direct definition
      ((Closed2Open(start ,t0)×E) ◁ fref) ∪ ((Closed2Infinity(t0)×E) ◁ gref
          )
  fcste expression (A: ℙ(E),v: F)
    direct definition
      (λ x · x∈ A | v)
  boundedBy predicate (A: ℙ(E),far: E ⇸ RReal,fmin: RReal,fmax: RReal)
    well−definedness A ⊆ dom(far)
    direct definition
      ∀ x_ · x_ ∈ A ⇒ fmin ↦ far(x_) ∈ leq ∧ far(x_) ↦ fmax ∈ leq
AXIOMATIC DEFINITIONS
 continuity:
   OPERATORS
     C0 expression (A: ℙ(E),B: ℙ(F)) : ℙ(ℙ(E×F))
       well−definedness A ≠ ∅,B ≠ ∅
     D1 expression (A2: ℙ(RReal),B: ℙ(F)) : ℙ(ℙ(RReal×F))
     Cn expression (n: ℕ,A: ℙ(E),B: ℙ(F)) : ℙ(ℙ(E×F))
     Dn expression (n: ℕ,A2: ℙ(RReal),B: ℙ(F)) : ℙ(ℙ(RReal×F))
       well−definedness n > 0
   AXIOMS
     cid:
       ∀ A,B · A ⊆ E ∧ B ⊆ F
         ⇒ (C0(A,B) = Cn(0,A,B))
     did:
       ∀ A,B · A ⊆ RReal ∧ B ⊆ F
         ⇒ (D1(A,B) = Dn(1,A,B))
     c_in_c:
       ∀ A,B,k · A ⊆ E ∧ B ⊆ F ∧ k ∈ ℕ
         ⇒ (Cn(k+1,A,B) ⊂ Cn(k,A,B))
     d_in_d:
       ∀ A,B,k · A ⊆ RReal ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 0
         ⇒ (Dn(k+1,A,B) ⊂ Dn(k,A,B))
     c_in_d:
       ∀ A,B,k · A ⊆ RReal ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 0
         ⇒ (Cn(k,A,B) ⊂ Dn(k,A,B))
     d_in_c:
       ∀ A,B,k · A ⊆ RReal ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 0
         ⇒ (Dn(k,A,B) ⊂ Cn(k−1,A,B))
     c_inclusion_stable:
       ∀ A,C,B,k ·
         A ⊆ E ∧ C ⊆ A ∧ B ⊆ F ∧ k ∈ ℕ
           ⇒
             (Cn(k,A,B) ⊆ Cn(k,C,B))
     d_inclusion_stable:
       ∀ A,C,B,k ·
         A ⊆ RReal ∧ C ⊆ A ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 0
           ⇒
             (Dn(k,A,B) ⊆ Dn(k,C,B))
     d_restriction:
       ∀ A,C,B,k,f ·
```

```
          A ⊆ RReal ∧ C ⊆ A ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 0 ∧
          f ∈ RReal ⇸ B ∧ A ⊆ dom(f) ∧ f ∈ Dn(k,A,B) ⇒
            (C ◁ f) ∈ Dn(k,C,B) derivative:
   OPERATORS
     Der expression (A2: ℙ(RReal),B: ℙ(F),fa2b: RReal ⇸ B) : ℙ(RReal×F)
       well−definedness fa2b ∈ D1(A2,B),A2 ⊆ dom(fa2b)
     Dern expression (n: ℕ,A2: ℙ(RReal),B: ℙ(F),fa2b: RReal ⇸ B) : ℙ(RReal×
         F)
       well−definedness n > 0,A2 ⊆ dom(fa2b)
   AXIOMS
     derType:
       ∀ A,B,f · A ⊆ RReal ∧ B ⊆ F ∧ f ∈ (RReal ⇸ B) ∧ A ⊆ dom(f) ∧
          f ∈ D1(A,B) ⇒ Der(A,B,f) ∈ (A → F)
     derDef_1:
       ∀ A,B,f · A ⊆ RReal ∧ B ⊆ F ∧ f ∈ (RReal ⇸ B) ∧ A ⊆ dom(f) ∧
          f ∈ Dn(1,A,B) ⇒ Dern(1,A,B,f) = Der(A,B,f)
     dernDef_n:
       ∀ A,B,f,k · A ⊆ RReal ∧ B ⊆ F ∧ f ∈ (RReal ⇸ B) ∧ A ⊆ dom(f) ∧ k ∈
            ℕ ∧ k > 1 ∧
          f ∈ Dn(k,A,B) ⇒ Dern(k,A,B,f) = Der(A,B,Dern(k−1,A,B,f))
     c_def:
       ∀ A,B,k,f · A ⊆ RReal ∧ B ⊆ F ∧ k ∈ ℕ ∧ k > 1 ∧ f ∈ (RReal ⇸ B) ∧
            dom(f) ⊆ A ⇒ (
          (f ∈ Cn(k,A,B) ⇔ (f ∈ Dn(k,A,B) ∧ Dern(k,A,B,f) ∈ C0(A,B)))
       )
 cartesian:
   AXIOMS
     cartesian_continuity:
       ∀ A,B,C,f,g,k ·
          A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ (A → B) ∧ g ∈ (A → C) ∧ k ∈ ℕ ⇒
            (f ∈ Cn(k,A,B) ∧ g ∈ Cn(k,A,C) ⇔ (bind(f,g) ∈ Cn(k,A,B×C)))
     cartesian_derivability:
       ∀ A,B,C,f,g,k ·
          A ⊆ RReal ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ (A → B) ∧ g ∈ (A → C) ∧ k ∈ ℕ ∧
              k > 0 ⇒
            (f ∈ Dn(k,A,B) ∧ g ∈ Dn(k,A,C) ⇔ (bind(f,g) ∈ Dn(k,A,B×C)))
     cartesian_der:
       ∀ A,B,C,f,g ·
          A ⊆ RReal ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ (A → B) ∧ g ∈ (A → C) ∧
          f ∈ D1(A,B) ∧ g ∈ D1(A,C) ⇒
            (Der(A,B×C,bind(f,g)) = bind(Der(A,B,f),Der(A,C,g)))
     cartesian_dern:
       ∀ A,B,C,f,g,k ·
          A ⊆ RReal ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ (A → B) ∧ g ∈ (A → C) ∧ k ∈ ℕ ∧
              k > 0 ∧
          f ∈ Dn(k,A,B) ∧ g ∈ Dn(k,A,C) ⇒
            (Dern(k,A,B×C,bind(f,g)) = bind(Dern(k,A,B,f),Dern(k,A,C,g)))
 partials:
   AXIOMS
     partial_cue:
        ∀ A,B,C,f,k ·
          A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ (A×B→C) ∧ k ∈ ℕ ⇒ (
            f ∈ Cn(k,A×B,C) ⇔ ((∀ y0 · y0 ∈ B ⇒ partial1(f,y0) ∈ Cn(k,A,C))
                ∧ (∀ x0 · x0 ∈ A ⇒ partial2(f,x0) ∈ Cn(k,B,C)))
        )
  real_functions:
    AXIOMS
      cste_cue:
        ∀ l · l ∈ RReal ⇒ (∀ k · k ∈ ℕ ⇒ (Rfcste(l) ∈ Cn(k,RReal,RReal)))
      cste_der:
        ∀ l · l ∈ RReal ⇒ (∀ k · k ∈ ℕ ∧ k > 0 ⇒ (Rfcste(l) ∈ Dn(k,RReal,
            RReal)))
      cste_der_def:
        ∀ l · l ∈ RReal ⇒ (Der(RReal,RReal,Rfcste(l)) = Rfcste(Rzero))
      plus_cue:
        ∀ A,f,g,k ·
          A ⊆ RReal ∧
          f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
          k ∈ ℕ ∧
          f ∈ Cn(k,A,RReal) ∧ g ∈ Cn(k,A,RReal) ⇒
            (Rfplus(f ↦ g) ∈ Cn(k,A,RReal))
      plus_der:
        ∀ A,f,g,k ·
          A ⊆ RReal ∧
          f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
          k ∈ ℕ ∧ k > 0 ∧
          f ∈ Dn(k,A,RReal) ∧ g ∈ Dn(k,A,RReal) ⇒
            (Rfplus(f ↦ g) ∈ Dn(k,A,RReal))
      plus_der_def:
        ∀ A,f,g,k ·
          A ⊆ RReal ∧
          f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
          k ∈ ℕ ∧ k > 0 ∧
          f ∈ Dn(k,A,RReal) ∧ g ∈ Dn(k,A,RReal) ⇒
            (Dern(k,A,RReal,Rfplus(f ↦ g)) = Rfplus(Dern(k,A,RReal,f) ↦
                Dern(k,A,RReal,g)))
      times_cue:
        ∀ A,f,g,k ·
          A ⊆ RReal ∧
          f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
          k ∈ ℕ ∧
          f ∈ Cn(k,A,RReal) ∧ g ∈ Cn(k,A,RReal) ⇒
            (Rftimes(f ↦ g) ∈ Cn(k,A,RReal))
      times_der:
        ∀ A,f,g,k ·
          A ⊆ RReal ∧
          f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
          k ∈ ℕ ∧ k > 0 ∧
          f ∈ Dn(k,A,RReal) ∧ g ∈ Dn(k,A,RReal) ⇒
            (Rftimes(f ↦ g) ∈ Dn(k,A,RReal))
    times_der_def:
      ∀ A,f,g ·
        A ⊆ RReal ∧
        f ∈ (A → RReal) ∧ g ∈ (A → RReal) ∧
        f ∈ D1(A,RReal) ∧ g ∈ D1(A,RReal) ⇒
          (Der(A,RReal,Rftimes(f ↦ g)) = Rfplus(Rftimes(Der(A,RReal,f) ↦
              g) ↦ Rftimes(f ↦ Der(A,RReal,g))))
    scal_cue:
      ∀ A,l,f,k ·
        A ⊆ RReal ∧
        l ∈ RReal ∧ f ∈ (A → RReal) ∧
        k ∈ ℕ ∧
        f ∈ Cn(k,A,RReal) ⇒
          (Rfscal(l ↦ f) ∈ Cn(k,A,RReal))
    scal_der:
      ∀ A,l,f,k ·
        A ⊆ RReal ∧
        l ∈ RReal ∧ f ∈ (A → RReal) ∧
        k ∈ ℕ ∧ k > 0 ∧
        f ∈ Dn(k,A,RReal) ⇒
          (Rfscal(l ↦ f) ∈ Dn(k,A,RReal))
    scal_der_def:
      ∀ A,l,f,k ·
        A ⊆ RReal ∧
        l ∈ RReal ∧ f ∈ (A → RReal) ∧
        k ∈ ℕ ∧ k > 0 ∧
        f ∈ Dn(k,A,RReal) ⇒
          (Dern(k,A,RReal,Rfscal(l ↦ f)) = Rfscal(l ↦ Dern(k,A,RReal,f)))
    comp_cue:
      ∀ A,B,f,g,k ·
        A ⊆ RReal ∧ B ⊆ RReal ∧
        f ∈ A → B ∧ g ∈ B → RReal ∧
        k ∈ ℕ ∧
        f ∈ Cn(k,A,B) ∧ g ∈ Cn(k,B,RReal) ⇒
          (g ∘ f ∈ Cn(k,A,RReal))
    comp_der:
      ∀ A,B,f,g,k ·
        A ⊆ RReal ∧ B ⊆ RReal ∧
        f ∈ A → B ∧ g ∈ B → RReal ∧
        k ∈ ℕ ∧ k > 0 ∧
        f ∈ Dn(k,A,B) ∧ g ∈ Dn(k,B,RReal) ⇒
          (g ∘ f ∈ Dn(k,A,RReal))
    comp_der_def:
      ∀ A,B,f,g ·
        A ⊆ RReal ∧ B ⊆ RReal ∧
        f ∈ A → B ∧ g ∈ B → RReal ∧
        f ∈ D1(A,B) ∧ g ∈ D1(B,RReal) ⇒
          (Der(A,RReal,g ∘ f) = Rftimes((Der(B,RReal,g) ∘ f) ↦ Der(A,B,f))
              )
  lipschitz:
```

```
  OPERATORS
    lipschitzContinuous predicate (A: ℙ(E),B: ℙ(F),fab: E ⇸ F) :
      well−definedness A ⊆ dom(fab)
    kLipschitzContinuous predicate (A: ℙ(E),B: ℙ(F),fab: E ⇸ F,r: RReal) :
      well−definedness A ⊆ dom(fab),Rzero ↦ r ∈ leq
  AXIOMS
    klip_lip:
      ∀ A,B,f · A ⊆ E ∧ B ⊆ F ∧ f ∈ E ⇸ F ∧ A ⊆ dom(f) ⇒ (
        lipschitzContinuous(A,B,f) ⇔ (∃ k · k ∈ RReal ∧ Rzero↦k ∈ leq ⇒
            kLipschitzContinuous(A,B,f,k))
      )
    lip_cue:
      ∀ A,B,f · A ⊆ E ∧ B ⊆ F ∧ f ∈ E ⇸ F ∧ A ⊆ dom(f) ⇒ (
        lipschitzContinuous(A,B,f) ⇒ (f ∈ C0(A,B))
      )
    lip_inclusion_stable:
      ∀ A,B,C,f ·
        A ⊆ E ∧ B ⊆ F ∧ C ⊆ A ∧ A ⊆ dom(f) ∧ f ∈ E ⇸ F ⇒ (
          lipschitzContinuous(A,B,f) ⇒ lipschitzContinuous(C,B,f)
        )
    klip_inclusion_stable:
      ∀ A,B,C,f,k ·
        A ⊆ E ∧ B ⊆ F ∧ C ⊆ A ∧ f ∈ E ⇸ F ∧ A ⊆ dom(f) ∧ k ∈ RReal ∧
            Rzero ↦ k ∈ leq ⇒ (
          kLipschitzContinuous(A,B,f,k) ⇒ kLipschitzContinuous(C,B,f,k)
        )
trigo:
  OPERATORS
    pi expression () : RReal
    sin expression () : ℙ(RReal×RReal)
    cos expression () : ℙ(RReal×RReal)
  AXIOMS
    pi_pos:
      Rzero ↦ pi ∈ lt
    sin_dom:
      sin ∈ RReal → RReal
    cos_dom:
      cos ∈ RReal → RReal
    sin_bounded:
      boundedBy(RReal,sin,uminus(Rone),Rone)
    sin_ran:
      ran(sin) = Closed2Closed(uminus(Rone),Rone)
    cos_bounded:
      boundedBy(RReal,cos,uminus(Rone),Rone)
    cos_ran:
      ran(cos) = Closed2Closed(uminus(Rone),Rone)
    cos_cinf:
      ∀ n · n ∈ ℕ ⇒ cos ∈ Cn(n,RReal,RReal)
    sin_cinf:
      ∀ n · n ∈ ℕ ⇒ sin ∈ Cn(n,RReal,RReal)
```

```
        cos0 :
          cos(Rzero) = Rone
        cospi :
          cos(pi) = uminus(Rone)
        cospi2 :
          cos(divide(pi↦Rtwo)) = Rzero
        cosmpi2 :
          cos(uminus(divide(pi↦Rtwo))) = Rzero
        sin0 :
          cos(Rzero) = Rzero
        sinpi :
          cos(pi) = uminus(Rzero)
        sinpi2 :
          cos(divide(pi↦Rtwo)) = Rone
        sinmpi2 :
          cos(uminus(divide(pi↦Rtwo))) = uminus(Rone)
        sin_odd :
          ∀ x · x ∈ RReal ⇒ sin(uminus(x)) = uminus(sin(x))
        cos_even :
          ∀ x · x ∈ RReal ⇒ cos(uminus(x)) = cos(x)
        sin_refl_pi4 :
          ∀ x · x ∈ RReal ⇒ sin(minus(divide(pi↦Rtwo) ↦ x)) = cos(x)
        cos_refl_pi4 :
          ∀ x · x ∈ RReal ⇒ cos(minus(divide(pi↦Rtwo) ↦ x)) = sin(x)
        sin_refl_pi2 :
          ∀ x · x ∈ RReal ⇒ sin(minus(pi ↦ x)) = sin(x)
        cos_refl_pi2 :
          ∀ x · x ∈ RReal ⇒ cos(minus(pi ↦ x)) = uminus(cos(x))
        sin_2pi :
          ∀ x · x ∈ RReal ⇒ sin(plus(x ↦ times(Rtwo ↦ pi))) = sin(x)
        cos_2pi :
          ∀ x · x ∈ RReal ⇒ cos(plus(x ↦ times(Rtwo ↦ pi))) = cos(x)
  basic_topology :
    OPERATORS
        IsOpen predicate (A: ℙ(E)) :
    AXIOMS
        open_continuity :
          ∀ x,f,A,B ·
            x ∈ A ∧ A ⊆ E ∧ B ⊆ F ∧
            f ∈ E ⇸ F ∧ A ⊆ dom(f) ∧ f ∈ C0(A,F) ∧
            IsOpen(B) ∧ f(x) ∈ B ⇒
              (∃ y · y ∈ A ∧ f(y) ∈ B)
        open_continuity_R :
          ∀ x,f,A,B ·
            A ⊆ RReal ∧ B ⊆ F ∧
            x ∈ RReal ∧
            f ∈ RReal ⇸ F ∧ f ∈ C0(A,F) ∧
            IsOpen(B) ∧ f(x) ∈ B ⇒
              (∃ y · y ∈ A ∧ x ↦ y ∈ lt ∧ f(y) ∈ B)
THEOREMS
```

```
functionEquality:
  ∀ f, g · f ∈ E ⇸ F ∧ g ∈ E ⇸ F ⇒ (
    (f = g) ⇔ (dom(f) = dom(g) ∧ (∀ x · x ∈ dom(f) ⇒ (f(x) = g(x))))
  )
bind_type:
  ∀ A,B,C,f,g · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B ∧ g ∈ A → C ⇒
    bind(f,g) ∈ A → B×C
proj1_type:
  ∀ A,B,C,f · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B×C ⇒ (
    fproj1(f) ∈ A → B
  )
proj2_type:
  ∀ A,B,C,f · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B×C ⇒ (
    fproj2(f) ∈ A → C
  )
partial1_type:
  ∀ A,B,C,f,y · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A×B → C ∧ y ∈ B ⇒ (
    partial1(f,y) ∈ A → C
  )
partial2_type:
  ∀ A,B,C,f,x · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A×B → C ∧ x ∈ A ⇒ (
    partial2(f,x) ∈ B → C
  )
partialComp_type:
  ∀ A,B,B2,f,g ·
    A ⊆ E ∧ B ⊆ F ∧ B2 ⊆ G ∧
    f ∈ A×B → B2 ∧ g ∈ A → B ⇒
      partialComp(f,g) ∈ A → B2
proj1_bind:
  ∀ A,B,C,f,g · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B ∧ g ∈ A → C ⇒ (
    fproj1(bind(f,g)) = f
  )
proj2_bind:
  ∀ A,B,C,f,g · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B ∧ g ∈ A → C ⇒ (
    fproj2(bind(f,g)) = g
  )
bind_proj:
  ∀ A,B,C,f · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B×C ⇒ (
    bind(fproj1(f),fproj2(f)) = f
  )
proj_cue:
  ∀ A,B,C,f,k · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → B×C ∧ k ∈ ℕ ⇒ (
    f ∈ Cn(k,A,B×C) ⇔ (fproj1(f) ∈ Cn(k,A,B) ∧ fproj2(f) ∈ Cn(k,A,C))
  )
unpartialized1_cue:
  ∀ A,B,C,f,k · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ B → C ∧ k ∈ ℕ ∧ f ∈ Cn(k,B,
      C) ⇒
    (unpartialize1(A,f) ∈ Cn(k,A×B,C))
unpartialized2_cue:
  ∀ A,B,C,f,k · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → C ∧ k ∈ ℕ ∧ f ∈ Cn(k,A,
      C) ⇒
    (unpartialize2(B,f) ∈ Cn(k,A×B,C))
unpartialize2_partial1:
  ∀ A,B,C,f · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ A → C
    ⇒
      (∀ y · y ∈ B ⇒ partial1(unpartialize2(B,f),y) = f)
unpartialize1_partial2:
  ∀ A,B,C,f · A ⊆ E ∧ B ⊆ F ∧ C ⊆ G ∧ f ∈ B → C
    ⇒
      (∀ x · x ∈ A ⇒ partial2(unpartialize1(A,f),x) = f)
MeanValue:
  ∀ a,b,f,M ·
    a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧
    f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
        Closed2Closed(a,b),RReal) ∧
    M ∈ RReal ∧ (∀ x · x ∈ Closed2Closed(a,b) ⇒ (abs(Der(Closed2Closed(a,
        b),RReal,f)(x)) ↦ M ∈ leq)) ⇒
      (abs(divide(minus(f(b) ↦ f(a)) ↦ minus(b ↦ a))) ↦ M ∈ leq)
functionConcat:
  ∀ A, B, f, g ·
    A ⊆ E ∧ B ⊆ E ∧ f ∈ A → F ∧ g ∈ B → F ∧
    (∀ x · x ∈ (A ∩ B) ⇒ f(x) = g(x)) ⇒
      (f ∪ g) ∈ (A ∪ B) → F
functionConcatTyping:
  ∀ f, g ·
    f ∈ E ⇸ F ∧ g ∈ E ⇸ F ∧
    (∀ x · x ∈ (dom(f) ∩ dom(g)) ⇒ f(x) = g(x)) ⇒
      (f ∪ g) ∈ (dom(f) ∪ dom(g)) → F
restrictionTyping:
  ∀ A,B,f · A ⊆ E ∧ B ⊆ A ∧ f ∈ A → F ⇒ ((B ◁ f) ∈ B → F)
restrictionDomain:
  ∀ f,A · f ∈ E ⇸ F ∧ A ⊆ E ⇒ dom(A ◁ f) = A
partialFunction:
  ∀ A,B,f1,f2 ·
    A ⊆ E ∧ B ⊆ E ∧ A ∩ B = ∅ ∧
    f1 ∈ A → F ∧ f2 ∈ B → F ⇒ (
      (A ◁ (f1 ∪ f2)) = f1 ∧
      (B ◁ (f1 ∪ f2)) = f2
    )
strictlyPositiveWeakening:
  ∀ A,f ·
    A ⊆ RReal ∧ f ∈ RReal ⇸ RReal ∧ A ⊆ dom(f) ∧
    strictlyPositive(A,f) ⇒
      positive(A,f)
strictlyNegativeWeakening:
  ∀ A,f ·
    A ⊆ RReal ∧ f ∈ RReal ⇸ RReal ∧ A ⊆ dom(f) ∧
    strictlyNegative(A,f) ⇒
      negative(A,f)
```

```
positiveDer2Increasing:
  ∀ A, f ·
    A ⊆ RReal ∧
    f ∈ A → RReal ∧ f ∈ D1(A,RReal) ⇒ (
      positive(A,Der(A,RReal,f))
      ⇔
        increasing(A,f)
    )
strictlyPositiveDer2StrictlyIncreasing:
  ∀ A, f ·
    A ⊆ RReal ∧
    f ∈ RReal ⇸ RReal ∧ A ⊆ dom(f) ∧ f ∈ D1(A,RReal) ⇒ (
      strictlyPositive(A,Der(A,RReal,f))
      ⇔
        strictlyIncreasing(A,f)
    )
negativeDer2Decreasing:
  ∀ A, f ·
    A ⊆ RReal ∧
    f ∈ A → RReal ∧ f ∈ D1(A,RReal) ⇒ (
      negative(A,Der(A,RReal,f))
      ⇔
        decreasing(A,f)
    )
strictlyNegativeDer2StrictlyDecreasing:
  ∀ A, f ·
    A ⊆ RReal ∧
    f ∈ RReal ⇸ RReal ∧ A ⊆ dom(f) ∧ f ∈ D1(A,RReal) ⇒ (
      strictlyNegative(A,Der(A,RReal,f))
      ⇔
        strictlyDecreasing(A,f)
    )
zeroDer2Constant:
  ∀ A, f ·
    A ⊆ RReal ∧
    f ∈ RReal ⇸ RReal ∧ A ⊆ dom(f) ∧ f ∈ D1(A,RReal) ⇒ (
      (∀ x · x ∈ A ⇒ Der(A,RReal,f)(x) = Rzero)
      ⇔
        constant(A,f)
    )
meanValue_geq:
  ∀ a,b,m,f ·
    a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧ m ∈ RReal ∧
    f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
        Closed2Closed(a,b),RReal) ∧
    f(a) ↦ m ∈ geq ∧ positive(Closed2Closed(a,b),Der(Closed2Closed(a,b),
        RReal,f)) ⇒
      (∀ t · t ∈ Closed2Closed(a,b) ⇒ f(t) ↦ m ∈ geq)
meanValue_geq_strict:
  ∀ a,b,m,f ·
```

```
      a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧ m ∈ RReal ∧
      f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
          Closed2Closed(a,b),RReal) ∧
      f(a) ↦ m ∈ geq ∧ strictlyPositive(Closed2Closed(a,b),Der(
          Closed2Closed(a,b),RReal,f)) ⇒
        (∀ t · t ∈ Open2Closed(a,b) ⇒ f(t) ↦ m ∈ gt)
  meanValue_leq:
    ∀ a,b,m,f ·
      a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧ m ∈ RReal ∧
      f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
          Closed2Closed(a,b),RReal) ∧
      f(a) ↦ m ∈ leq ∧ negative(Closed2Closed(a,b),Der(Closed2Closed(a,b),
          RReal,f)) ⇒
        (∀ t · t ∈ Closed2Closed(a,b) ⇒ f(t) ↦ m ∈ leq)
  meanValue_leq_strict:
    ∀ a,b,m,f ·
      a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧ m ∈ RReal ∧
      f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
          Closed2Closed(a,b),RReal) ∧
      f(a) ↦ m ∈ leq ∧ strictlyNegative(Closed2Closed(a,b),Der(
          Closed2Closed(a,b),RReal,f)) ⇒
        (∀ t · t ∈ Open2Closed(a,b) ⇒ f(t) ↦ m ∈ lt)
  meanValue_positivity:
    ∀ a,b,f ·
      a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧
      f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
          Closed2Closed(a,b),RReal) ∧
      f(a) ↦ Rzero ∈ geq ∧ positive(Closed2Closed(a,b),Der(Closed2Closed(a
          ,b),RReal,f)) ⇒
        positive(Closed2Closed(a,b),f)
  meanValue_negativity:
    ∀ a,b,f ·
      a ∈ RReal ∧ b ∈ RReal ∧ a ↦ b ∈ lt ∧
      f ∈ RReal ⇸ RReal ∧ Closed2Closed(a,b) ⊆ dom(f) ∧ f ∈ D1(
          Closed2Closed(a,b),RReal) ∧
      f(a) ↦ Rzero ∈ leq ∧ negative(Closed2Closed(a,b),Der(Closed2Closed(a
          ,b),RReal,f)) ⇒
        negative(Closed2Closed(a,b),f)
  bind_bind_partialComp:
    ∀ f1,f2,x1,x2,Af,Bf,Cf,Ax,Bx,Cx ·
      Af ⊆ E ∧ Bf ⊆ F ∧ Cf ⊆ G ∧
      Ax ⊆ Af ∧ Bx ⊆ Bf ∧ Cx ⊆ Cf ∧
      f1 ∈ Af×(Bf×Cf) → Bf ∧ f2 ∈ Af×(Bf×Cf) → Cf ∧
      x1 ∈ Ax → Bx ∧ x2 ∈ Ax → Cx ⇒
        partialComp(
          bind(f1,f2),
          bind(x1,x2)
        ) = bind(
          partialComp(f1,bind(x1,x2)),
          partialComp(f2,bind(x1,x2))
        )
  until_type:
    ∀ s,t0,f,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal ⇸ E ∧ g ∈ RReal ⇸ E ∧
      Closed2Open(s,t0) ⊆ dom(f) ∧ Closed2Infinity(t0) ⊆ dom(g) ⇒
        until(s,f,t0,g) ∈ Closed2Infinity(s) → E
  until_restrict1:
    ∀ s,t0,f,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal ⇸ E ∧ g ∈ RReal ⇸ E ∧
      Closed2Open(s,t0) ⊆ dom(f) ∧ Closed2Infinity(t0) ⊆ dom(g) ⇒
        (Closed2Open(s,t0) ◁ until(s,f,t0,g)) = f
  until_restrict2:
    ∀ s,t0,f,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal ⇸ E ∧ g ∈ RReal ⇸ E ∧
      Closed2Open(s,t0) ⊆ dom(f) ∧ Closed2Infinity(t0) ⊆ dom(g) ⇒
        (Closed2Infinity(t0) ◁ until(s,f,t0,g)) = g
  until_bind1:
    ∀ s,t0,f1,f2,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f1 ∈ RReal ⇸ E ∧ f2 ∈ RReal ⇸ E ∧ g ∈ RReal ⇸ F ∧
      Closed2Open(s,t0) ⊆ dom(f1) ∧ Closed2Infinity(t0) ⊆ dom(f2) ⇒ (
        bind(until(s,f1,t0,f2),g) = until(s,bind(f1,g),t0,bind(f2,g))
      )
  until_bind2:
    ∀ s,t0,f,g1,g2 ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal ⇸ E ∧ g1 ∈ RReal ⇸ F ∧ g2 ∈ RReal ⇸ F ∧
      Closed2Open(s,t0) ⊆ dom(g1) ∧ Closed2Infinity(t0) ⊆ dom(g2) ⇒ (
        bind(f,until(s,g1,t0,g2)) = until(s,bind(f,g1),t0,bind(f,g2))
      )
  untilF_type:
    ∀ s,t0,f,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal×E ⇸ F ∧ g ∈ RReal×E ⇸ F ∧
      Closed2Open(s,t0)×∅ ⊆ dom(f) ∧ Closed2Infinity(t0)×∅ ⊆ dom(g) ⇒
        untilF(s,f,t0,g) ∈ RReal×E ⇸ F
  untilF_type_strong:
    ∀ s,t0,f,g,A ·
      A ⊆ E ∧
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f ∈ RReal×A → F ∧ g ∈ RReal×A → F ⇒
        untilF(s,f,t0,g) ∈ Closed2Infinity(s)×A → F
  untilF_bind1:
    ∀ s,t0,f1,f2,g ·
      s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
      f1 ∈ RReal×E ⇸ F ∧ f2 ∈ RReal×E ⇸ F ∧ g ∈ RReal×E ⇸ G ∧
      Closed2Open(s,t0)×∅ ⊆ dom(f1) ∧ Closed2Infinity(t0)×∅ ⊆ dom(f2) ⇒ (
          bind(untilF(s,f1,t0,f2),g) = untilF(s,bind(f1,g),t0,bind(f2,g))
        )
    untilF_bind2:
      ∀ s,t0,f,g1,g2 ·
        s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
        f ∈ RReal×E ⇸ F ∧ g1 ∈ RReal×E ⇸ G ∧ g2 ∈ RReal×E ⇸ G ∧
        Closed2Open(s,t0)×∅ ⊆ dom(g1) ∧ Closed2Infinity(t0)×∅ ⊆ dom(g2) ⇒ (
          bind(f,untilF(s,g1,t0,g2)) = untilF(s,bind(f,g1),t0,bind(f,g2))
        )
    fcste_cue:
      ∀ A,l · A ⊆ E ∧ l ∈ F ⇒ (fcste(A,l) ∈ C0(A,F))
    fcste_type:
      ∀ A,l · A ⊆ E ∧ l ∈ F ⇒ fcste(A,l) ∈ A → F
PROOF RULES
    typing:
    Metavariables
      A: ℙ(E)
      B: ℙ(F)
      C: ℙ(G)
      fab: ℙ(E×F)
      gac: ℙ(E×G)
      fa_bc: ℙ(E×(F×G))
      fab_c: ℙ(E×F×G)
      x: E
      y: F
      A2: ℙ(E)
    Rewrite Rules
      domainRestrictionInvolutive: A ◁ (A ◁ fab)
        rhs1: ⊤ ⇒ A ◁ fab
      domainRestrictionComplete: A ◁ fab
        rhs1: dom(fab) = A ⇒ fab
      basicDomainRestrictionTyping: (A ◁ fab) ∈ E ⇸ F
        rhs1: ⊤ ⇒ fab ∈ E ⇸ F
      basicBindTyping: bind(fab,gac) ∈ E ⇸ F×G
        rhs1: ⊤ ⇒ fab ∈ E ⇸ F ∧ gac ∈ E ⇸ G
      bindDomain: dom(bind(fab,gac))
        rhs1: ⊤ ⇒ dom(fab) ∩ dom(gac)
      fcsteTyping: fcste(A,y) ∈ A → F
        rhs1: ⊤ ⇒ ⊤
      bindDomSubset: A ⊆ dom(bind(fab,gac))
        rhs1: ⊤ ⇒ A ⊆ dom(fab) ∧ A ⊆ dom(gac)
      fproj1Typing: fproj1(fa_bc) ∈ E ⇸ F
        rhs1: ⊤ ⇒ fa_bc ∈ E ⇸ F×G
      fproj2Typing: fproj2(fa_bc) ∈ E ⇸ G
        rhs1: ⊤ ⇒ fa_bc ∈ E ⇸ F×G
      fproj1Domain: dom(fproj1(fa_bc))
        rhs1: ⊤ ⇒ dom(fa_bc)
      fproj2Domain: dom(fproj2(fa_bc))
        rhs1: ⊤ ⇒ dom(fa_bc)
    Inference Rules
    bindType: fab ∈ A → B, gac ∈ A → C ⊢ bind(fab, gac) ∈ A→B×C
    proj1Type: fa_bc ∈ A → B×C ⊢ fproj1(fa_bc) ∈ A → B
    proj2Type: fa_bc ∈ A → B×C ⊢ fproj2(fa_bc) ∈ A → C
    partial1Type: fab_c ∈ A×B → C, y ∈ B ⊢ partial1(fab_c, y) ∈ A → C
    partial2Type: fab_c ∈ A×B → C, x ∈ A ⊢ partial2(fab_c, x) ∈ B → C
    domainRestrictionType: A2 ⊆ A, fab ∈ A → B ⊢ (A2 ◁ fab) ∈ A2 → B
binding:
Metavariables
    A: ℙ(E)
    B: ℙ(F)
    C: ℙ(G)
    AR: ℙ(RReal)
    fab: ℙ(E×F)
    gac: ℙ(E×G)
    fab2: ℙ(E×F)
    gac2: ℙ(E×G)
    fa_bc: ℙ(E×(F×G))
    farb: ℙ(RReal×F)
    garc: ℙ(RReal×G)
    far_bc: ℙ(RReal×(F×G))
    n: ℤ
    x: E
Rewrite Rules
    bindProjRew: bind(fproj1(fa_bc), fproj2(fa_bc))
        rhs1: ⊤ ⇒ fa_bc
    bindDerRew: Der(AR, B×C, bind(farb, garc))
        rhs1: ⊤ ⇒ bind(Der(AR, B, farb), Der(AR, C, garc))
    bindEquality: bind(fab, gac)=bind(fab2, gac2)
        rhs1: ⊤ ⇒ fab = fab2 ∧ gac = gac2
    bindC0Rew: bind(fab, gac) ∈ C0(A, B×C)
        rhs1: ⊤ ⇒ fab ∈ C0(A, B) ∧ gac ∈ C0(A, C)
    bindD1Rew: bind(farb, garc) ∈ D1(AR, B×C)
        rhs1: ⊤ ⇒ farb ∈ D1(AR, B) ∧ garc ∈ D1(AR, C)
    bindCnRew: bind(farb, garc) ∈ Cn(n, AR, B×C)
        rhs1: n ≥ 0 ⇒ farb ∈ Cn(n, AR, B) ∧ garc ∈ Cn(n, AR, C)
    bindDnRew: bind(farb, garc) ∈ Dn(n, AR, B×C)
        rhs1: n > 0 ⇒ farb ∈ Dn(n, AR, B) ∧ garc ∈ Dn(n, AR, C)
    bindRestrict: A ◁ bind(fab, gac)
        rhs1: ⊤ ⇒ bind(A ◁ fab, A ◁ gac)
    bindEvaulate: bind(fab, gac)(x)
        rhs1: ⊤ ⇒ fab(x)↦gac(x)
continuity:
Metavariables
    A: ℙ(E)
    B: ℙ(F)
    AR: ℙ(RReal)
    n: ℤ
Rewrite Rules
    C0_to_Cn0: C0(A, B)
        rhs1: ⊤ ⇒ Cn(0, A, B)
        D1_toDn1: D1(AR,B)
          rhs1: ⊤ ⇒ Dn(1,AR,B)
        Cn0_to_C0: Cn(0,A,B)
          rhs1: ⊤ ⇒ C0(A,B)
        Dn1_to_D1: Dn(1,AR,B)
          rhs1: ⊤ ⇒ D1(AR,B)
      misc:
      Metavariables
        f: ℙ(E×F)
        g: ℙ(E×F)
      Rewrite Rules
        fun_equality: f = g
          rhs1: ⊤ ⇒ dom(f) = dom(g) ∧ (∀x · x ∈ dom(f) ⇒ f(x) = g(x))
      direct_product:
      Metavariables
        fab: ℙ(E×F)
        gac: ℙ(E×G)
      Rewrite Rules
        dirprodType: fab ⊗ gac ∈ E ⇸ F×G
          rhs1: ⊤ ⇒ fab ∈ E ⇸ F ∧ gac ∈ E ⇸ G
        dirprodDomain: dom(fab ⊗ gac)
          rhs1: ⊤ ⇒ dom(fab) ∩ dom(gac)
END
```

Theory of Piecewise Functions

Listings A.8: Piecewise theory

```
THEORY
  IMPORT THEORY Functions
  TYPE PARAMETERS E,F,G
  OPERATORS
     partitionS predicate (X: ℙ(E),Xs: ℙ(ℙ(E)))
        direct definition
          (∀ X1,X2 · X1 ∈ Xs ∧ X2 ∈ Xs ∧ X1 ≠ X2 ⇒ X1 ∩ X2 = ∅) ∧
          union(Xs) = X
     piecewiseContinuous predicate (Ix: ℙ(ℙ(E)),B: ℙ(F),f: union(Ix) → B)
        well−definedness Ix ≠ ∅,∀ I1,I2 · I1 ∈ Ix ∧ I2 ∈ Ix ∧ I1 ≠ I2 ⇒ I1 ∩
            I2 = ∅
        direct definition
          ∀ I0 · I0 ∈ Ix ⇒ (I0 ◁ f) ∈ C0(I0,B)
     partialPiecewiseContinuous predicate (Ix: ℙ(ℙ(E)),B: ℙ(F),C: ℙ(G),g: union(
          Ix)×B → C)
        well−definedness Ix ≠ ∅,∀ I1,I2 · I1 ∈ Ix ∧ I2 ∈ Ix ∧ I1 ≠ I2 ⇒ I1 ∩
            I2 = ∅
        direct definition
          ∀ I0 · I0 ∈ Ix ⇒ ((I0×B) ◁ g) ∈ C0(I0×B,C)
     piecewiseLipschitzContinuous predicate (Ix: ℙ(ℙ(E)),B: ℙ(F),f: union(Ix) → B
          )
```

```
      well−definedness Ix ≠ ∅,∀ I1,I2 · I1 ∈ Ix ∧ I2 ∈ Ix ∧ I1 ≠ I2 ⇒ I1 ∩
          I2 = ∅
      direct definition
        ∀ I0 · I0 ∈ Ix ⇒ lipschitzContinuous(I0,B,I0 ◁ f)
  THEOREMS
    untilPiecewise:
      ∀ s,t0,f,g ·
        s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
        f ∈ RReal ⇸ E ∧ g ∈ RReal ⇸ E ∧
        Closed2Open(s,t0) ⊆ dom(f) ∧ Closed2Infinity(t0) ⊆ dom(g) ∧
        f ∈ C0(Closed2Open(s,t0),E) ∧ g ∈ C0(Closed2Infinity(t0),E) ⇒
          piecewiseContinuous({Closed2Open(s,t0),Closed2Infinity(t0)},E,until
              (s,f,t0,g))
    untilFPartialPiecewise:
      ∀ s,t0,f,g ·
        s ∈ RReal ∧ t0 ∈ RReal ∧ s ↦ t0 ∈ leq ∧
        f ∈ RReal×E ⇸ F ∧ g ∈ RReal×E ⇸ F ∧
        Closed2Open(s,t0)×∅ ⊆ dom(f) ∧ Closed2Infinity(t0)×∅ ⊆ dom(g) ∧
        f ∈ C0(Closed2Open(s,t0)×E,F) ∧ g ∈ C0(Closed2Infinity(t0)×E,F) ⇒
          partialPiecewiseContinuous({Closed2Open(s,t0),Closed2Infinity(t0)},
              E,F,untilF(s,f,t0,g))
END
```

A.4 Theory of Differential Equations

This section presents the theory of differential equations detailed in Section 4.3.2.4. This fundamental theory defines differential equations as a particular type, with associated constructors, as well as multiple operators to characterise their properties and solutions. It also defines several theorems/axioms for establishing various properties of differential equations (e.g. solvability).

Listings A.9: Differential Equations theory

```
THEORY
  IMPORT THEORY Piecewise
  TYPE PARAMETERS E,F,UF,STATES,F1,F2
  DATA TYPES
    DE(F)
    CONSTRUCTORS
      ode(fun:ℙ(RReal×F×F),initial:F,initialArg:RReal)
      aode(afun:ℙ(F×F),ainitial:F,ainitialArg:RReal)
    CDE(F,UF)
    CONSTRUCTORS
      code(cfun:ℙ((RReal×F×UF)×F),cinit:F,cinitArg:RReal)
      caode(cafun:ℙ((F×UF)×F),cainit:F,cainitArg:RReal)
  OPERATORS
    autonomousToODE expression (e: DE(F))
      well−definedness ∃ f,i,ia · f ∈ F ⇸ F ∧ i ∈ F ∧ ia ∈ RReal ∧ e = aode(
          f,i,ia)
      direct definition
    ode((λ t_↦eta_ · t_ ∈ RReal ∧ eta_ ∈ F ∧ eta_ ∈ dom(afun(e)) | afun(e
        )(eta_)),initial(e),initialArg(e))
solutionOf predicate (DR: ℙ(RReal),eta: RReal ⇸ F,eq: DE(F))
  well−definedness DR ⊆ dom(eta)
SolutionsOf expression (DR: ℙ(RReal),eq: DE(F))
  direct definition
    { eta_ | eta_ ∈ RReal ⇸ F ∧ DR ⊆ dom(eta_) ∧ solutionOf(DR,eta_,eq)
        }
CauchyLipschitzCondition predicate (DR: ℙ(RReal),DF: ℙ(F),eq: DE(F))
PiecewiseCauchyLipschitzCondition predicate (DRs: ℙ(ℙ(RReal)),DF: ℙ(F),eq:
    DE(F))
  well−definedness DRs ≠ ∅,∀ DR1,DR2 · DR1 ∈ DRs ∧ DR2 ∈ DRs ∧ DR1 ≠ DR2
      ⇒ DR1 ∩ DR2 = ∅
Solvable predicate (DR: ℙ(RReal),eq: DE(F))
  direct definition
    ∃ x · x ∈ RReal ⇸ F ∧ DR ⊆ dom(x) ∧ solutionOf(DR,x,eq)
AppendSolutionBAP predicate (eq: DE(F),DR: ℙ(RReal),A: ℙ(DR),B: ℙ(DR),
    eta: RReal ⇸ F,etap: RReal ⇸ F)
  well−definedness A ∩ B = ∅,Solvable(B,eq),DR ⊆ dom(eta)
  direct definition
    DR ⊆ dom(etap) ∧
    (A ◁ etap = A ◁ eta) ∧
    solutionOf(B,B ◁ etap, eq)
CBAP predicate (t: RRealPlus,tp: RRealPlus,eta: RReal ⇸ F,etap: RReal
    ⇸ F,Pred: ℙ((RReal⇸F)×(RReal⇸F)),Inv: ℙ(F))
  well−definedness Closed2Closed(Rzero,t) ⊆ dom(eta),Closed2Closed(Rzero,
      tp) ⊆ dom(etap)
  direct definition
    t ↦ tp ∈ lt ∧
    Closed2Open(Rzero,t) ◁ eta = Closed2Open(Rzero,t) ◁ etap ∧
    ((Closed2Closed(t,tp) ◁ eta) ↦ (Closed2Closed(t,tp) ◁ etap)) ∈ Pred
        ∧
    (∀ t_ · t_ ∈ Closed2Closed(t,tp) ⇒ etap(t_) ∈ Inv)
CBAPsolutionOf predicate (t: RRealPlus,tp: RRealPlus,eta: RReal ⇸ F,etap
    : RReal ⇸ F,eq: DE(F),Inv: ℙ(F))
  well−definedness Closed2Closed(Rzero,t) ⊆ dom(eta),Closed2Closed(Rzero,
      tp) ⊆ dom(etap),Solvable(Closed2Closed(t,tp),eq),t ↦ tp ∈ lt
  direct definition
    CBAP(t,tp,eta,etap,(RReal⇸F)×SolutionsOf(Closed2Closed(t,tp),eq),Inv
        )
CBAPParallelEq predicate (t: RRealPlus,tp: RRealPlus,eta1: RReal ⇸ F1,
    eta1p: RReal ⇸ F1,eq1: DE(F1),eta2: RReal ⇸ F2,eta2p: RReal ⇸ F2,
    eq2: DE(F2),Inv12: ℙ(F1×F2))
  well−definedness Closed2Closed(Rzero,t) ⊆ dom(eta1),Closed2Closed(Rzero
      ,tp) ⊆ dom(eta1p),Closed2Closed(Rzero,t) ⊆ dom(eta2),Closed2Closed(
      Rzero,tp) ⊆ dom(eta2p),t ↦ tp ∈ lt
  direct definition
    t ↦ tp ∈ lt ∧
    Closed2Open(Rzero,t) ◁ eta1 = Closed2Open(Rzero,t) ◁ eta1p ∧
    solutionOf(Closed2Closed(t,tp),Closed2Closed(t,tp) ◁ eta1p,eq1) ∧
```

```
        eta1(t) = eta1p(t) ∧
        Closed2Open(Rzero,t) ◁ eta2 = Closed2Open(Rzero,t) ◁ eta2p ∧
        solutionOf(Closed2Closed(t,tp),Closed2Closed(t,tp) ◁ eta2p,eq2) ∧
        eta2(t) = eta2p(t) ∧
        (∀ t_ · t_ ∈ Closed2Closed(t,tp) ⇒ eta1p(t_)↦eta2p(t_) ∈ Inv12)
  VerifiesOn predicate (DR: ℙ(RReal),eta: RReal ⇸ F,Inv: ℙ(F))
    well−definedness DR ⊆ dom(eta)
    direct definition
      ∀ t_ · t_ ∈ DR ⇒ eta(t_) ∈ Inv
  withControl expression (DR: ℙ(RReal),ce: CDE(F,UF),u: RReal ⇸ UF)
    well−definedness DR ⊆ dom(u)
  Controllable predicate (DR: ℙ(RReal),ce: CDE(F,UF))
    direct definition
      ∃ u · u ∈ RReal ⇸ UF ∧ DR ⊆ dom(u) ∧ Solvable(DR,withControl(DR,ce,u
          ))
  ControllableOn predicate (DR: ℙ(RReal),DU: ℙ(UF),ce: CDE(F,UF))
    direct definition
      ∃ u · u ∈ RReal ⇸ UF ∧ DR ⊆ dom(u) ∧ Solvable(DR,withControl(DR,ce,u
          )) ∧ (∀ t_ · t_ ∈ DR ⇒ u(t_) ∈ DU)
  SolvableWith predicate (DR: ℙ(RReal),ce: CDE(F,UF),u: RReal ⇸ UF)
    well−definedness DR ⊆ dom(u)
    direct definition
      Solvable(DR,withControl(DR,ce,u))
  CBAPFIS predicate (t: RRealPlus,tp: RRealPlus,eta: RReal ⇸ F,Pred: ℙ((
      RReal ⇸ F)×(RReal ⇸ F)),Inv: ℙ(F))
    well−definedness Closed2Closed(Rzero,t) ⊆ dom(eta),t ↦ tp ∈ lt
    direct definition
      ∃ etap ·
        etap ∈ RReal ⇸ F ∧ Closed2Closed(t,tp) ⊆ dom(etap) ∧
        ((Closed2Closed(t,tp) ◁ eta) ↦ (Closed2Closed(t,tp) ◁ etap)) ∈
            Pred ∧
        (∀ t_ · t_ ∈ Closed2Closed(t,tp) ⇒ etap(t_) ∈ Inv)
  CBAPsolutionOfFIS predicate (t: RRealPlus,tp: RRealPlus,eta: RReal ⇸ F,
      eq: DE(F),Inv: ℙ(F))
    well−definedness Closed2Closed(Rzero,t) ⊆ dom(eta),t ↦ tp ∈ lt,
        Solvable(Closed2Closed(t,tp),eq)
    direct definition
      ∃ etap ·
        etap ∈ RReal ⇸ F ∧ Closed2Closed(t,tp) ⊆ dom(etap) ∧
        solutionOf(Closed2Closed(t,tp),etap,eq) ∧
        eta(t) = etap(t) ∧
        (∀ t_ · t_ ∈ Closed2Closed(t,tp) ⇒ etap(t_) ∈ Inv)
  CBAPParallelEqFIS predicate (t: RRealPlus,tp: RRealPlus,eta1: RReal ⇸ F1
      ,eq1: DE(F1),eta2: RReal ⇸ F2,eq2: DE(F2),Inv12: ℙ(F1×F2))
    well−definedness t ↦ tp ∈ lt,Closed2Closed(Rzero,t) ⊆ dom(eta1),
        Closed2Closed(Rzero,t) ⊆ dom(eta2),Solvable(Closed2Closed(t,tp),eq1
        ),Solvable(Closed2Closed(t,tp),eq2)
    direct definition
      ∃ eta1p,eta2p ·
        eta1p ∈ RReal ⇸ F1 ∧ Closed2Closed(t,tp) ⊆ dom(eta1p) ∧
        eta2p ∈ RReal ⇸ F2 ∧ Closed2Closed(t,tp) ⊆ dom(eta2p) ∧
        solutionOf(Closed2Closed(t,tp),eta1p,eq1) ∧
        solutionOf(Closed2Closed(t,tp),eta2p,eq2) ∧
        eta1(t) = eta1p(t) ∧ eta2(t) = eta2p(t) ∧
        (∀ t_ · t_ ∈ Closed2Closed(t,tp) ⇒ (eta1p(t_)↦eta2p(t_)) ∈ Inv12)
THEOREMS
  solutionDer:
    ∀ fu,DR,DF,init,initArg,eta,k ·
      DR ⊆ RReal ∧ DF ⊆ F ∧
      fu ∈ (DR×DF→F) ∧ k ∈ ℕ ∧ k > 0 ∧ (∀ x0 · x0 ∈ DF ⇒ partial1(fu,x0)
          ∈ Dn(k,DR,F)) ∧
      init ∈ DF ∧ initArg ∈ DR ∧
      eta ∈ RReal ⇸ F ∧ DR ⊆ dom(eta) ∧
      solutionOf(DR,eta,ode(fu,init,initArg)) ⇒
        eta ∈ Dn(k+1,DR,F)
  SolutionsOf_solutionOf:
    ∀ DR,eq,eta ·
      DR ⊆ RReal ∧ eq ∈ DE(F) ∧ eta ∈ RReal ⇸ F ∧ DR ⊆ dom(eta) ⇒
        (eta ∈ SolutionsOf(DR,eq) ⇔ solutionOf(DR,eta,eq))
  CauchyLipschitz:
    ∀ eq,DR,DF ·
      DR ⊆ RReal ∧ DF ⊆ F ∧ eq ∈ DE(F) ∧
      CauchyLipschitzCondition(DR,DF,eq)
        ⇒
          Solvable(DR,eq)
  concatSolutions:
    ∀ DR1,DR2,eta1,eta2,eq ·
      DR1 ⊆ RReal ∧ DR2 ⊆ RReal ∧ DR1 ∩ DR2 = ∅ ∧
      eta1 ∈ RReal ⇸ F ∧ DR1 ⊆ dom(eta1) ∧
      eta2 ∈ RReal ⇸ F ∧ DR2 ⊆ dom(eta2) ∧
      eq ∈ DE(F) ∧
      solutionOf(DR1,eta1,eq) ∧ solutionOf(DR2,eta2,eq) ⇒
        solutionOf(DR1 ∪ DR2,eta1 ∪ eta2, eq)
  appendSolutionExistence:
    ∀ DR, A, B, eq, eta ·
      DR ⊆ RReal ∧
      A ⊆ DR ∧ B ⊆ DR ∧ A ∩ B = ∅ ∧ A ∪ B = DR ∧
      eq ∈ DE(F) ∧ Solvable(B,eq) ∧
      eta ∈ RReal ⇸ F ∧ DR ⊆ dom(eta) ⇒
        (∃ etap · etap ∈ DR → F ∧ AppendSolutionBAP(eq,DR,A,B,eta,etap))
  concatSolvable:
    ∀ DR1,DR2,eq ·
      DR1 ⊆ RReal ∧ DR2 ⊆ RReal ∧ DR1 ∩ DR2 = ∅ ∧
      eq ∈ DE(F) ∧ Solvable(DR1,eq) ∧ Solvable(DR2,eq) ⇒
        Solvable(DR1 ∪ DR2,eq)
  PiecewiseCauchyLipschitz:
    ∀ eq,DRs,DF ·
      DRs ⊆ ℙ(RReal) ∧ DRs ≠ ∅ ∧ (∀ DR1,DR2 · DR1 ∈ DRs ∧ DR2 ∈ DRs ∧ DR1
          ≠ DR2 ⇒ DR1 ∩ DR2 = ∅) ∧
      DF ⊆ F ∧ eq ∈ DE(F) ∧
      PiecewiseCauchyLipschitzCondition(DRs,DF,eq) ⇒
        Solvable(union(DRs),eq)
solution_restriction:
  ∀ DR1,DR2,eq,eta ·
     DR1 ⊆ RReal ∧ DR2 ⊆ DR1 ∧ eq ∈ DE(F) ∧
     eta ∈ RReal ⇸ F ∧ solutionOf(DR1,eta,eq)
       ⇒
         solutionOf(DR2,eta,eq)
solvable_restriction:
  ∀ DR1,DR2,eq ·
     DR1 ⊆ RReal ∧ DR2 ⊆ DR1 ∧ eq ∈ DE(F) ∧
     Solvable(DR1,eq)
       ⇒
         Solvable(DR2,eq)
CBAPsolutionOf_FIS:
  ∀ t,eta,eq,Inv ·
     t ∈ RRealPlus ∧
     eta ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,t) ⊆ dom(eta) ∧
     eq ∈ DE(F) ∧
     Inv ∈ ℙ(F) ∧ IsOpen(Inv) ∧ eta(t) ∈ Inv ∧
     Solvable(Closed2Infinity(t),eq)
     ⇒ (
       ∃ tp,etap ·
         tp ∈ RRealPlus ∧ t ↦ tp ∈ lt ∧
         etap ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,tp) ⊆ dom(etap) ∧
         Solvable(Closed2Closed(t,tp),eq) ∧
         CBAPsolutionOf(t,tp,eta,etap,eq,Inv)
     )
CBAPsolutionOf_INV:
  ∀ t,tp,eta,etap,eq,LocalInv,GlobalInv ·
     t ∈ RRealPlus ∧ tp ∈ RRealPlus ∧ t ↦ tp ∈ lt ∧
     eta ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,t) ⊆ dom(eta) ∧
     etap ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,tp) ⊆ dom(etap) ∧
     eq ∈ DE(F) ∧ Solvable(Closed2Closed(t,tp),eq) ∧
     LocalInv ⊆ F ∧ eta(t) ∈ LocalInv ∧
     GlobalInv ⊆ F ∧ (∀ t_ · t_ ∈ Closed2Closed(Rzero,t) ⇒ eta(t_) ∈
         GlobalInv) ∧
     CBAPsolutionOf(t,tp,eta,etap,eq,LocalInv ∩ GlobalInv)
       ⇒
         (∀ t_ · t_ ∈ Closed2Closed(Rzero,tp) ⇒ etap(t_) ∈ GlobalInv)
CBAPFIS_act_FIS:
  ∀ t,tp,eta,Pred,Inv ·
     t ∈ RRealPlus ∧ tp ∈ RRealPlus ∧ eta ∈ RReal ⇸ F ∧
     Pred ∈ ℙ((RReal ⇸ F)×(RReal ⇸ F)) ∧ Inv ∈ ℙ(F) ∧
     Closed2Closed(Rzero,t) ⊆ dom(eta) ∧ t ↦ tp ∈ lt ∧
     CBAPFIS(t,tp,eta,Pred,Inv)
       ⇒ (
         ∃ etap · etap ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,tp) ⊆ dom(etap)
              ∧
           CBAP(t,tp,eta,etap,Pred,Inv)
        )
    CBAPsolutionOfFIS_act_FIS:
      ∀ t,tp,eta,eq,Inv ·
        t ∈ RRealPlus ∧ tp ∈ RRealPlus ∧ eta ∈ RReal ⇸ F ∧
        eq ∈ DE(F) ∧ Inv ∈ ℙ(F) ∧
        Closed2Closed(Rzero,t) ⊆ dom(eta) ∧ t ↦ tp ∈ lt ∧
        CBAPsolutionOfFIS(t,tp,eta,eq,Inv)
          ⇒ (
            ∃ etap · etap ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,tp) ⊆ dom(etap)
                ∧
              CBAPsolutionOf(t,tp,eta,etap,eq,Inv)
        )
    CBAPFIS_restriction:
      ∀ t,tp,eta,Pred,Inv ·
        t ∈ RRealPlus ∧ tp ∈ RRealPlus ∧ eta ∈ RReal ⇸ F ∧
        Pred ∈ ℙ((RReal ⇸ F)×(RReal ⇸ F)) ∧ Inv ∈ ℙ(F) ∧
        Closed2Closed(Rzero,t) ⊆ dom(eta) ∧ t ↦ tp ∈ lt ∧
        CBAPFIS(t,tp,eta,Pred,Inv) ⇒ (
          ∀ tc · tc ∈ RRealPlus ∧ t ↦ tc ∈ lt ∧ tc ↦ tp ∈ leq ⇒
              CBAPFIS(t,tc,eta,Pred,Inv)
        )
    CBAPsolutionOfFIS_implies_CBAPFIS:
      ∀ t,tp,eta,eq,Inv ·
        t ∈ RRealPlus ∧ tp ∈ RRealPlus ∧ t ↦ tp ∈ lt ∧
        eta ∈ RReal ⇸ F ∧ Closed2Closed(Rzero,t) ⊆ dom(eta) ∧
        eq ∈ DE(F) ∧ Solvable(Closed2Closed(t,tp),eq) ∧
        Inv ∈ ℙ(F) ⇒ (
            CBAPsolutionOfFIS(t,tp,eta,eq,Inv)
          ⇔
            CBAPFIS(t,tp,eta,(RReal⇸F)×SolutionsOf(Closed2Closed(t,tp),eq),
                Inv)
        )
  PROOF RULES
    sof:
    Metavariables
      DR: ℙ(RReal)
      eq: DE(F)
      eta: ℙ(RReal×F)
    Rewrite Rules
      Solutions_to_solution: eta ∈ SolutionsOf(DR,eq)
        rhs1: ⊤ ⇒ solutionOf(DR,eta,eq)
      solution_to_Solutions: solutionOf(DR,eta,eq)
        rhs1: ⊤ ⇒ eta ∈ SolutionsOf(DR,eq)
    eq_typing:
    Metavariables
      DR: ℙ(RReal)
      ceq: CDE(F,UF)
      u: ℙ(RReal×UF)
    Rewrite Rules
      type_withControl: withControl(DR,ceq,u) ∈ DE(F)
```

```
        rhs1: ⊤ ⇒ ⊤
END
```

A.5 Theories of Approximation

This section gives the theories of approximation presented in Section 4.3.2.5. These two theories define the basic mathematical foundations for formalising approximation in Event-B, and in particular the general concept of approximation, together with various axioms to help handling it in models and proofs.

Note that the general theory of approximation is split into two components: *ApproximationBase* defines approximation as axiomatic operators, while *Approximation* expands on these operators to build more complex structure (explicitly). This separation is due to a limitation of the theory plug-in, that prevents axiomatic operators to be referenced by directly defined operators in the same theory component.

Theory for Approximation – Base

Listings A.10: Approximation (Base) theory

```
THEORY
  IMPORT THEORY DiffEq
  TYPE PARAMETERS E,F1,F2,F3,F
  OPERATORS
    absdiff commutative expression (a: RReal,b: RReal)
      direct definition
        abs(minus(a ↦ b))
  AXIOMATIC DEFINITIONS
   approx_rel:
    OPERATORS
      DeltaNeighborhood predicate (delta: RRealPlus,a: F,b: F) :
    AXIOMS
      deltaN_commutative:
        ∀ delta,a,b · delta ∈ RRealPlus ∧ a ∈ F ∧ b ∈ F ⇒
          (DeltaNeighborhood(delta,a,b) ⇔ DeltaNeighborhood(delta,b,a))
      deltaN_refl:
        ∀ delta,a · delta ∈ RRealPlus ∧ a ∈ F ⇒ DeltaNeighborhood(delta,a,a)
      deltaN_widen:
        ∀ a,b,delta1 · delta1 ∈ RRealPlus ∧ a ∈ F ∧ b ∈ F ∧ DeltaNeighborhood
            (delta1,a,b) ⇒
          (∀ delta2 · delta2 ∈ RRealPlus ∧ delta1 ↦ delta2 ∈ leq ⇒
              DeltaNeighborhood(delta2,a,b))
      deltaN_pseudo_trans:
        ∀ a,b,c,delta1,delta2 ·
          delta1 ∈ RRealPlus ∧ delta2 ∈ RRealPlus ∧
          a ∈ F ∧ b ∈ F ∧ c ∈ F ∧
          DeltaNeighborhood(delta1,a,b) ∧ DeltaNeighborhood(delta2,b,c) ⇒
            DeltaNeighborhood(plus(delta1↦delta2),a,c)
      deltaN_R:
        ∀ a,b,delta · a ∈ RReal ∧ b ∈ RReal ∧ delta ∈ RRealPlus ⇒
          (DeltaNeighborhood(delta,a,b) ⇔ (absdiff(a,b) ↦ delta ∈ leq))
      deltaN_R_add:
        ∀ a,b,k,delta · a ∈ RReal ∧ b ∈ RReal ∧ k ∈ RReal ∧ delta ∈ RRealPlus
            ⇒
          (DeltaNeighborhood(delta,a,b) ⇔ DeltaNeighborhood(delta,plus(a↦k),
              plus(b↦k)))
      deltaN_R_add2:
        ∀ a,b,c,d,delta1,delta2 ·
          a ∈ RReal ∧ b ∈ RReal ∧ c ∈ RReal ∧ d ∈ RReal ∧
          delta1 ∈ RRealPlus ∧ delta2 ∈ RRealPlus ∧
          DeltaNeighborhood(delta1,a,b) ∧ DeltaNeighborhood(delta2,c,d) ⇒
            DeltaNeighborhood(plus(delta1↦delta2),plus(a↦c),plus(b↦d))
END
```

Theory of Approximation

Listings A.11: Approximation theory

```
THEORY
  IMPORT THEORY ApproximationBase
  TYPE PARAMETERS E,F1,F2,F3,UF1,UF2,F
  OPERATORS
    FDeltaNeighborhood predicate (PE: ℙ(E),delta: RRealPlus,a: F,f2: E ⇸ F)
      well−definedness PE ⊆ dom(f2)
      direct definition
        ∀ x · x ∈ PE ⇒ DeltaNeighborhood(delta,a,f2(x))
    DeltaApproximation predicate (PE: ℙ(E),delta: RRealPlus,f1: E ⇸ F,f2: E
        ⇸ F)
      well−definedness PE ⊆ dom(f1),PE ⊆ dom(f2)
      direct definition
        ∀ x · x ∈ PE ⇒ DeltaNeighborhood(delta,f1(x),f2(x))
    DeltaApproximationEq predicate (DR: ℙ(RReal),delta: RRealPlus,e1: DE(F),
        e2: DE(F))
      well−definedness Solvable(DR,e1),Solvable(DR,e2)
      direct definition
        ∀ eta1,eta2 ·
          eta1 ∈ RReal ⇸ F ∧ eta2 ∈ RReal ⇸ F ∧
          DR ⊆ dom(eta1) ∧ DR ⊆ dom(eta2) ∧
          solutionOf(DR,eta1,e1) ∧ solutionOf(DR,eta2,e2)
            ⇒ DeltaApproximation(DR,delta,eta1,eta2)
    DeltaApproximationEqObs predicate (DR: ℙ(RReal),delta: RRealPlus,ee1: DE(
        F1),g1: F1 → F,ee2: DE(F2),g2: F2 → F)
      well−definedness Solvable(DR,ee1),Solvable(DR,ee2)
      direct definition
        ∀ eta1,eta2 ·
          eta1 ∈ RReal ⇸ F1 ∧ eta2 ∈ RReal ⇸ F2 ∧
          DR ⊆ dom(eta1) ∧ DR ⊆ dom(eta2) ∧
          solutionOf(DR,eta1,ee1) ∧ solutionOf(DR,eta2,ee2)
            ⇒ DeltaApproximation(DR,delta,g1 ∘ eta1,g2 ∘ eta2)
    DeltaApproximationEqF predicate (DR: ℙ(RReal),delta: RRealPlus,e: DE(F),g
        : RReal ⇸ F)
      well−definedness Solvable(DR,e),DR ⊆ dom(g)
      direct definition
        ∀ eta ·
          eta ∈ RReal ⇸ F ∧ DR ⊆ dom(eta) ∧ solutionOf(DR,eta,e)
            ⇒ DeltaApproximation(DR,delta,eta,g)
    DeltaShrink expression (delta: RRealPlus,SF: ℙ(F))
      direct definition
        { f2 | f2 ∈ F ∧ (∃ f1 · f1 ∈ SF ∧ DeltaNeighborhood(delta,f1,f2)) }
    DeltaNeighborhoodSet expression (delta: RRealPlus,x: E)
      direct definition
        { y | y ∈ E ∧ DeltaNeighborhood(delta,x,y) }
  AXIOMATIC DEFINITIONS
  simulation_functions:
    OPERATORS
      SimulationFunctions expression (DF1: ℙ(F1),DF2: ℙ(F2),DUF1: ℙ(UF1),DUF2:
          ℙ(UF2),f1: F1×UF1 ⇸ F1,f2: F2×UF2 ⇸ F2,g1: F1 → F,g2: F2 → F) :
          ℙ(ℙ((F1×F2)×RReal))
        well−definedness DF1×DUF1 ⊆ dom(f1),DF2×DUF2 ⊆ dom(f2)
    AXIOMS
      SimulationFunctions_char:
        ∀ DF1,DF2,DUF1,DUF2,f1,f2,g1,g2,V ·
          DF1 ⊆ F1 ∧ DF2 ⊆ F2 ∧ DUF1 ⊆ UF1 ∧ DUF2 ⊆ UF2 ∧
          f1 ∈ F1×UF1 ⇸ F1 ∧ f2 ∈ F2×UF2 ⇸ F2 ∧
          g1 ∈ F1 → F ∧ g2 ∈ F2 → F ∧
          DF1×DUF1 ⊆ dom(f1) ∧ DF2×DUF2 ⊆ dom(f2) ∧
          V ∈ SimulationFunctions(DF1,DF2,DUF1,DUF2,f1,f2,g1,g2) ∧
          V ∈ F1×F2 ⇸ RReal
            ⇒ (
              DF1×DF2 ⊆ dom(V)
                  ∧ (∀ x1,x2 · x1 ∈ DF1 ∧ x2 ∈ DF2 ⇒ V(x1↦x2) ∈ RRealPlus)
            )
      SimulationFunction_control_ODE:
        ∀ DR,eta01,eta02,t0,DF1,DF2,DUF1,DUF2,f1,f2,g1,g2,u1,u2,V ·
          DR ⊆ RReal ∧ t0 ∈ DR ∧ eta01 ∈ F1 ∧ eta02 ∈ F2 ∧
          DF1 ⊆ F1 ∧ DF2 ⊆ F2 ∧ DUF1 ⊆ UF1 ∧ DUF2 ⊆ UF2 ∧
          f1 ∈ F1×UF1 ⇸ F1 ∧ DF1×DUF1 ⊆ dom(f1) ∧
          f2 ∈ F2×UF2 ⇸ F2 ∧ DF2×DUF2 ⊆ dom(f2) ∧
          g1 ∈ F1 → F ∧ g2 ∈ F2 → F ∧
          u1 ∈ RReal ⇸ UF1 ∧ DR ⊆ dom(u1) ∧ ran(u1) ⊆ DUF1 ∧
          u2 ∈ RReal ⇸ UF2 ∧ DR ⊆ dom(u2) ∧ ran(u2) ⊆ DUF2 ∧
          V ∈ SimulationFunctions(DF1,DF2,DUF1,DUF2,f1,f2,g1,g2) ∧
          SolvableWith(DR,caode(f1,eta01,t0),u1) ∧ SolvableWith(DR,caode(f2,
              eta02,t0),u2) ⇒
            (∃ delta ·
              delta ∈ RRealPlus ∧
              DeltaApproximationEqObs(DR,delta,
                withControl(DR,caode(f1,eta01,t0),u1), g1,
                withControl(DR,caode(f2,eta02,t0),u2), g2
```

```
                    )
                 )
         SimulationFunction_control_ODE_delta:
            ∀ DR, eta01 , eta02 , t0 , DF1, DF2, DUF1, DUF2, f1 , f2 , g1 , g2 , u1 , u2 , V, delta  ·
               DR ⊆ RReal ∧ t0 ∈ DR ∧ eta01 ∈ F1 ∧ eta02 ∈ F2 ∧
               DF1 ⊆ F1 ∧ DF2 ⊆ F2 ∧ DUF1 ⊆ UF1 ∧ DUF2 ⊆ UF2 ∧
               f1 ∈ F1×UF1 ⇸ F1 ∧ DF1×DUF1 ⊆ dom(f1) ∧
               f2 ∈ F2×UF2 ⇸ F2 ∧ DF2×DUF2 ⊆ dom(f2) ∧
               g1 ∈ F1 → F ∧ g2 ∈ F2 → F ∧
               u1 ∈ RReal ⇸ UF1 ∧ DR ⊆ dom(u1) ∧ ran(u1) ⊆ UF1 ∧
               u2 ∈ RReal ⇸ UF2 ∧ DR ⊆ dom(u2) ∧ ran(u2) ⊆ UF2 ∧
               V ∈ SimulationFunctions(DF1, DF2, DUF1, DUF2, f1 , f2 , g1 , g2) ∧
               delta ∈ RReal ∧ Rzero ↦ delta ∈ lt ∧ boundedBy(DF1×DF2, V, Rzero ,
                    delta) ∧
               SolvableWith(DR, caode(f1 , eta01 , t0) , u1) ∧ SolvableWith(DR, caode(f2 ,
                    eta02 , t0) , u2) ⇒
                  DeltaApproximationEqObs(DR, delta ,
                     withControl(DR, caode(f1 , eta01 , t0) , u1) ,  g1 ,
                     withControl(DR, caode(f2 , eta02 , t0) , u2) ,  g2
                 )
   neighborhoods:
     AXIOMS
         deltaNSet_open:
            ∀ delta , x · delta ∈ RRealPlus ∧ Rzero ↦ delta ∈ lt ∧ x ∈ E ⇒
               IsOpen(DeltaNeighborhoodSet(delta , x))
   THEOREMS
      deltaA_restriction:
         ∀ PE, PE2, delta , f1 , f2  ·
            PE ⊆ E ∧ PE2 ⊆ PE ∧ delta ∈ RRealPlus ∧
            f1 ∈ E ⇸ F ∧ f2 ∈ E ⇸ F ∧
            PE ⊆ dom(f1) ∧ PE ⊆ dom(f2) ∧
            DeltaApproximation(PE, delta , f1 , f2)
               ⇒
                  DeltaApproximation(PE2, delta , f1 , f2)
      deltaAeq_restriction:
         ∀ DR, DR2, delta , e1 , e2  ·
            DR ⊆ RReal ∧ DR2 ⊆ DR ∧ delta ∈ RRealPlus ∧
            e1 ∈ DE(F) ∧ e2 ∈ DE(F) ∧
            Solvable(DR, e1) ∧ Solvable(DR2, e2) ∧
            DeltaApproximationEq(DR, delta , e1 , e2)
               ⇒
                  DeltaApproximationEq(DR2, delta , e1 , e2)
      deltaAeqobs_restriction:
         ∀ DR, DR2, delta , e1 , e2 , g1 , g2  ·
            DR ⊆ RReal ∧ DR2 ⊆ DR ∧ delta ∈ RRealPlus ∧
            e1 ∈ DE(F1) ∧ e2 ∈ DE(F2) ∧
            g1 ∈ F1 → F ∧ g2 ∈ F2 → F ∧
            Solvable(DR, e1) ∧ Solvable(DR2, e2) ∧
            DeltaApproximationEqObs(DR, delta , e1 , g1 , e2 , g2)
               ⇒
            DeltaApproximationEqObs(DR2, delta, e1, g1, e2, g2)
deltaA_comp:
  ∀ DF1, PE, f, g, h, delta ·
    DF1 ⊆ F1 ∧ PE ⊆ E ∧
    g ∈ F1 ⇸ E ∧ f ∈ E ⇸ F ∧ h ∈ E ⇸ F ∧
    DF1 ⊆ dom(g) ∧ PE ⊆ dom(f) ∧ PE ⊆ dom(h) ∧
    DeltaApproximation(PE, delta, f, h) ∧
    (∀ x · x ∈ DF1 ⇒ g(x) ∈ PE) ⇒
      DeltaApproximation(DF1, delta, f ∘ g, h ∘ g)
deltaA_commutative:
  ∀ PE, delta, f1, f2 ·
    PE ⊆ E ∧ delta ∈ RRealPlus ∧ f1 ∈ E ⇸ F ∧ f2 ∈ E ⇸ F ∧
      PE ⊆ dom(f1) ∧ PE ⊆ dom(f2) ⇒
    (DeltaApproximation(PE, delta, f1, f2) ⇔ DeltaApproximation(PE, delta, f2
        , f1))
deltaAeq_commutative:
  ∀ DR, delta, e1, e2 ·
    DR ⊆ RReal ∧ delta ∈ RRealPlus ∧ e1 ∈ DE(F) ∧ e2 ∈ DE(F) ∧ Solvable(
        DR, e1) ∧ Solvable(DR, e2) ⇒
      (DeltaApproximationEq(DR, delta, e1, e2) ⇔ DeltaApproximationEq(DR,
          delta, e2, e1))
deltaAeqobs_commutative:
  ∀ DR, delta, e1, e2, g1, g2 ·
    DR ⊆ RReal ∧ delta ∈ RRealPlus ∧
    e1 ∈ DE(F1) ∧ e2 ∈ DE(F2) ∧
    Solvable(DR, e1) ∧ Solvable(DR, e2) ∧
    g1 ∈ F1 → F ∧ g2 ∈ F2 → F
      ⇒
        (DeltaApproximationEqObs(DR, delta, e1, g1, e2, g2)
        ⇔
         DeltaApproximationEqObs(DR, delta, e2, g2, e1, g1))
deltaApp_induction:
  ∀ delta, eqA, eqC, etaA, etaC, etaAp, etaCp, t, tp, InvA, InvC ·
    delta ∈ RRealPlus ∧
    t ∈ RRealPlus ∧
    eqA ∈ DE(F) ∧ eqC ∈ DE(F) ∧
    etaA ∈ RReal ⇸ F ∧ Closed2Closed(Rzero, t) ⊆ dom(etaA) ∧
    etaC ∈ RReal ⇸ F ∧ Closed2Closed(Rzero, t) ⊆ dom(etaC) ∧
    DeltaApproximationEq(Closed2Closed(t, tp), delta, eqA, eqC) ∧
    DeltaApproximation(Closed2Closed(Rzero, t), delta, etaA, etaC) ∧
    CBAPsolutionOf(t, tp, etaA, etaAp, eqA, InvA) ∧ CBAPsolutionOf(t, tp, etaC,
        etaCp, eqC, InvC)
      ⇒ DeltaApproximation(Closed2Closed(Rzero, tp), delta, etaAp, etaCp)
deltaApp_obs_induction:
  ∀ delta, eqA, eqC, gA, gC, etaA, etaC, etaAp, etaCp, t, tp, InvA, InvC ·
    delta ∈ RRealPlus ∧
    t ∈ RRealPlus ∧
    eqA ∈ DE(F1) ∧ eqC ∈ DE(F2) ∧ gA ∈ F1 → F ∧ gC ∈ F2 → F ∧
    etaA ∈ RReal ⇸ F1 ∧ Closed2Closed(Rzero, t) ⊆ dom(etaA) ∧
    etaC ∈ RReal ⇸ F2 ∧ Closed2Closed(Rzero, t) ⊆ dom(etaC) ∧
```

```
        DeltaApproximationEqObs(Closed2Closed(t,tp),delta,eqA,gA,eqC,gC) ∧
        DeltaApproximation(Closed2Closed(Rzero,t),delta,gA ∘ etaA,gC ∘ etaC)
            ∧
        CBAPsolutionOf(t,tp,etaA,etaAp,eqA,InvA) ∧ CBAPsolutionOf(t,tp,etaC,
            etaCp,eqC,InvC)
          ⇒ DeltaApproximation(Closed2Closed(Rzero,tp),delta,gA ∘ etaAp,gC ∘
              etaCp)
    DeltaShrink_INV:
      ∀ delta,xA,xC,IA,IC ·
        delta ∈ RRealPlus ∧
        IA ⊆ F ∧ xA ∈ F ∧
        IC ⊆ F ∧ xC ∈ F ∧ xC ∈ IC ∧
        DeltaShrink(delta, IC) ⊆ IA ⇒
          xA ∈ IA
END
```

Appendix B

Models

This appendix gives the complete and "raw" Event-B models of the case studies presented in this manuscript, including their associated *domain theories*. Note that the notations used is taken straight from the model (no notation shortcut is used).

We first give the complete generic model (Section B.1), as it is the base of every other model presented afterwards.

The models are presented in order of appearance in the manuscript: automatic brake (Section B.2), signalised left-turn assist (Section B.3), water tanks (Section B.4), robot control (Section B.5) and inverted pendulum (Section B.6).

B.1 Generic Model

This section presents the generic model as discussed in Chapter 5. The model consists of a machine that abstract controller-plant loop hybrid system, accompanied by a basic context that defines the discrete and continuous state-space of the hybrid system being developed.

Context

Listings B.1: Generic model context

```
CONTEXT
   GenericCtx
SETS
   STATES
CONSTANTS
   S
AXIOMS
   axm1:  S = ...
END
```

Machine

Listings B.2: Generic model

```
MACHINE
   Generic
SEES
   GenericCtx
VARIABLES  t, x_p, x_s
INVARIANTS
   inv1:  $t \in RRealPlus$
   inv2:  $x\_p \in RRealPlus \pto S$
   inv3:  $x\_s \in STATES$
   inv4:  $Closed2Closed(Rzero, t) \subseteq dom(x\_p)$
EVENTS
   INITIALISATION
   THEN
      act1:  $t := Rzero$
      act2:  $x\_p :\in \{Rzero\} \rightarrow S$
      act3:  $x\_s :\in STATES$
   END

   Behave
   ANY  e, Inv, tp
   WHERE
      grd1:  $e \in DE(S)$
      grd2:  $Solvable(Closed2Infinity(t), e)$
      grd3:  $Inv \in \mathbb{P}(S)$
      grd4:  $x\_p(t) \in Inv$
      grd5:  $tp \in RRealPlus$
      grd6:  $t \mapsto tp \in lt \land CBAPsolutionOfFIS(t, tp, x\_p, e, Inv)$
   THEN
      act1:
         $t, x\_p :|$
            $t' = tp \land$
            $x\_p' \in RReal \pto S \land$
            $Closed2Closed(Rzero, t') \subseteq dom(x\_p') \land$
            $CBAPsolutionOf(t, t', x\_p, x\_p', e, Inv)$
   END

   Actuate
   ANY  e, s, Inv, tp
   WHERE
      grd1:  $e \in DE(S)$
      grd2:  $Solvable(Closed2Infinity(t), e)$
      grd3:  $s \subseteq STATES$
      grd4:  $x\_s \in s$
      grd5:  $Inv \in \mathbb{P}(S)$
      grd6:  $x\_p(t) \in Inv$
      grd7:  $tp \in RRealPlus$
      grd8:  $t \mapsto tp \in lt \land CBAPsolutionOfFIS(t, tp, x\_p, e, Inv)$
   THEN
      act1:
        t, x_p :|
            t' = tp∧
            x_p' ∈ RReal ⇸ S∧
            Closed2Closed(Rzero, t') ⊆ dom(x_p')∧
            CBAPsolutionOf(t, t', x_p, x_p', e, Inv)
    END

    Transition
    ANY    s
    WHERE
        grd1 :  s ∈ ℙ1(STATES)
    THEN
        act1 :  x_s :∈ s
    END

    Sense
    ANY    s ,  p
    WHERE
        grd1 :  s ∈ ℙ1(STATES)
        grd2 :  p ∈ ℙ(STATES × RReal × S)
        grd3 :  (x_s ↦ t ↦ x_p(t)) ∈ p
    THEN
        act1 :  x_s :∈ s
    END

END
```

B.2 Automatic Brake Case Study

This section gives the model for the automatic brake case study developed in Section 5.4.1.

Context

Listings B.3: Automatic brake context

```
CONTEXT
    AutobrakeCtx
EXTENDS
    GenericCtx
CONSTANTS
    stabilizing
    accelerating
    braking
    nearing_stop
    stopped
    A
    b
    v0
    SP
```

f2__speed
f1__deceleration
f1__stable
f1__acceleration
f__deceleration
f__stable
f__acceleration
eod
Vmax
x0

AXIOMS

axm1 : $partition(STATES, \{stabilizing\}, \{accelerating\}, \{braking\}, \{nearing\_\_stop\}, \{stopped\})$
axm2 : $A \in RReal$
axm3 : $Rzero \mapsto A \in lt$
axm4 : $b \in RReal$
axm5 : $Rzero \mapsto b \in lt$
axm51 : $b \neq Rzero$
axm6 : $v0 \in RReal$
axm7 : $Rzero \mapsto v0 \in lt$
axm62 : $x0 \in RReal$
axm72 : $Rzero \mapsto x0 \in lt$
axm8 : $SP \in RReal$
axm9 : $Rzero \mapsto SP \in lt$
axm010 : $Vmax \in RReal$
axm011 : $Rzero \mapsto Vmax \in lt$
axm154 : $f2\_\_speed \in (RRealPlus \times S) \rightarrow RReal$
axm155 : $f2\_\_speed = (\lambda t\_\_ \mapsto (v\_\_ \mapsto x\_\_) \cdot t\_\_ \in RRealPlus \wedge (v\_\_ \mapsto x\_\_) \in S \mid v\_\_)$
axm156 : $f1\_\_deceleration \in ((RRealPlus \times RRealPlus) \rightarrow (RRealPlus \times S \rightarrow RReal))$
axm11 :
$\forall t\_\_init, v\_\_init \cdot t\_\_init \in RRealPlus \wedge v\_\_init \in RRealPlus \Rightarrow ($
$f1\_\_deceleration(t\_\_init \mapsto v\_\_init) =$
$(\lambda t\_\_ \mapsto (v\_\_ \mapsto x\_\_) \cdot t\_\_ \in RRealPlus \wedge (v\_\_ \mapsto x\_\_) \in S \wedge$
$(t\_\_ \mapsto plus(divide(v\_\_init \mapsto b) \mapsto t\_\_init) \in lt) \mid uminus(b)) \cup$
$(\lambda t\_\_ \mapsto (v\_\_ \mapsto x\_\_) \cdot t\_\_ \in RRealPlus \wedge (v\_\_ \mapsto x\_\_) \in S \wedge$
$(t\_\_ \mapsto plus(divide(v\_\_init \mapsto b) \mapsto t\_\_init) \in geq) \mid Rzero)$
$)$
axm10 : $f\_\_deceleration \in ((RRealPlus \times RRealPlus) \rightarrow (RRealPlus \times S \rightarrow S))$
axm102 : $\forall t\_\_init, v\_\_init \cdot t\_\_init \in RRealPlus \wedge v\_\_init \in RRealPlus$
$\Rightarrow (f1\_\_deceleration(t\_\_init \mapsto v\_\_init) \in RRealPlus \times S \rightarrow RReal)$
axm101 :
$\forall t\_\_init, v\_\_init \cdot t\_\_init \in RRealPlus \wedge v\_\_init \in RRealPlus \Rightarrow$
$f\_\_deceleration(t\_\_init \mapsto v\_\_init) = bind(f1\_\_deceleration(t\_\_init \mapsto v\_\_init), f2\_\_speed)$
axm12 : $f1\_\_stable \in (RRealPlus \times S \rightarrow RReal)$
axm13 : $f1\_\_stable = (\lambda t\_\_ \mapsto (v\_\_ \mapsto x\_\_) \cdot t\_\_ \in RRealPlus \wedge (v\_\_ \mapsto x\_\_) \in S \mid Rzero)$
axm130 : $f\_\_stable \in (RRealPlus \times S \rightarrow S)$
axm131 : $f\_\_stable = bind(f1\_\_stable, f2\_\_speed)$
axm132 : $f\_\_stable \in C0(RRealPlus \times S, S)$
axm14 : $f1\_\_acceleration \in (RRealPlus \times S \rightarrow RReal)$
axm15 : $f1\_\_acceleration = (\lambda t\_\_ \mapsto (v\_\_ \mapsto x\_\_) \cdot t\_\_ \in RRealPlus \wedge (v\_\_ \mapsto x\_\_) \in S \mid A)$
axm150 : $f\_\_acceleration \in (RRealPlus \times S \rightarrow S)$

```
axm151: $f\_\_acceleration = bind(f1\_\_acceleration, f2\_\_speed)$
axm152: $f\_\_acceleration \in C0(RRealPlus \times S, S)$
axm16: $\forall t0 \cdot t0 \in RRealPlus \Rightarrow lipschitzContinuous(S, S, partial2(f\_\_stable, t0))$
axm17: $\forall t0 \cdot t0 \in RRealPlus \Rightarrow lipschitzContinuous(S, S, partial2(f\_\_acceleration, t0))$
axm22: $eod \in (RRealPlus \times RRealPlus \rightarrow RRealPlus)$
axm21: $eod = (\lambda ti \mapsto vi \cdot ti \in RRealPlus \land vi \in RRealPlus \mid plus(divide(vi \mapsto b) \mapsto ti))$
axm20:
  $\forall eta1, eta2, t\_\_init, v\_\_init, x\_\_init \cdot$
        $t\_\_init \in RRealPlus \land v\_\_init \in RRealPlus \land x\_\_init \in RReal \land$
    $eta1 \in Closed2Closed(t\_\_init, eod(t\_\_init \mapsto v\_\_init)) \rightarrow S \land$
    $solutionOf($
        $Closed2Closed(t\_\_init, eod(t\_\_init \mapsto v\_\_init)), eta1,$
        $ode($
            $(\lambda t\_ \mapsto (v\_ \mapsto x\_) \cdot t\_ \in RRealPlus \land (v\_ \mapsto x\_) \in S \land (t\_ \mapsto eod(t\_\_init \mapsto v\_\_init) \in lt)$
                $\mid (uminus(b) \mapsto v\_)),$
            $(v\_\_init \mapsto x\_\_init),$
            $t\_\_init$
        $)$
    $) \land$
    $eta2 \in Closed2Infinity(eod(t\_\_init \mapsto v\_\_init)) \rightarrow S \land$
    $solutionOf($
        $Closed2Infinity(eod(t\_\_init \mapsto v\_\_init)), eta2,$
        $ode($
          $(\lambda t\_ \mapsto (v\_ \mapsto x\_) \cdot t\_ \in RRealPlus \land (v\_ \mapsto x\_) \in S \land (t\_ \mapsto eod(t\_\_init \mapsto v\_\_init) \in geq)$
                $\mid (Rzero \mapsto v\_)),$
            $eta1(eod(t\_\_init \mapsto v\_\_init)),$
            $eod(t\_\_init \mapsto v\_\_init)$
        $)$
    $) \Rightarrow$
        $solutionOf($
            $Closed2Infinity(t\_\_init), eta1 \cup eta2,$
                        $ode($
                $f\_\_deceleration(t\_\_init \mapsto v\_\_init),$
                $(v\_\_init \mapsto x\_\_init),$
                $t\_\_init$
            $)$
        $)$
axm153:
  $\forall t\_\_init, v\_\_init, x\_\_init \cdot$
    $t\_\_init \in RRealPlus \land v\_\_init \in RRealPlus \land x\_\_init \in RReal \Rightarrow$
        $Solvable($
            $Closed2Infinity(t\_\_init),$
            $ode($
                $f\_\_deceleration(t\_\_init \mapsto v\_\_init),$
                $(v\_\_init \mapsto x\_\_init),$
                $t\_\_init$
            $)$
        $)$
END
```

Machine

Listings B.4: Automatic brake machine

```
MACHINE
   Autobrake
REFINES
   Generic
SEES
   AutobrakeCtx
VARIABLES  $t$, $x\_s$, $v$, $x$
INVARIANTS
   inv1: $v \in RReal \mapsto RReal$
   inv2: $Closed2Closed(Rzero, t) \subseteq dom(v)$
   inv3: $x \in RReal \mapsto RReal$
   inv4: $Closed2Closed(Rzero, t) \subseteq dom(x)$
   inv5: $x\_p = bind(v, x)$
   inv6: $boundedBy(Closed2Closed(Rzero, t), v, Rzero, Vmax)$
   inv7: $\forall t\_ \cdot t\_ \in Closed2Closed(Rzero, t) \wedge x(t\_) \mapsto SP \in geq \Rightarrow x\_s = stopped$
EVENTS
   INITIALISATION
   WITH
      $x\_p'$: $x\_p' = bind(v', x')$
   THEN
      act1: $t := Rzero$
      act2: $v, x :| v' = \{Rzero \mapsto v0\} \wedge x' = \{Rzero \mapsto x0\}$
      act4: $x\_s := stabilizing$
   END

   Behave
   REFINES  $Behave$
   ANY   $tp$, $e$
   WHERE
      grd0: $tp \in RRealPlus \wedge t \mapsto tp \in lt$
      grd1: $e \in DE(S)$
      grd2: $Solvable(Closed2Closed(t, tp), e)$
   WITH
      $x\_p'$: $x\_p' = bind(v', x')$
      $Inv$: $Inv = S$
   THEN
      act1:
         $t, x, v :|$
            $t' = tp \wedge$
            $x' \in RRealPlus \rightarrow RReal \wedge v' \in RRealPlus \rightarrow RReal \wedge$
            $CBAPsolutionOf(t, t', bind(v, x), bind(v', x'), e, S)$
   END

   ctrl_transition_accelerate
   REFINES  $Transition$
   WHERE
```

```
    grd1 :  plus(x(t) ↦ divide(times(v(t) ↦ v(t)) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt
WITH
    s :  s = {accelerating}
THEN
    act1 :  x_s := accelerating
END

ctrl_transition_stabilize
REFINES  Transition
WHERE
    grd1 :  plus(x(t) ↦ divide(times(v(t) ↦ v(t)) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt
WITH
    s :  s = {stabilizing}
THEN
    act1 :  x_s := stabilizing
END

ctrl_transition_brake
REFINES  Transition
WHERE
    grd1 :  plus(x(t) ↦ divide(times(v(t) ↦ v(t)) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt
WITH
    s :  s = {braking}
THEN
    act1 :  x_s := braking
END

ctrl_sense_near_stop
REFINES  Sense
WHERE
    grd1 :  plus(x(t) ↦ divide(times(v(t) ↦ v(t)) ↦ times(Rtwo ↦ b))) ↦ SP ∈ geq
    grd2 :  v(t) ↦ Rzero ∈ gt
WITH
    s :  s = {nearing_stop}
    p :  p = STATES × RReal × {v_ ↦ x_ |
        plus(x_ ↦ divide(times(v_ ↦ v_) ↦ times(Rtwo ↦ b))) ↦ SP ∈ geq ∧ v_ ↦ Rzero ∈ gt}
THEN
    act1 :  x_s := nearing_stop
END

ctrl_sense_stopping
REFINES  Sense
WHERE
    grd1 :  v(t) = Rzero
WITH
    s :  s = {stabilizing, stopped}
    p :  p = STATES × RReal × {v_ ↦ x_ | v_ = Rzero ∧ x_ ∈ RReal}
THEN
    act1 :
        x_s :|
        $(x\_s = nearing\_stop \Rightarrow x\_s' = stopped) \land$
        $(x\_s \neq nearing\_stop \Rightarrow x\_s' = stabilizing)$
END

ctrl_actuate_brake
REFINES  Actuate
ANY  $tp$
WHERE
  grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
  grd1 :  $x\_s \in \{braking, nearing\_stop\}$
WITH
  $e$ :  $e = ode(f\_deceleration(t \mapsto v(t)), v(t) \mapsto x(t), t)$
  $s$ :  $s = \{braking, nearing\_stop\}$
  $x\_p'$ :  $x\_p' = bind(v', x')$
  $Inv$ :  $Inv = \{(v\_ \mapsto x\_) \mid x\_ \in RReal \land Rzero \mapsto v\_ \in leq\}$
THEN
  act1 :
    $t, v, x :|$
      $t' = tp \land$
      $v' \in RReal \pfun RReal \land Closed2Closed(Rzero, t') \subseteq dom(v') \land$
      $x' \in RReal \pfun RReal \land Closed2Closed(Rzero, t') \subseteq dom(x') \land$
      $CBAPsolutionOf($
        $t, t', bind(v, x), bind(v', x'),$
        $ode(f\_deceleration(t \mapsto v(t)), v(t) \mapsto x(t), t),$
        $\{(v\_ \mapsto x\_) \mid x\_ \in RReal \land Rzero \mapsto v\_ \in leq\}$
      $)$
END

ctrl_actuate_stabilize
REFINES  Actuate
ANY  $tp$
WHERE
  grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
  grd1 :  $x\_s \in \{stabilizing, stopped\}$
WITH
  $e$ :  $e = ode(f\_stable, v(t) \mapsto x(t), t)$
  $s$ :  $s = \{stabilizing, stopped\}$
  $x\_p'$ :  $x\_p' = bind(v', x')$
  $Inv$ :
    $Inv = \{(v\_ \mapsto x\_) \mid v\_ \in RReal \land x\_ \in RReal \land$
      $plus(x\_ \mapsto divide(times(v\_ \mapsto v\_) \mapsto times(Rtwo \mapsto b))) \mapsto SP \in lt\}$
THEN
  act1 :
    $t, v, x :|$
      $t' = tp \land$
      $v' \in RReal \pfun RReal \land Closed2Closed(Rzero, t') \subseteq dom(v') \land$
      $x' \in RReal \pfun RReal \land Closed2Closed(Rzero, t') \subseteq dom(x') \land$
      $CBAPsolutionOf($
        $t, t', bind(v, x), bind(v', x'),$
        $ode(f\_stable, v(t) \mapsto x(t), t),$
```

```
            {(v_ ↦ x_) | v_ ∈ RReal ∧ x_ ∈ RReal∧
                plus(x_ ↦ divide(times(v_ ↦ v_) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt}
          )
  END

  ctrl_actuate_accelerate
  REFINES  Actuate
  ANY   tp
  WHERE
     grd0 :  tp ∈ RRealPlus ∧ t ↦ tp ∈ lt
     grd1 :  x_s = accelerating
  WITH
     e :   e = ode(f_acceleration, v(t) ↦ x(t), t)
     s :   s = {accelerating}
     x_p′ :   x_p′ = bind(v′, x′)
     Inv :
        Inv = {(v_ ↦ x_) | v_ ∈ RReal ∧ x_ ∈ RReal∧
           plus(x_ ↦ divide(times(v_ ↦ v_) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt∧
           v_ ↦ Vmax ∈ leq}
  THEN
     act1 :
        t, v, x :|
           t′ = tp∧
           v′ ∈ RReal ⇸ RReal ∧ Closed2Closed(Rzero, t′) ⊆ dom(v′)∧
           x′ ∈ RReal ⇸ RReal ∧ Closed2Closed(Rzero, t′) ⊆ dom(x′)∧
           CBAPsolutionOf(
              t, t′, bind(v, x), bind(v′, x′),
              ode(f_acceleration, v(t) ↦ x(t), t),
              {(v_ ↦ x_) | v_ ∈ RReal ∧ x_ ∈ RReal∧
                  plus(x_ ↦ divide(times(v_ ↦ v_) ↦ times(Rtwo ↦ b))) ↦ SP ∈ lt∧
                  v_ ↦ Vmax ∈ leq}
           )
  END

END
```

B.3 Signalised Left-Turn Assist

This section presents the model for the signalised left-turn assist case study, developed in Section 5.4.2.

Context

Listings B.5: SLTA context

```
CONTEXT
   LeftTurnAssistCtx
EXTENDS
   GenericCtx
```

CONSTANTS

```
waiting
turning
passed
Amax
Amin
B
k
q
Vmax
Tsv
Tpov
Tto0
Ttovmax
fsv1_decelerate
fsv1_accelerate
fsv1_accelerate_min
fsv1_stable
fsv2_speed
fpov_speed
f_decelerate
f_accelerate
f_accelerate_min
f_stable
ppov_init
vpov_init
```

AXIOMS

axm1 : $partition(STATES, \{waiting\}, \{turning\}, \{passed\})$
axm2 : $Amax \in RRealPlusStar$
axm3 : $Amin \in RRealPlusStar$
axm32 : $Amin \mapsto Amax \in lt$
axm4 : $B \in RRealPlusStar$
axm5 : $k \in RRealPlusStar$
axm6 : $q \in RRealPlusStar$
axm7 : $Vmax \in RRealPlusStar$
axm8 : $Tsv = (\lambda asv \mapsto vsv \mapsto psv \cdot$
$asv \in RReal \wedge Rzero \mapsto asv \in lt \wedge vsv \in RRealPlus \wedge psv \in RRealPlus \wedge psv \mapsto q \in leq \mid$
$divide(plus(uminus(vsv) \mapsto sqrt(plus(times(vsv \mapsto vsv) \mapsto times(times(Rtwo \mapsto asv) \mapsto$
$minus(q \mapsto psv)))))$
$\mapsto asv)$
$)$
axm9 : $Tsv \in (RReal \times RReal \times RReal) \mathrel{+\!\!\!\!\to} RReal$
axm10 : $Tpov = (\lambda ppov \cdot ppov \in RReal \mid divide(minus(ppov \mapsto k) \mapsto Vmax))$
axm11 : $Tpov \in RReal \to RReal$
axm12 : $Tto0 = (\lambda a\_ \mapsto v0\_ \cdot a\_ \in RReal \wedge a\_ \mapsto Rzero \in lt \wedge v0\_ \in RRealPlus \mid$
$uminus(divide(v0\_ \mapsto a\_)))$
axm13 : $Ttovmax = (\lambda a\_ \mapsto v0\_ \cdot$
$a\_ \in RReal \wedge a\_ \mapsto Rzero \in gt \wedge v0\_ \in RRealPlus \wedge v0\_ \mapsto Vmax \in leq$
$\mid divide(minus(Vmax \mapsto v0\_) \mapsto a\_))$
axm14 : $fsv1\_decelerate = (\lambda a\_ \mapsto v0\_ \cdot$

$a\_ \in RReal \land a\_ \mapsto Rzero \in lt \land uminus(B) \mapsto a\_ \in leq \land v0\_ \in RRealPlus \mid$
$\quad untilF(Rzero, fcste(RReal \times S, a\_), Tto0(a\_ \mapsto v0\_), fcste(RReal \times S, Rzero))$
$)$
axm15 : $fsv1\_accelerate = (\lambda a\_ \mapsto v0\_ \cdot$
$a\_ \in RReal \land a\_ \mapsto Rzero \in gt \land v0\_ \in RRealPlus \land v0\_ \mapsto Vmax \in leq \mid$
$\quad untilF(Rzero, fcste(RReal \times S, a\_), Ttovmax(a\_ \mapsto v0\_), fcste(RReal \times S, Rzero))$
$)$
axm16 : $fsv1\_accelerate\_min = (\lambda a\_ \mapsto v0\_ \cdot$
$a\_ \in RReal \land a\_ \mapsto Amin \in gt \land v0\_ \in RRealPlus \land v0\_ \mapsto Vmax \in leq \mid$
$\quad untilF(Rzero, fcste(RReal \times S, a\_), Ttovmax(a\_ \mapsto v0\_), fcste(RReal \times S, Rzero))$
$)$
axm17 : $fsv1\_stable = (\lambda t\_ \mapsto eta\_ \cdot$
$t\_ \in RRealPlus \land eta\_ \in S \mid Rzero)$
axm18 : $fsv2\_speed = (\lambda t\_ \mapsto (vsv\_ \mapsto psv\_ \mapsto ppov\_) \cdot$
$t\_ \in RRealPlus \land (vsv\_ \mapsto psv\_ \mapsto ppov\_) \in S \mid vsv\_)$
axm22 : $fpov\_speed = (\lambda v\_ \cdot$
$v\_ \in RReal \land v\_ \in Closed2Closed(uminus(Vmax), Rzero) \mid$
$\quad fcste(RReal \times S, v\_)$
$)$
axm23 : $f\_decelerate = (\lambda vpov\_ \mapsto a\_ \mapsto v0\_ \cdot$
$vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \land a\_ \in Closed2Open(uminus(B), Rzero) \land$
$\quad v0\_ \in RRealPlus \mid$
$\quad\quad bind(bind(fsv1\_decelerate(a\_ \mapsto v0\_), fsv2\_speed), fpov\_speed(vpov\_))$
$)$
axm24 : $f\_accelerate = (\lambda vpov\_ \mapsto a\_ \mapsto v0\_ \cdot$
$vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \land a\_ \in Open2Closed(Rzero, Amax) \land$
$\quad v0\_ \in RRealPlus \land v0\_ \mapsto Vmax \in leq \mid$
$\quad\quad bind(bind(fsv1\_accelerate(a\_ \mapsto v0\_), fsv2\_speed), fpov\_speed(vpov\_))$
$)$
axm25 : $f\_accelerate\_min = (\lambda vpov\_ \mapsto a\_ \mapsto v0\_ \cdot$
$vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \land a\_ \in Open2Closed(Amin, Amax) \land$
$\quad v0\_ \in RRealPlus \land v0\_ \mapsto Vmax \in leq \mid$
$\quad\quad bind(bind(fsv1\_accelerate\_min(a\_ \mapsto v0\_), fsv2\_speed), fpov\_speed(vpov\_))$
$)$
axm26 : $f\_stable = (\lambda vpov\_ \cdot$
$vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \mid$
$\quad bind(bind(fsv1\_stable, fsv2\_speed), fpov\_speed(vpov\_))$
$)$
axm33 :
$\forall vpov\_, a\_, v0\_ \cdot$
$\quad vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \land a\_ \in Closed2Open(uminus(B), Rzero) \land$
$\quad\quad v0\_ \in RRealPlus \Rightarrow$
$\quad\quad\quad partialPiecewiseContinuous($
$\quad\quad\quad\quad \{Closed2Open(Rzero, Tto0(a\_ \mapsto v0\_)), Closed2Infinity(Tto0(a\_ \mapsto v0\_))\},$
$\quad\quad\quad\quad S, S,$
$\quad\quad\quad\quad f\_decelerate(vpov\_ \mapsto a\_ \mapsto v0\_)$
$\quad\quad\quad)$
axm34 :
$\forall vpov\_, a\_, v0\_ \cdot$
$\quad vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \land a\_ \in Open2Closed(Rzero, Amax) \land$

```
        $v0\_ \in RRealPlus \wedge v0\_ \mapsto Vmax \in leq \Rightarrow$
          $partialPiecewiseContinuous($
            $\{Closed2Open(Rzero, Ttovmax(a\_ \mapsto v0\_)), Closed2Infinity(Ttovmax(a\_ \mapsto v0\_))\},$
            $S, S,$
            $f\_accelerate(vpov\_ \mapsto a\_ \mapsto v0\_)$
          $)$
  axm35 :
    $\forall vpov\_, a\_, v0\_\cdot$
      $vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \wedge a\_ \in Closed2Closed(Amin, Amax) \wedge$
        $v0\_ \in RRealPlus \wedge v0\_ \mapsto Vmax \in leq \Rightarrow$
          $partialPiecewiseContinuous($
            $\{Closed2Open(Rzero, Ttovmax(a\_ \mapsto v0\_)), Closed2Infinity(Ttovmax(a\_ \mapsto v0\_))\},$
            $S, S,$
            $f\_accelerate\_min(vpov\_ \mapsto a\_ \mapsto v0\_)$
          $)$
  axm36 :
    $\forall vpov\_\cdot$
      $vpov\_ \in Closed2Closed(uminus(Vmax), Rzero) \Rightarrow$
        $f\_stable(vpov\_) \in C0(RRealPlus \times S, S)$
  axm37 :  $ppov\_init \in RReal$
  axm38 :  $ppov\_init \mapsto k \in gt$
  axm39 :  $vpov\_init \in Closed2Closed(uminus(Vmax), Rzero)$
END
```

Machine

Listings B.6: SLTA machine

```
MACHINE
  LeftTurnAssist
REFINES
  Generic
SEES
  LeftTurnAssistCtx
VARIABLES  $t$ , $x\_s$ , $ppov$ , $psv$ , $vsv$ , $vpov$ , $asv$
INVARIANTS
  inv1 :  $ppov \in RReal \nrightarrow RReal$
  inv2 :  $Closed2Closed(Rzero, t) \subseteq dom(ppov)$
  inv5 :  $vsv \in RReal \nrightarrow RReal$
  inv3 :  $psv \in RReal \nrightarrow RReal$
  inv4 :  $Closed2Closed(Rzero, t) \subseteq dom(psv)$
  inv6 :  $Closed2Closed(Rzero, t) \subseteq dom(vsv)$
  inv7 :  $vpov \in Closed2Closed(uminus(Vmax), Rzero)$
  inv8 :  $asv \in Closed2Closed(uminus(B), Amax)$
  inv9 :  $x\_p = bind(bind(vsv, psv), ppov)$
  inv10 :  $\forall t\_ \cdot t\_ \in Closed2Closed(Rzero, t) \wedge ppov(t\_) \mapsto k \in lt \Rightarrow$
    $(psv(t\_) \mapsto Rzero \in leq \vee psv(t\_) \mapsto q \in geq)$
EVENTS
  INITIALISATION
  WITH
```

```
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
THEN
    act1 :  $t := Rzero$
    act2 :  $vsv, psv, ppov := \{Rzero \mapsto Rzero\}, \{Rzero \mapsto Rzero\}, \{Rzero \mapsto ppov\_init\}$
    act3 :  $x\_s := waiting$
    act4 :  $vpov := vpov\_init$
    act5 :  $asv := Rzero$
END

**Behave**
REFINES  Behave
ANY  $e$ ,  $tp$ ,  $v$
WHERE
    grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
    grd1 :  $e \in DE(S)$
    grd2 :  $Solvable(Closed2Closed(t, tp), e)$
    grd3 :  $v \in Closed2Closed(Rzero, Vmax)$
WITH
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
    $Inv$ :  $Inv = S$
THEN
    act1 :
        $t, vsv, psv, ppov :\mid$
            $t' = tp \land$
            $vsv' \in RReal \twoheadrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(vsv') \land$
            $psv' \in RReal \twoheadrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(psv') \land$
            $ppov' \in RReal \twoheadrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(ppov') \land$
            $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'), e, S)$
    act2 :  $vpov := uminus(v)$
END

**ctrl_transition_attempt_turn**
REFINES  Transition
WHERE
    grd1 :  $x\_s = waiting$
    grd2 :  $Tsv(Amin \mapsto vsv(t) \mapsto psv(t)) \mapsto Tpov(ppov(t)) \in lt$
WITH
    $s$ :  $s = \{turning\}$
THEN
    act1 :  $x\_s := turning$
END

**ctrl_sense_turn_end**
REFINES  Sense
WHERE
    grd1 :  $psv(t) \mapsto q \in geq$
WITH
    $s$ :  $s = \{passed\}$
    $p$ :  $p = STATES \times RReal \times \{vsv\_ \mapsto psv\_ \mapsto ppov\_ \mid$
        $vsv\_ \in RReal \land psv\_ \mapsto q \in geq \land ppov\_ \in RReal\}$
```

```
THEN
    act1 :  $x\_s := passed$
END

ctrl__actuate__waiting
REFINES  Actuate
ANY  $tp$
WHERE
    grd0 :  $tp \in RRealPlus \wedge t \mapsto tp \in lt$
    grd1 :  $x\_s = waiting$
WITH
    $e$ :  $e = ode(f\_stable(vpov), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t)$
    $s$ :  $s = \{waiting\}$
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
    $Inv$ :  $Inv = S$
THEN
    act1 :
       $t, vsv, psv, ppov :|$
          $t' = tp \wedge$
          $vsv' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(vsv') \wedge$
          $psv' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(psv') \wedge$
          $ppov' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(ppov') \wedge$
          $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'),$
             $ode(f\_stable(vpov), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t),$
          $S)$
END

ctrl__actuate__turning
REFINES  Actuate
ANY  $tp$ ,  $a$
WHERE
    grd0 :  $tp \in RRealPlus \wedge t \mapsto tp \in lt$
    grd1 :  $x\_s = turning$
    grd2 :  $a \in Closed2Closed(Amin, Amax)$
WITH
    $e$ :  $e = ode(f\_accelerate\_min(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t)$
    $s$ :  $s = \{turning\}$
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
    $Inv$ :  $Inv = \{vsv\_ \mapsto psv\_ \mapsto ppov\_ \mid vsv\_ \in RReal \wedge psv\_ \mapsto q \in leq \wedge ppov\_ \in RReal\}$
THEN
    act1 :
       $t, vsv, psv, ppov :|$
          $t' = tp \wedge$
          $vsv' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(vsv') \wedge$
          $psv' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(psv') \wedge$
          $ppov' \in RReal \pfun RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(ppov') \wedge$
          $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'),$
             $ode(f\_accelerate\_min(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t),$
             $\{vsv\_ \mapsto psv\_ \mapsto ppov\_ \mid vsv\_ \in RReal \wedge psv\_ \mapsto q \in leq \wedge ppov\_ \in RReal\}$
          $)$
    act2 :  $asv := a$
END

ctrl__actuate__passed__stable
REFINES  $Actuate$
ANY   $tp$
WHERE
    grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
    grd1 :  $x\_s = turning$
WITH
    $e$ :  $e = ode(f\_stable(vpov), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t)$
    $s$ :  $s = \{passed\}$
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
    $Inv$ :  $Inv = S$
THEN
    act1 :
        $t, vsv, psv, ppov :|$
            $t' = tp \land$
            $vsv' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(vsv') \land$
            $psv' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(psv') \land$
            $ppov' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(ppov') \land$
            $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'),$
                $ode(f\_stable(vpov), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t),$
            $S)$
END

ctrl__actuate__passed__accelerate
REFINES  $Actuate$
ANY   $tp$ ,  $a$
WHERE
    grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
    grd1 :  $x\_s = turning$
    grd2 :  $a \in Open2Closed(Rzero, Amax)$
WITH
    $e$ :  $e = ode(f\_accelerate(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t)$
    $s$ :  $s = \{passed\}$
    $x\_p'$ :  $x\_p' = bind(bind(vsv', psv'), ppov')$
    $Inv$ :  $Inv = S$
THEN
    act1 :
        $t, vsv, psv, ppov :|$
            $t' = tp \land$
            $vsv' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(vsv') \land$
            $psv' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(psv') \land$
            $ppov' \in RReal \rightarrow\!\!\!\!\!\!+ RReal \land Closed2Closed(Rzero, t') \subseteq dom(ppov') \land$
            $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'),$
                $ode(f\_accelerate(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t),$
            $S)$
END
```

```
  ctrl__actuate__passed__decelerate
  REFINES  Actuate
  ANY  tp ,  a
  WHERE
      grd0 :  $tp \in RRealPlus \land t \mapsto tp \in lt$
      grd1 :  $x\_\_s = turning$
      grd2 :  $a \in Closed2Open(uminus(B), Rzero)$
  WITH
      e :  $e = ode(f\_\_decelerate(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t)$
      s :  $s = \{passed\}$
      x__p′ :  $x\_\_p' = bind(bind(vsv', psv'), ppov')$
      Inv :  $Inv = S$
  THEN
      act1 :
         $t, vsv, psv, ppov :|$
            $t' = tp \land$
            $vsv' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(vsv') \land$
            $psv' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(psv') \land$
            $ppov' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(ppov') \land$
            $CBAPsolutionOf(t, t', bind(bind(vsv, psv), ppov), bind(bind(vsv', psv'), ppov'),$
               $ode(f\_\_decelerate(a \mapsto vpov \mapsto vsv(t)), (vsv(t) \mapsto psv(t) \mapsto ppov(t)), t),$
            $S)$
  END

END
```

B.4 Tank Case Studies

This section presents the models for the liquid tank case studies developed in Section 6.5, including domain theories of flow and valves.

The models are given following the case study: abstract tank model (`WaterTank_base`) is presented first, as it serves as a base for every other model. The single-to-many case study (one controller and two tanks) is given next, followed by the many-to-many case study (2 components, each consisting of one controller and one plant).

B.4.1 Tank Theories

The theories relating to water tanks are presented here. The *Valves* theory models status and behaviour of the pumps, and the *Flow* theory contains differential equations for the tanks.

Theory of Valves

Listings B.7: Valves theory

```
THEORY
  IMPORT THEORY  LinComb
  DATA TYPES
      Status
      CONSTRUCTORS
      ValveOpen()
      ValveClosed()
    InOutValve
    CONSTRUCTORS
      InOut(in_status:Status,out_status:Status)
  OPERATORS
    openIn expression (iov: InOutValve)
    closeIn expression (iov: InOutValve)
    openOut expression (iov: InOutValve)
    closeOut expression (iov: InOutValve)
    rstatus expression (st: Status)
    in_rstatus expression (iov: InOutValve)
    out_rstatus expression (iov: InOutValve)
    InOutPossible expression ()
      direct definition
        { InOut(ValveOpen,ValveOpen),
          InOut(ValveOpen,ValveClosed),
          InOut(ValveClosed,ValveOpen),
          InOut(ValveClosed,ValveClosed) }
END
```

Theory of Flow

Listings B.8: Flow theory

```
THEORY
  IMPORT THEORY Valves
  DATA TYPES
    TankState
    CONSTRUCTORS
      Normal()
      Emptying()
      Filling()
      Stable()
  OPERATORS
    isFlow predicate (state: TankState,DR: ℙ(RReal),Phi: RRealPlus ⇸ RReal,
       Qmin: RReal,Qmax: RReal)
      well−definedness DR ⊆ dom(Phi)
    isFlowODE predicate (state: TankState,DR: ℙ(RReal),Php: RRealPlus×RReal
        ⇸ RReal,Qmin: RReal,Qmax: RReal)
      well−definedness DR×Closed2Closed(Qmin,Qmax) ⊆ dom(Php)
    isFlowEq predicate (state: TankState,DR: ℙ(RReal),eq: DE(RReal),Qmin:
       RReal,Qmax: RReal)
      well−definedness Solvable(DR,eq)
      direct definition
        ∀ Q · Q ∈ RRealPlus ⇸ RReal ∧ DR ⊆ dom(Q) ∧ solutionOf(DR,Q,eq) ⇒
            isFlow(state,DR,Q,Qmin,Qmax)
    Delta expression (delta_in: RReal ,delta_out: RReal)
      well−definedness Rzero ↦ delta_in ∈ lt,Rzero ↦ delta_out ∈ lt
      direct definition
```

```
        (λ io_ · io_ ∈ InOutValve | minus(times(delta_in ↦ in_rstatus(io_)) ↦
            times(delta_out ↦ out_rstatus(io_))))
    flowIO expression (Qmin: RReal,Qmax: RReal,delta_in: RReal,delta_out:
        RReal)
      well−definedness Rzero ↦ Qmin ∈ leq,Rzero ↦ Qmax ∈ lt,Qmin ↦ Qmax ∈
          leq,Rzero ↦ delta_in ∈ lt,Rzero ↦ delta_out ∈ lt
      direct definition
        (λ io_ · io_ ∈ InOutValve |
          (λ t0_ ↦ q0_ · t0_ ∈ RRealPlus ∧ q0_ ∈ RReal |
            (λ t_ ↦ q_ ·
              t_ ∈ RRealPlus ∧ q_ ∈ RReal ∧
              plus(times(Delta(delta_in,delta_out)(io_) ↦ minus(t_ ↦ t0_))
                  ↦ q0_) ∈ Closed2Closed(Qmin,Qmax) |
                Delta(delta_in,delta_out)(io_)
            ) ∪
            (λ t_ ↦ q_ ·
              t_ ∈ RRealPlus ∧ q_ ∈ RReal ∧
              plus(times(Delta(delta_in,delta_out)(io_) ↦ minus(t_ ↦ t0_))
                  ↦ q0_) ∉ Closed2Closed(Qmin,Qmax) |
                Rzero
            )
          )
        )
    flowIOParts expression (Qmin: RReal,Qmax: RReal,delta_in: RReal,delta_out:
        RReal,iov: InOutValve,t0: RRealPlus,q0: RReal)
      well−definedness Rzero ↦ Qmin ∈ lt,Rzero ↦ Qmax ∈ lt,Rzero ↦ delta_in
          ∈ lt,Rzero ↦ delta_out ∈ lt,Rzero ↦ q0 ∈ leq,q0 ↦ Qmax ∈ leq
      direct definition
        {
          { t_ | t_ ∈ RRealPlus ∧ plus(times(Delta(delta_in,delta_out)(iov) ↦
              minus(t_ ↦ t0)) ↦ q0) ∈ Closed2Closed(Qmin,Qmax) },
          { t_ | t_ ∈ RRealPlus ∧ plus(times(Delta(delta_in,delta_out)(iov) ↦
              minus(t_ ↦ t0)) ↦ q0) ↦ Qmin ∈ lt },
          { t_ | t_ ∈ RRealPlus ∧ Qmax ↦ plus(times(Delta(delta_in,delta_out)
              (iov) ↦ minus(t_ ↦ t0)) ↦ q0) ∈ lt }
        }
    FlowIOODE expression (Qmin: RReal,Qmax: RReal,delta_in: RReal,delta_out:
        RReal)
      well−definedness Rzero ↦ Qmin ∈ leq,Rzero ↦ Qmax ∈ lt,Qmin ↦ Qmax ∈
          leq,Rzero ↦ delta_in ∈ lt,Rzero ↦ delta_out ∈ lt
      direct definition
        (λ io_ · io_ ∈ InOutValve |
          (λ t0_ ↦ q0_ · t0_ ∈ RRealPlus ∧ q0_ ∈ RReal |
            ode(flowIO(Qmin,Qmax,delta_in,delta_out)(io_)(t0_↦q0_),q0_,t0_)
          )
        )
    TankModeChange expression (mode: TankState)
    NoFlow expression ()
      direct definition
        (λ t_↦Q_ · t_ ∈ RRealPlus ∧ Q_ ∈ RReal | Rzero)
```

```
THEOREMS
  flowode_yields_flow:
    ∀ DR, Qmin, Qmax, Phip, st, t0, q0 ·
      DR ⊆ RRealPlus ∧
      Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
      Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
      Qmin ↦ Qmax ∈ leq ∧
      st ∈ TankState ∧
      t0 ∈ DR ∧ q0 ∈ RReal ∧
      Phip ∈ RRealPlus×RReal ⇸ RReal ∧
      DR×Closed2Closed(Qmin,Qmax) ⊆ dom(Phip) ∧
      isFlowODE(st,DR,Phip,Qmin,Qmax) ∧
      Solvable(DR,ode(Phip,q0,t0))
      ⇒ (
        ∀ Phi · Phi ∈ RRealPlus ⇸ RReal ∧ DR ⊆ dom(Phi) ∧ solutionOf(DR,
            Phi,ode(Phip,q0,t0)) ⇒
          isFlow(st,DR,Phi,Qmin,Qmax)
      )
  flow_type:
    ∀ Qmin,Qmax,delta_in,delta_out,iov,t0,q0 ·
      Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
      Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
      Qmin ↦ Qmax ∈ leq ∧
      delta_in ∈ RReal ∧ Rzero ↦ delta_in ∈ lt ∧
      delta_out ∈ RReal ∧ Rzero ↦ delta_out ∈ lt ∧
      iov ∈ InOutValve ∧
      t0 ∈ RRealPlus ∧
      q0 ∈ RReal ⇒ (
        flowIO(Qmin,Qmax,delta_in,delta_out)(iov)(t0↦q0) ∈ RRealPlus×RReal
            → RReal
      )
  flow_lipschitz:
    ∀ Qmin,Qmax,delta_in,delta_out,iov,t0,q0 ·
      Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
      Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
      Qmin ↦ Qmax ∈ leq ∧
      delta_in ∈ RReal ∧ Rzero ↦ delta_in ∈ lt ∧
      delta_out ∈ RReal ∧ Rzero ↦ delta_out ∈ lt ∧
      iov ∈ InOutValve ∧
      t0 ∈ RRealPlus ∧
      q0 ∈ RReal ∧ Rzero ↦ q0 ∈ leq ∧ q0 ↦ Qmax ∈ leq ⇒ (
        ∀ t_ · t_ ∈ RRealPlus ⇒
          lipschitzContinuous(RReal,RReal,partial2(flowIO(Qmin,Qmax,
              delta_in,delta_out)(iov)(t0↦q0),t_))
      )
  flow_piecewise_continuous:
    ∀ Qmin,Qmax,delta_in,delta_out,iov,t0,q0 ·
      Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
      Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
      Qmin ↦ Qmax ∈ leq ∧
```

```
        delta_in ∈ RReal ∧ Rzero ↦ delta_in ∈ lt ∧
        delta_out ∈ RReal ∧ Rzero ↦ delta_out ∈ lt ∧
        iov ∈ InOutValve ∧
        t0 ∈ RRealPlus ∧
        q0 ∈ RReal ∧ Rzero ↦ q0 ∈ leq ∧ q0 ↦ Qmax ∈ leq ⇒ (
          partialPiecewiseContinuous(
            flowIOParts(Qmax,Qmin,delta_in,delta_out,iov,t0,q0),
            RReal,RReal,
            flowIO(Qmin,Qmax,delta_in,delta_out)(iov)(t0↦q0)
          )
        )
  flow_pw_CL_cond:
    ∀ Qmin,Qmax,delta_in,delta_out,iov,t0,q0 ·
        Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
        Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
        Qmin ↦ Qmax ∈ leq ∧
        delta_in ∈ RReal ∧ Rzero ↦ delta_in ∈ lt ∧
        delta_out ∈ RReal ∧ Rzero ↦ delta_out ∈ lt ∧
        iov ∈ InOutValve ∧
        t0 ∈ RRealPlus ∧
        q0 ∈ RReal ∧ Rzero ↦ q0 ∈ leq ∧ q0 ↦ Qmax ∈ leq ⇒ (
          PiecewiseCauchyLipschitzCondition(
            flowIOParts(Qmin,Qmax,delta_in,delta_out,iov,t0,q0),
            RReal,
            FlowIOODE(Qmin,Qmax,delta_in,delta_out)(iov)(t0↦q0)
          )
        )
  flowio_is_flowode_single_tank_policy:
    ∀ DR,Qmin,Qmax,delta_in,delta_out,state ·
        DR ⊆ RRealPlus ∧
        Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
        Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
        Qmin ↦ Qmax ∈ leq ∧
        delta_in ∈ RReal ∧ Rzero ↦ delta_in ∈ lt ∧
        delta_out ∈ RReal ∧ Rzero ↦ delta_out ∈ lt ∧
        state ∈ TankState
        ⇒ (
          ∀ iov,t0,q0 ·
            iov ∈ TankModeChange(state) ∧
            t0 ∈ DR ∧ q0 ∈ RReal ∧
            Qmin ↦ q0 ∈ leq ∧ q0 ↦ Qmax ∈ leq
            ⇒
              isFlowODE(state,DR,flowIO(Qmin,Qmax,delta_in,delta_out)(iov)(t0
                  ↦q0),Qmin,Qmax)
        )
  flow_LC:
    ∀ DR,
        Phi1, l, Q1min, Q1max,
        Phi2, m, Q2min, Q2max ·
        DR ⊆ RReal ∧
```

```
      l ∈ RReal ∧ Rzero ↦ l ∈ lt ∧
      Q1min ∈ RReal ∧ Q1max ∈ RReal ∧ Q1min ↦ Q1max ∈ leq ∧
      Rzero ↦ Q1min ∈ leq ∧ Rzero ↦ Q1max ∈ lt ∧
      Phi1 ∈ RRealPlus ⇸ RReal ∧ DR ⊆ dom(Phi1) ∧
      m ∈ RReal ∧ Rzero ↦ m ∈ lt ∧
      Q2min ∈ RReal ∧ Q2max ∈ RReal ∧ Q2min ↦ Q2max ∈ leq ∧
      Rzero ↦ Q2min ∈ leq ∧ Rzero ↦ Q2max ∈ lt ∧
      Phi2 ∈ RRealPlus ⇸ RReal ∧ DR ⊆ dom(Phi2) ⇒ (
        (∀ st · st ∈ TankState ⇒ isFlow(st,DR,Phi1,Q1min,Q1max) ∧ isFlow(st
            ,DR,Phi2,Q2min,Q2max))
      ⇔
        (∀ st · st ∈ TankState ⇒ isFlow(
                    st,
                    DR,
                    LinComb2(l,Phi1,m,Phi2),
                    sLinComb2(l,Q1min,m,Q2min),
                    sLinComb2(l,Q1max,m,Q2max)
                    )
        )
      )
  flowODE_LC:
    ∀ DR,
      Phip1, l, Q1min, Q1max,
      Phip2, m, Q2min, Q2max ·
      DR ⊆ RReal ∧
      Q1min ∈ RReal ∧ Q1max ∈ RReal ∧ Q1min ↦ Q1max ∈ leq ∧
      Rzero ↦ Q1min ∈ leq ∧ Rzero ↦ Q1max ∈ lt ∧
      Phip1 ∈ RRealPlus×RReal ⇸ RReal ∧ DR×Closed2Closed(Q1min,Q1max) ⊆
          dom(Phip1) ∧
      m ∈ RReal ∧ Rzero ↦ m ∈ lt ∧
      Q2min ∈ RReal ∧ Q2max ∈ RReal ∧ Q2min ↦ Q2max ∈ leq ∧
      Rzero ↦ Q2min ∈ leq ∧ Rzero ↦ Q2max ∈ lt ∧
      Phip2 ∈ RRealPlus×RReal ⇸ RReal ∧ DR×Closed2Closed(Q2min,Q2max) ⊆
          dom(Phip2) ⇒ (
        (∀ st · st ∈ TankState ⇒ isFlowODE(st,DR,Phip1,Q1min,Q1max) ∧
            isFlowODE(st,DR,Phip2,Q2min,Q2max))
      ⇔
        (∀ st · st ∈ TankState ⇒ isFlowODE(
                    st,
                    DR,
                    LinComb2(l,Phip1,m,Phip2),
                    sLinComb2(l,Q1min,m,Q2min),
                    sLinComb2(l,Q1max,m,Q2max)
                    )
        )
      )
  flowode_is_floweq:
    ∀ DR,Qmin,Qmax,Phip,st,t0,q0 ·
      DR ⊆ RRealPlus ∧
      Qmin ∈ RReal ∧ Rzero ↦ Qmin ∈ leq ∧
```

```
          Qmax ∈ RReal ∧ Rzero ↦ Qmax ∈ lt ∧
          Qmin ↦ Qmax ∈ leq ∧
          st ∈ TankState ∧
          t0 ∈ DR ∧ q0 ∈ RReal ∧
          Phip ∈ RRealPlus×RReal ⇸ RReal ∧ DR×Closed2Closed(Qmin,Qmax) ⊆ dom(
              Phip) ∧
          isFlowODE(st,DR,Phip,Qmin,Qmax) ∧
          Solvable(DR,ode(Phip,q0,t0))
          ⇒
            isFlowEq(st,DR,ode(Phip,q0,t0),Qmin,Qmax)
     thm1 :
        ⊤
END
```

B.4.2 Abstract Tank Model

This section presents the model of the abstract tank, that serves as a base for other models.

Context

Listings B.9: Abstract tank context

```
CONTEXT
   WaterTank_base_Ctx
EXTENDS
   GenericCtx
CONSTANTS
   Vmax
   V0
   delta_in
   delta_out
   Vlow
   Vhigh
   dv_min
   dv_max
AXIOMS
   axm10 :  Vmax ∈ RReal
   axm11 :  Rzero ↦ Vmax ∈ lt
   axm20 :  V0 ∈ RReal
   axm21 :  Rzero ↦ V0 ∈ lt
   axm12 :  V0 ↦ Vmax ∈ lt
   axm30 :  delta_in ∈ RReal
   axm31 :  Rzero ↦ delta_in ∈ lt
   axm32 :  delta_out ∈ RReal
   axm33 :  Rzero ↦ delta_out ∈ lt
   axm34 :  delta_out ↦ delta_in ∈ lt
   axm40 :  Vlow ∈ RReal
   axm41 :  Rzero ↦ Vlow ∈ lt
   axm42 :  Vlow ↦ Vmax ∈ lt
   axm43 :  Vhigh ∈ RReal
   axm44 :  Rzero ↦ Vhigh ∈ lt
   axm45 :  Vhigh ↦ Vmax ∈ lt
   axm46 :  Vlow ↦ Vhigh ∈ lt
   axm47 :  Vlow ↦ V0 ∈ leq
   axm48 :  V0 ↦ Vhigh ∈ leq
   axm50 :  dv_min ∈ RReal
   axm51 :  dv_min ↦ Rzero ∈ lt
   axm52 :  dv_max ∈ RReal
   axm53 :  Rzero ↦ dv_max ∈ lt
   axm54 :  dv_min ↦ dv_max ∈ lt
END
```

Machine

Listings B.10: Abstract tank machine

```
MACHINE
   WaterTank_base
REFINES
   Generic
SEES
   WaterTank_base_Ctx
VARIABLES   t ,  V ,  x_s
INVARIANTS
   inv1 :  V ∈ RRealPlus ⇸ S
   inv2 :  Closed2Closed(Rzero, t) ⊆ dom(V)
   inv3 :  V = x_p
   inv4 :  V ∈ D1(Closed2Infinity(t), RReal)∧
   boundedBy(Closed2Infinity(t), Der(Closed2Infinity(t), RReal, V), dv_min, dv_max)
   inv5 :  boundedBy(RRealPlus, V, Vlow, Vhigh)
EVENTS
   INITIALISATION
   WITH
      x_p′ :  x_p′ = V′
   THEN
      act1 :  t := Rzero
      act2 :  V := {Rzero ↦ V0}
      act3 :  x_s := Stable
   END

   Behave
   REFINES  Behave
   ANY   e ,  tp
   WHERE
      grd1 :  e ∈ DE(S)
      grd2 :  Solvable(Closed2Infinity(t), e)
      grd3 :  Vlow ↦ V(t) ∈ lt
      grd4 :  Vhigh ↦ V(t) ∈ gt
      grd5 :  tp ∈ RRealPlus
      grd6 :  t ↦ tp ∈ lt∧
```

```
    $CBAPsolutionOfFIS(t, tp, V, e, \{V\_ \mid V\_ \in RReal \land Vlow \mapsto V\_ \in lt \land Vhigh \mapsto V\_ \in gt\})$
WITH
  $x\_p'$ :  $x\_p' = V'$
  $Inv$ :  $Inv = \{V\_ \mid V\_ \in RReal \land Vlow \mapsto V\_ \in lt \land Vhigh \mapsto V\_ \in gt\}$
THEN
  act1 :
    $t, V :|$
      $t' = tp \land$
      $V' \in RRealPlus \nrightarrow S \land Closed2Closed(Rzero, t') \subseteq dom(V') \land$
      $CBAPsolutionOf($
        $t, t',$
        $V, V',$
        $e,$
        $\{V\_ \mid V\_ \in RReal \land Vlow \mapsto V\_ \in lt \land Vhigh \mapsto V\_ \in gt\}$
      $)$
END

ctrl_sense_too_high
REFINES Sense
WHERE
  grd1 :  $Vhigh \mapsto V(t) \in leq$
WITH
  $s$ :  $s = \{Emptying\}$
  $p$ :  $p = STATES \times RRealPlus \times \{V\_ \mid Vhigh \mapsto V\_ \in leq\}$
THEN
  act1 :  $x\_s := Emptying$
END

ctrl_sense_too_low
REFINES Sense
WHERE
  grd1 :  $V(t) \mapsto Vlow \in leq$
WITH
  $s$ :  $s = \{Filling\}$
  $p$ :  $p = STATES \times RRealPlus \times \{V\_ \mid V\_ \mapsto Vlow \in leq\}$
THEN
  act1 :  $x\_s := Filling$
END

ctrl_transition_emptying
REFINES Transition
WHERE
  grd1 :  $Vlow \mapsto V(t) \in lt$
WITH
  $s$ :  $s = \{Emptying\}$
THEN
  act1 :  $x\_s := Emptying$
END

ctrl_transition_filling
```

```
REFINES Transition
WHERE
    grd1 :  $V(t) \mapsto Vhigh \in lt$
WITH
    s :  $s = \{Filling\}$
THEN
    act1 :  $x\_s := Filling$
END

ctrl_transition_normal
REFINES Transition
WHERE
    grd1 :  $Vlow \mapsto V(t) \in lt$
    grd2 :  $V(t) \mapsto Vhigh \in lt$
WITH
    s :  $s = \{Normal\}$
THEN
    act1 :  $x\_s := Normal$
END

ctrl_transition_stable
REFINES Transition
WHERE
    grd1 :  $Vlow \mapsto V(t) \in lt$
    grd2 :  $V(t) \mapsto Vhigh \in lt$
WITH
    s :  $s = \{Stable\}$
THEN
    act1 :  $x\_s := Stable$
END

ctrl_actuate_pumps
REFINES Actuate
ANY  e ,  ss ,  tp
WHERE
    grd1 :  $e \in DE(S)$
    grd2 :  $Solvable(Closed2Infinity(t), e)$
    grd3 :  $isFlowEq(ss, Closed2Infinity(t), e, Rzero, Vmax)$
    grd4 :  $ss \in STATES$
    grd5 :  $x\_s = ss$
    grd6 :  $Vlow \mapsto V(t) \in lt$
    grd7 :  $Vhigh \mapsto V(t) \in gt$
    grd8 :  $tp \in RRealPlus$
    grd9 :  $t \mapsto tp \in lt \wedge$
        $CBAPsolutionOfFIS(t, tp, V, e, \{V\_ \mid V\_ \in RReal \wedge Vlow \mapsto V\_ \in lt \wedge Vhigh \mapsto V\_ \in gt\})$
WITH
    x_p' :  $x\_p' = V'$
    s :  $s = \{ss\}$
    Inv :  $Inv = \{V\_ \mid V\_ \in RReal \wedge Vlow \mapsto V\_ \in lt \wedge Vhigh \mapsto V\_ \in gt\}$
THEN
```

```
    act1:
      $t, V :|$
        $t' \in RRealPlus \wedge t \mapsto t' \in lt \wedge$
        $V' \in RRealPlus \twoheadrightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V') \wedge$
        $CBAPsolutionOf($
          $t, t',$
          $V, V',$
          $e,$
          $\{V\_ \mid V\_ \in RReal \wedge Vlow \mapsto V\_ \in lt \wedge Vhigh \mapsto V\_ \in gt\}$
        $)$
  END

END
```

B.4.3 Centralised Control/S2M Tank Model

This section presents the model for the single-to-many situation (one controller with two tanks), corresponding to Section 6.5.4.

Context

Listings B.11: S2M tank context

```
CONTEXT
   WaterTank_2Tanks_Cylinder_Ctx
EXTENDS
   WaterTank_base_Ctx
CONSTANTS
   B1
   B2
   H1max
   H2max
   H10
   H20
   delta_in_h1
   delta_out_h1
   delta_in_h2
   delta_out_h2
AXIOMS
   axm1: $B1 \in RReal$
   axm2: $Rzero \mapsto B1 \in lt$
   axm3: $B2 \in RReal$
   axm4: $Rzero \mapsto B2 \in lt$
   axm5: $H1max \in RReal$
   axm6: $Rzero \mapsto H1max \in lt$
   axm7: $H2max \in RReal$
   axm8: $Rzero \mapsto H2max \in lt$
   axm9: $H10 \in RReal$
   axm10: $Rzero \mapsto H10 \in leq$
   axm11: $H10 \mapsto H1max \in leq$
   axm12 :  $H20 \in RReal$
   axm13 :  $Rzero \mapsto H20 \in leq$
   axm14 :  $H20 \mapsto H2max \in leq$
   axm15 :  $Vmax = sLinComb2(B1, H1max, B2, H2max)$
   axm16 :  $V0 = sLinComb2(B1, H10, B2, H20)$
   axm17 :  $delta\_in\_h1 \in RReal$
   axm18 :  $Rzero \mapsto delta\_in\_h1 \in lt$
   axm19 :  $delta\_out\_h1 \in RReal$
   axm20 :  $Rzero \mapsto delta\_out\_h1 \in lt$
   axm21 :  $delta\_in\_h2 \in RReal$
   axm22 :  $Rzero \mapsto delta\_in\_h2 \in lt$
   axm23 :  $delta\_out\_h2 \in RReal$
   axm24 :  $Rzero \mapsto delta\_out\_h2 \in lt$
END
```

Machine

Listings B.12: S2M tank machine

```
MACHINE
   WaterTank_1Ctrl_2Tanks_Cylinder
REFINES
   WaterTank_base
SEES
   WaterTank_2Tanks_Cylinder_Ctx
VARIABLES  t ,  x_s ,  h1 ,  h2
INVARIANTS
   inv1 :  $h1 \in RRealPlus \nrightarrow RReal$
   inv2 :  $Closed2Closed(Rzero, t) \subseteq dom(h1)$
   inv3 :  $h2 \in RRealPlus \nrightarrow RReal$
   inv4 :  $Closed2Closed(Rzero, t) \subseteq dom(h2)$
   inv5 :  $V = LinComb2(B1, h1, B2, h2)$
EVENTS
   INITIALISATION
   THEN
      act1 :  $t := Rzero$
      act2 :  $h1, h2 := \{Rzero \mapsto H10\}, \{Rzero \mapsto H20\}$
      act3 :  $x\_s := Stable$
   END

   Behave
   REFINES  Behave
   ANY   e ,  tp
   WHERE
      grd1 :  $e \in DE(S)$
      grd2 :  $Solvable(Closed2Infinity(t), e)$
      grd3 :  $Vlow \mapsto LinComb2(B1, h1, B2, h2)(t) \in lt$
      grd4 :  $Vhigh \mapsto LinComb2(B1, h1, B2, h2)(t) \in gt$
      grd5 :  $tp \in RRealPlus$
      grd6 :
        $t \mapsto tp \in lt \land CBAPsolutionOfFIS(t, tp,$
            $LinComb2(B1, h1, B2, h2),$
            $e,$
            $\{V\_ \mid V\_ \in RReal \land Vlow \mapsto V\_ \in lt \land Vhigh \mapsto V\_ \in gt\}$
        $)$
WITH
    $V'$:  $V' = LinComb2(B1, h1', B2, h2')$
THEN
    act1:
        $t, h1, h2 :|$
            $t' = tp \land$
            $h1' \in RRealPlus \nrightarrow S \land Closed2Closed(Rzero, t') \subseteq dom(h1') \land$
            $h2' \in RRealPlus \nrightarrow S \land Closed2Closed(Rzero, t') \subseteq dom(h2') \land$
            $CBAPsolutionOf($
                $t, t',$
                $LinComb2(B1, h1, B2, h2), LinComb2(B1, h1', B2, h2'),$
                $e,$
                $\{V\_ \mid V\_ \in RReal \land Vlow \mapsto V\_ \in lt \land Vhigh \mapsto V\_ \in gt\}$
            $)$
END

ctrl_sense_too_high
REFINES ctrl_sense_too_high
WHERE
    grd1:  $Vhigh \mapsto LinComb2(B1, h1, B2, h2)(t) \in leq$
THEN
    act1:  $x\_s := Emptying$
END

ctrl_sense_too_low
REFINES ctrl_sense_too_low
WHERE
    grd1:  $LinComb2(B1, h1, B2, h2)(t) \mapsto Vlow \in leq$
THEN
    act1:  $x\_s := Filling$
END

ctrl_transition_emptying
REFINES ctrl_transition_emptying
WHERE
    grd1:  $Vlow \mapsto LinComb2(B1, h1, B2, h2)(t) \in lt$
THEN
    act1:  $x\_s := Emptying$
END

ctrl_transition_filling
REFINES ctrl_transition_filling
WHERE
    grd1:  $LinComb2(B1, h1, B2, h2)(t) \mapsto Vhigh \in lt$
THEN
```

```
    act1 :  $x\_s := Filling$
END

ctrl_transition_normal
REFINES  $ctrl\_transition\_normal$
WHERE
    grd1 :  $Vlow \mapsto LinComb2(B1, h1, B2, h2)(t) \in lt$
    grd2 :  $LinComb2(B1, h1, B2, h2)(t) \mapsto Vhigh \in lt$
THEN
    act1 :  $x\_s := Normal$
END

ctrl_transition_stable
REFINES  $ctrl\_transition\_stable$
WHERE
    grd1 :  $Vlow \mapsto LinComb2(B1, h1, B2, h2)(t) \in lt$
    grd2 :  $LinComb2(B1, h1, B2, h2)(t) \mapsto Vhigh \in lt$
THEN
    act1 :  $x\_s := Stable$
END

ctrl_actuate_pumps
REFINES  $ctrl\_actuate\_pumps$
ANY   $io$ ,  $ss$ ,  $tp$
WHERE
    grd4 :  $ss \in STATES$
    grd5 :  $x\_s = ss$
    grd6 :  $io \in TankModeChange(x\_s)$
    grd7 :  $Vlow \mapsto LinComb2(B1, h1, B2, h2)(t) \in lt$
    grd8 :  $Vhigh \mapsto LinComb2(B1, h1, B2, h2)(t) \in gt$
    grd9 :  $tp \in RRealPlus$
    grd10 :
        $t \mapsto tp \in lt \wedge CBAPParallelEqFIS(t, tp,$
            $h1, FlowIOODE(Rzero, H1max, delta\_in\_h1, delta\_out\_h1)(io)(t \mapsto h1(t)),$
            $h2, FlowIOODE(Rzero, H2max, delta\_in\_h2, delta\_out\_h2)(io)(t \mapsto h2(t)),$
            $\{h1\_ \mapsto h2\_ \mid h1\_ \in RReal \wedge h2\_ \in RReal \wedge$
                $Vlow \mapsto plus(times(B1 \mapsto h1\_) \mapsto times(B2 \mapsto h2\_)) \in lt \wedge$
                $Vhigh \mapsto plus(times(B1 \mapsto h1\_) \mapsto times(B2 \mapsto h2\_)) \in gt$
            $\})$
WITH
    $V'$ :  $V' = LinComb2(B1, h1', B2, h2')$
    $e$ :
        $e \in DE(S) \wedge$
        $isFlowEq(ss, Closed2Closed(t, tp), e, Rzero, Vmax) \wedge$
        $Solvable(Closed2Closed(t, tp), e) \wedge$
        $solutionOf(Closed2Closed(t, tp), LinComb2(B1, h1', B2, h2'), e)$
THEN
    act1 :
        $t, h1, h2 :\mid$
            $t' = tp \wedge$
            $h1' \in RRealPlus \rightarrow\!\!\!\!+ S \wedge Closed2Closed(Rzero, t') \subseteq dom(h1') \wedge$
            $h2' \in RRealPlus \rightarrow\!\!\!\!+ S \wedge Closed2Closed(Rzero, t') \subseteq dom(h2') \wedge$
            $CBAPParallelEq($
               $t, t',$
               $h1, h1', FlowIOODE(Rzero, H1max, delta\_in\_h1, delta\_out\_h1)(io)(t \mapsto h1(t)),$
               $h2, h2', FlowIOODE(Rzero, H2max, delta\_in\_h2, delta\_out\_h2)(io)(t \mapsto h2(t)),$
               $\{h1\_ \mapsto h2\_ \mid h1\_ \in RReal \wedge h2\_ \in RReal \wedge$
                  $Vlow \mapsto plus(times(B1 \mapsto h1\_) \mapsto times(B2 \mapsto h2\_)) \in lt \wedge$
                  $Vhigh \mapsto plus(times(B1 \mapsto h1\_) \mapsto times(B2 \mapsto h2\_)) \in gt$
               $\}$
            $)$
   END

END
```

B.4.4 Distributed Control/M2M Tank Model

This section presents the model for the many-to-many situation (two controllers, each controlling one tank), corresponding to Section 6.5.5.

Context

Listings B.13: M2M tank context

```
CONTEXT
   WaterTank__2Ctrl__2Tanks__Ctx
EXTENDS
   WaterTank__base__Ctx
CONSTANTS
   V1max
   V2max
   V10
   V20
   Policy
AXIOMS
   axm1:  $V1max \in RReal$
   axm2:  $Rzero \mapsto V1max \in lt$
   axm3:  $V2max \in RReal$
   axm4:  $Rzero \mapsto V2max \in lt$
   axm5:  $V10 \in RReal$
   axm6:  $Rzero \mapsto V10 \in leq$
   axm7:  $V10 \mapsto V1max \in leq$
   axm8:  $V20 \in RReal$
   axm9:  $Rzero \mapsto V20 \in leq$
   axm10:  $V20 \mapsto V2max \in leq$
   axm11:  $Policy \in STATES \times STATES \times STATES$
   axm12:
      $Policy = \{$
         $(Emptying \mapsto Emptying) \mapsto Emptying,$
         $(Emptying \mapsto Stable) \mapsto Emptying,$
```

```
        (Stable ↦ Emptying) ↦ Emptying,
        (Filling ↦ Filling) ↦ Filling,
        (Filling ↦ Stable) ↦ Filling,
        (Stable ↦ Filling) ↦ Filling,
        (Stable ↦ Stable) ↦ Stable,
        (Emptying ↦ Filling) ↦ Normal,
        (Filling ↦ Emptying) ↦ Normal
    } ∪ {(x_ ↦ y_) ↦ z_ | z_ = Normal ∧ (x_ = Normal ∨ y_ = Normal)}
END
```

Machine

Listings B.14: M2M tank machine

```
MACHINE
   WaterTank_2Ctrl_2Tanks
REFINES
   WaterTank_base
SEES
   WaterTank_2Ctrl_2Tanks_Ctx
VARIABLES  t , V1 , V2 , V1_sim , V2_sim , x_s1 , x_s2 , Delta_sim_1 , Delta_sim_2
INVARIANTS
   inv1 :  V1 ∈ RRealPlus ⇸ S
   inv21 :  Closed2Closed(Rzero, t) ⊆ dom(V1)
   inv2 :  V2 ∈ RRealPlus ⇸ S
   inv22 :  Closed2Closed(Rzero, t) ⊆ dom(V2)
   inv3 :  V1_sim ∈ RRealPlus ⇸ S
   inv23 :  Closed2Closed(Rzero, t) ⊆ dom(V1_sim)
   inv4 :  V2_sim ∈ RRealPlus ⇸ S
   inv24 :  Closed2Closed(Rzero, t) ⊆ dom(V2_sim)
   inv5 :  x_s1 ∈ STATES
   inv6 :  x_s2 ∈ STATES
   inv7 :  V = fadd(V1, V2)
   inv8 :  boundedBy(RRealPlus, fadd(V1, V2_sim), Vlow, Vhigh)
   inv9 :  boundedBy(RRealPlus, fadd(V1_sim, V2), Vlow, Vhigh)
   inv10 :  Delta_sim_1 ∈ RRealPlus
   inv11 :  Delta_sim_2 ∈ RRealPlus
   inv12 :  ∀t_ · t_ ∈ RRealPlus ⇒ abs(minus(V2(t_) ↦ V2_sim(t_))) ↦ Delta_sim_2 ∈ leq
   inv13 :  ∀t_ · t_ ∈ RRealPlus ⇒ abs(minus(V1(t_) ↦ V1_sim(t_))) ↦ Delta_sim_1 ∈ leq
   inv14 :  x_s ↦ x_s1 ↦ x_s2 ∈ Policy
EVENTS
   INITIALISATION
   THEN
      act1 :  t := Rzero
      act2 :  V1, V1_sim, V2, V2_sim :=
         {Rzero ↦ V10}, {Rzero ↦ V10}, {Rzero ↦ V20}, {Rzero ↦ V20}
      act3 :  x_s1, x_s2 := Stable, Stable
      act4 :  Delta_sim_1 :∈ RRealPlus
      act5 :  Delta_sim_2 :∈ RRealPlus
   END
```

```
Behave
REFINES  Behave
ANY   e ,  tp
WHERE
   grd1 :  $e \in DE(S)$
   grd2 :  $Solvable(Closed2Closed(t, tp), e)$
   grd3 :  $Vlow \mapsto plus(V1(t) \mapsto V2(t)) \in lt$
   grd4 :  $Vhigh \mapsto plus(V1(t) \mapsto V2(t)) \in gt$
   grd5 :  $tp \in RRealPlus$
   grd6 :  $t \mapsto tp \in lt \wedge$
      $CBAPsolutionOfFIS(t, tp, fadd(V1, V2), e,$
         $\{V\_ \mid V\_ \in RReal \wedge Vlow \mapsto V\_ \in lt \wedge Vhigh \mapsto V\_ \in gt\}$
      $)$
WITH
   $V'$ :  $V' = fadd(V1', V2')$
THEN
   act1 :
      $t, V1, V2 :\mid$
         $t' = tp \wedge$
         $V1' \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V1') \wedge$
         $V2' \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V2') \wedge$
         $CBAPsolutionOf($
            $t, t',$
            $fadd(V1, V2), fadd(V1', V2'),$
            $e,$
            $\{V\_ \mid V\_ \in RReal \wedge Vlow \mapsto V\_ \in lt \wedge Vhigh \mapsto V\_ \in gt\}$
         $)$
END

ctrl_sense_too_high_1
REFINES  ctrl_sense_too_high
WHERE
   grd1 :  $plus(Vhigh \mapsto Delta\_sim\_2) \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in leq$
THEN
   act1 :  $x\_s1 := Emptying$
END

ctrl_sense_too_high_2
REFINES  ctrl_sense_too_high
WHERE
   grd1 :  $plus(Vhigh \mapsto Delta\_sim\_1) \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in leq$
THEN
   act1 :  $x\_s2 := Emptying$
END

ctrl_sense_too_low_1
REFINES  ctrl_sense_too_low
WHERE
   grd1 :  $plus(V1(t) \mapsto V2\_sim(t)) \mapsto minus(Vlow \mapsto Delta\_sim\_2) \in leq$
```

THEN
  act1 : $x\_s1 := Filling$
END

ctrl__sense__too__low__2
REFINES *ctrl__sense__too__low*
WHERE
  grd1 : $plus(V1\_sim(t) \mapsto V2(t)) \mapsto minus(Vlow \mapsto Delta\_sim\_1) \in leq$
THEN
  act1 : $x\_s2 := Filling$
END

ctrl__transition__emptying__1
REFINES *ctrl__transition__emptying*
WHERE
  grd1 : $plus(Vlow \mapsto Delta\_sim\_2) \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in lt$
THEN
  act1 : $x\_s1 := Emptying$
END

ctrl__transition__emptying__2
REFINES *ctrl__transition__emptying*
WHERE
  grd1 : $plus(Vlow \mapsto Delta\_sim\_1) \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in lt$
THEN
  act1 : $x\_s2 := Emptying$
END

ctrl__transition__filling__1
REFINES *ctrl__transition__filling*
WHERE
  grd1 : $plus(V1(t) \mapsto V2\_sim(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_2) \in lt$
THEN
  act1 : $x\_s1 := Filling$
END

ctrl__transition__filling__2
REFINES *ctrl__transition__filling*
WHERE
  grd1 : $plus(V1\_sim(t) \mapsto V2(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_1) \in lt$
THEN
  act1 : $x\_s2 := Filling$
END

ctrl__transition__normal__1
REFINES *ctrl__transition__normal*
WHERE
  grd1 : $plus(Vlow \mapsto Delta\_sim\_2) \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in lt$
  grd2 : $plus(V1(t) \mapsto V2\_sim(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_2) \in lt$
THEN

```
    act1 :  $x\_s1 := Normal$
END

ctrl_transition_normal_2
REFINES  $ctrl\_transition\_normal$
WHERE
    grd1 :  $plus(Vlow \mapsto Delta\_sim\_1) \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in lt$
    grd2 :  $plus(V1\_sim(t) \mapsto V2(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_1) \in lt$
THEN
    act1 :  $x\_s2 := Normal$
END

ctrl_transition_stable_1
REFINES  $ctrl\_transition\_stable$
WHERE
    grd1 :  $plus(Vlow \mapsto Delta\_sim\_2) \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in lt$
    grd2 :  $plus(V1(t) \mapsto V2\_sim(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_2) \in lt$
THEN
    act1 :  $x\_s1 := Stable$
END

ctrl_transition_stable_2
REFINES  $ctrl\_transition\_stable$
WHERE
    grd1 :  $plus(Vlow \mapsto Delta\_sim\_1) \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in lt$
    grd2 :  $plus(V1\_sim(t) \mapsto V2(t)) \mapsto minus(Vhigh \mapsto Delta\_sim\_1) \in lt$
THEN
    act1 :  $x\_s2 := Stable$
END

ctrl_actuate_pumps
REFINES  $ctrl\_actuate\_pumps$
ANY   $ss,\ e1,\ e2,\ ss1,\ ss2,\ fV1\_sim,\ fV2\_sim,\ tp$
WHERE
    grd01 :  $ss \in STATES$
    grd02 :  $ss \mapsto ss1 \mapsto ss2 \in Policy$
    grd11 :  $e1 \in DE(S)$
    grd12 :  $Solvable(Closed2Closed(t, tp), e1)$
    grd13 :  $isFlowEq(ss1, Closed2Closed(t, tp), e1, Rzero, V1max)$
    grd14 :  $ss1 \in STATES$
    grd15 :  $x\_s1 = ss1$
    grd16 :  $fV1\_sim \in RRealPlus \nrightarrow S$
    grd17 :  $Closed2Closed(t, tp) \subseteq dom(fV1\_sim)$
    grd18 :  $\forall V1\_ \cdot V1\_ \in RRealPlus \nrightarrow S \land Closed2Closed(t, tp) \subseteq dom(V1\_) \land$
        $solutionOf(Closed2Closed(t, tp), V1\_, e1) \Rightarrow$
            $(\forall t\_ \cdot t\_ \in Closed2Closed(t, tp) \Rightarrow abs(minus(V1\_(t) \mapsto fV1\_sim(t))) \mapsto Delta\_sim\_1 \in leq)$
    grd21 :  $e2 \in DE(S)$
    grd22 :  $Solvable(Closed2Infinity(t), e2)$
    grd23 :  $isFlowEq(ss2, Closed2Infinity(t), e2, Rzero, V2max)$
    grd24 :  $ss2 \in STATES$
```

grd25 : $x\_s2 = ss2$
grd26 : $fV2\_sim \in Closed2Infinity(t) \rightarrow S$
grd27 : $\forall V2\_ \cdot V2\_ \in Closed2Infinity(t) \rightarrow S \wedge solutionOf(Closed2Infinity(t), V2\_, e2) \Rightarrow$
$(\forall t\_ \cdot t\_ \in Closed2Infinity(t) \Rightarrow abs(minus(V2\_(t) \mapsto fV2\_sim(t))) \mapsto Delta\_sim\_2 \in leq)$
grd30 : $Vlow \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in lt$
grd31 : $Vhigh \mapsto plus(V1(t) \mapsto V2\_sim(t)) \in gt$
grd32 : $Vlow \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in lt$
grd33 : $Vhigh \mapsto plus(V1\_sim(t) \mapsto V2(t)) \in gt$
grd000 : $tp \in RRealPlus$
grd001 :
$t \mapsto tp \in lt \wedge CBAPFIS(t, tp, bind(bind(V1, V2), bind(V1\_sim, V2\_sim)),$
$\{V1\_, V2\_, V1s\_, V2s\_, V1\_p, V2\_p, V1s\_p, V2s\_p \cdot$
$V1\_ \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V1\_) \wedge$
$V2\_ \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V2\_) \wedge$
$V1s\_ \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V1s\_) \wedge$
$V2s\_ \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V2s\_) \wedge$
$V1\_p \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V1\_p) \wedge$
$V2\_p \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V2\_p) \wedge$
$V1s\_p \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V1s\_p) \wedge$
$V2s\_p \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, tp) \subseteq dom(V2s\_p) \wedge$
$solutionOf(Closed2Closed(t, tp), V1\_p, e1) \wedge$
$solutionOf(Closed2Closed(t, tp), V2\_p, e2) \wedge$
$V1s\_p = V1s\_ \wedge$
$V2s\_p = V2s\_$
$\mid bind(bind(V1\_, V2\_), bind(V1s\_, V2s\_)) \mapsto bind(bind(V1\_p, V2\_p), bind(V1s\_p, V2s\_p))$
$\},$
$\{(V1\_ \mapsto V2\_) \mapsto (V1s\_ \mapsto V2s\_) \mid$
$Vlow \mapsto plus(V1\_ \mapsto V2s\_) \in lt \wedge$
$Vhigh \mapsto plus(V1\_ \mapsto V2s\_) \in gt \wedge$
$Vlow \mapsto plus(V1s\_ \mapsto V2\_) \in lt \wedge$
$Vhigh \mapsto plus(V1s\_ \mapsto V2\_) \in gt$
$\}$
$)$

WITH

V' : $V' = fadd(V1', V2')$
e : $e \in DE(S) \wedge Solvable(Closed2Infinity(t), e) \wedge \forall ss \cdot ss \mapsto s1 \mapsto s2 \in Policy \wedge$
$isFlowEq(ss, Closed2Infinity(t), e, Rzero, Vmax) \wedge$
$(\forall V1\_, V2\_ \cdot$
$V1\_ \in Closed2Infinity(t) \rightarrow S \wedge V2\_ \in Closed2Infinity(t) \rightarrow S \wedge$
$solutionOf(Closed2Infinity(t), V1\_, e1) \wedge$
$solutionOf(Closed2Infinity(t), V2\_, e2)$
$\Rightarrow solutionOf(Closed2Infinity(t), fadd(V1\_, V2\_), e)$
$)$

THEN

act1 :
$t, V1, V2, V1\_sim, V2\_sim :\mid$
$t' = tp \wedge$
$V1' \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V1') \wedge$
$V2' \in RRealPlus \rightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V2') \wedge$
$V1\_sim' \in RRealPlus \nrightarrow S \wedge Closed2Closed(Rzero, t') \subseteq dom(V1\_sim') \wedge$

```
        $V2\_sim' \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V2\_sim') \land$
        $CBAP($
          $t, t',$
          $bind(bind(V1, V2), bind(V1\_sim, V2\_sim)),$
          $bind(bind(V1', V2'), bind(V1\_sim', V2\_sim')),$
          $\{V1\_, V2\_, V1s\_, V2s\_, V1\_p, V2\_p, V1s\_p, V2s\_p\cdot$
            $V1\_ \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V1\_) \land$
            $V2\_ \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V2\_) \land$
            $V1s\_ \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V1s\_) \land$
            $V2s\_ \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V2s\_) \land$
            $V1\_p \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V1\_p) \land$
            $V2\_p \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V2\_p) \land$
            $V1s\_p \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V1s\_p) \land$
            $V2s\_p \in RRealPlus \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(V2s\_p) \land$
            $solutionOf(Closed2Closed(t, t'), V1\_p, e1) \land$
            $solutionOf(Closed2Closed(t, t'), V2\_p, e2) \land$
            $V1s\_p = V1s\_ \land$
            $V2s\_p = V2s\_$
        $\mid bind(bind(V1\_, V2\_), bind(V1s\_, V2s\_)) \mapsto bind(bind(V1\_p, V2\_p), bind(V1s\_p, V2s\_p))$
          $\},$
          $\{(V1\_ \mapsto V2\_) \mapsto (V1s\_ \mapsto V2s\_) \mid$
            $Vlow \mapsto plus(V1\_ \mapsto V2s\_) \in lt \land$
            $Vhigh \mapsto plus(V1\_ \mapsto V2s\_) \in gt \land$
            $Vlow \mapsto plus(V1s\_ \mapsto V2\_) \in lt \land$
            $Vhigh \mapsto plus(V1s\_ \mapsto V2\_) \in gt$
          $\}$
        $)$
  END

END
```

B.5 Planar Robot Control

This section details the models for the planar robot case study, developped in Section 7.5.1. The domain-specific theory for robots is first given. The abstract robot (with simpler dynamics) model is presented, followed by the concrete robot (with more accurate dynamics) model.

B.5.1 Planar Robot Theory

Listings B.15: Robot theory

```
THEORY
  IMPORT THEORY Approximation
  OPERATORS
    SecondOrder2DimensionSystemFunction expression (correction_coeff: RReal,
        controlArea4: ℙ((RReal×RReal)×(RReal×RReal)))
      direct definition
        (λ ((x1↦x2)↦(x3↦x4))↦((vx↦vy)↦(wx↦wy)) ·
        ((x1↦x2)↦(x3↦x4)) ∈ (RReal×RReal)×(RReal×RReal) ∧ ((vx↦vy)↦(wx
            ↦wy)) ∈ (RReal×RReal)×(RReal×RReal) ∧ ((vx↦vy)↦(wx↦wy)) ∈
            controlArea4
      | (minus(divide(vx↦Rtwo)↦minus(times(correction_coeff↦minus(x3↦wx)
          )↦x1))
        ↦ minus(divide(vy↦Rtwo)↦minus(times(correction_coeff↦minus(x4↦wy
            ))↦x2)))
        ↦(x1
        ↦ x2)
      )
  FirstOrder2DimensionSystemFunction expression (controlArea2: ℙ(RReal×RReal
      ))
    direct definition
      (λ (x1↦x2)↦(vx↦vy) ·
        (x1↦x2) ∈ RReal×RReal ∧ (vx↦vy) ∈ RReal×RReal ∧ (vx↦vy) ∈
            controlArea2
      | ( vx
        ↦ vy
        )
      )
  SecondOrder2DimensionSystem expression (correction_coeff: RReal,
      controlArea4: ℙ((RReal×RReal)×(RReal×RReal)),t0: RRealPlus,x0: (
      RReal×RReal)×(RReal×RReal))
    direct definition
      caode(
        SecondOrder2DimensionSystemFunction(correction_coeff,controlArea4),
        x0,
        t0
      )
  FirstOrder2DimensionSystem expression (controlArea2: ℙ(RReal×RReal),t0:
      RRealPlus,y0: RReal×RReal)
    direct definition
      caode(
        FirstOrder2DimensionSystemFunction(controlArea2),
        y0,
        t0
      )
  PointwiseSlopedControl expression (DR: ℙ(RReal),vx: RReal,vy: RReal,t0:
      RReal)
    well−definedness t0 ∈ DR
    direct definition
      (λ t · t ∈ DR ∧ t0 ↦ t ∈ leq |
        (vx ↦ vy)
        ↦
        (times(vx ↦ minus(t ↦ t0)) ↦ times(vy ↦ minus(t ↦ t0)))
      )
  PointwiseControl expression (DR: ℙ(RReal),vx: RReal,vy: RReal,t0: RReal)
    well−definedness t0 ∈ DR
    direct definition
      (λ t · t ∈ DR ∧ t0 ↦ t ∈ leq | (vx ↦ vy))
```

```
   FirstOrderSystemObserver expression ()
     direct definition
       (λ x · x ∈ RReal×RReal | x )
   SecondOrderSystemObserver expression ()
     direct definition
       (λ (x1↦x2)↦(x3↦x4) · (x1↦x2↦x3↦x4) ∈ RReal×RReal×RReal×RReal | x3
           ↦x4)
AXIOMATIC DEFINITIONS
 pc_control:
   AXIOMS
     second_order_control:
       ∀ DR,UF,vx,vy,cc,t0,x0 ·
         cc ∈ RReal ∧
         DR ⊆ RReal ∧ UF ⊆ RReal×RReal ∧
         (vx↦vy) ∈ UF ∧
         t0 ∈ RReal ∧ t0 ∈ DR ∧
         x0 ∈ (RReal×RReal)×(RReal×RReal) ⇒
           SolvableWith(
             DR,
             SecondOrder2DimensionSystem(cc,UF×(RReal×RReal),t0,x0),
             PointwiseSlopedControl(DR,vx,vy,t0)
           )
     first_order_control:
       ∀ DR,UF,vx,vy,t0,x0 ·
         DR ⊆ RReal ∧ UF ⊆ RReal×RReal ∧
         (vx↦vy) ∈ UF ∧
         t0 ∈ RReal ∧ t0 ∈ DR ∧
         x0 ∈ RReal×RReal ⇒
           SolvableWith(
             DR,
             FirstOrder2DimensionSystem(UF,t0,x0),
             PointwiseControl(DR,vx,vy,t0)
           )
 pc_sim:
   OPERATORS
     NeighborhoodSimulationCondition predicate (mu: RRealPlus,nu: RRealPlus,cc
         : RReal) :
       well−definedness Rzero ↦ cc ∈ lt
   AXIOMS
     nsc_def:
       ∀ mu,nu,cc · mu ∈ RRealPlus ∧ nu ∈ RRealPlus ∧ cc ∈ RReal ∧ Rzero ↦
           cc ∈ lt ⇒ (
         NeighborhoodSimulationCondition(mu,nu,cc) ⇔
         (times(nu ↦ plus(divide(Rone ↦ Rtwo) ↦ plus(times(Rtwo ↦ cc) ↦
             sqrt(plus(Rone ↦ times(plus(Rtwo ↦ Rtwo) ↦ cc)))))) ↦ mu ∈
             leq)
       )
     first_second_order_simulation_function:
       ∀ UF1,UF2,cc,mu,nu ·
         cc ∈ RReal ∧
```

```
            UF1 ⊆ RReal×RReal ∧ UF2 ⊆ RReal×RReal ∧
            mu ∈ RRealPlus ∧ nu ∈ RRealPlus ∧
            UF1 ⊆ DeltaNeighborhoodSet(mu,Rzero↦Rzero) ∧ UF2 ⊆
                DeltaNeighborhoodSet(nu,Rzero↦Rzero) ∧
            NeighborhoodSimulationCondition(mu,nu,cc)
             ⇒ (
              ∃ V · V ∈ SimulationFunctions(
                (RReal×RReal)×(RReal×RReal), RReal×RReal,
                UF1×(RReal×RReal),        UF2,
                SecondOrder2DimensionSystemFunction(cc,UF1×(RReal×RReal)),
                    SecondOrderSystemObserver,
                FirstOrder2DimensionSystemFunction(UF2),
                    FirstOrderSystemObserver
              )
            )
        first_second_order_simulation_delta:
          ∀ UF1,UF2,cc,mu,nu,V ·
            cc ∈ RReal ∧
            UF1 ⊆ RReal×RReal ∧ UF2 ⊆ RReal×RReal ∧
            mu ∈ RRealPlus ∧ nu ∈ RRealPlus ∧
            NeighborhoodSimulationCondition(mu,nu,cc) ∧
            V ∈ SimulationFunctions(
              (RReal×RReal)×(RReal×RReal), RReal×RReal,
              UF1×(RReal×RReal),        UF2,
              SecondOrder2DimensionSystemFunction(cc,UF1×(RReal×RReal)),
                  SecondOrderSystemObserver,
              FirstOrder2DimensionSystemFunction(UF2),
                  FirstOrderSystemObserver
            ) ⇒
              boundedBy(((RReal×RReal)×(RReal×RReal))×(RReal×RReal),V,Rzero,
                  times(Rtwo↦nu))
  distance:
    OPERATORS
      plannar_distance expression (r1: RReal×RReal,r2: RReal×RReal) : RReal
    AXIOMS
      dist_sym:
        ∀ x,y · x ∈ RReal×RReal ∧ y ∈ RReal×RReal ⇒
          plannar_distance(x,y) = plannar_distance(y,x)
      dist_sep:
        ∀ x,y · x ∈ RReal×RReal ∧ y ∈ RReal×RReal ⇒
          (plannar_distance(x,y) = Rzero ⇔ x = y)
      dist_tri:
        ∀ x,y,z · x ∈ RReal×RReal ∧ y ∈ RReal×RReal ∧ z ∈ RReal×RReal ⇒
          plannar_distance(x,z) ↦ plus(plannar_distance(x,y)↦
              plannar_distance(y,z)) ∈ leq
      open_ball_is_open:
        ∀ x,delta · x ∈ RReal×RReal ∧ delta ∈ RRealPlus ∧ Rzero ↦ delta ∈ lt
            ⇒
          IsOpen({ y | plannar_distance(x,y) ↦ delta ∈ gt })
      dist_choice:
        ∀ P,d · P ∈ RReal×RReal ∧ d ∈ RReal ∧ Rzero ↦ d ∈ lt ⇒ (
            ∃ Q · Q ∈ RReal×RReal ∧ d ↦ plannar_distance(P,Q) ∈ lt
          )
      neighborhood_plannar_distance:
        ∀ a,b,delta ·
          a ∈ RReal×RReal ∧ b ∈ RReal×RReal ∧ delta ∈ RRealPlus ∧ Rzero ↦
              delta ∈ lt ∧
          DeltaNeighborhood(delta,a,b)
            ⇒ plannar_distance(a,b) ↦ delta ∈ lt
  THEOREMS
    first_order_system_solvable:
      ∀ DR,UF,t0,y0,vx,vy ·
        DR ⊆ RReal ∧
        UF ⊆ RReal×RReal ∧
        t0 ∈ DR ∧ y0 ∈ RReal×RReal ∧
        vx↦vy ∈ UF ⇒
          SolvableWith(DR,FirstOrder2DimensionSystem(UF,t0,y0),
              PointwiseControl(DR,vx,vy,t0))
    second_order_system_solvable:
      ∀ DR,UF,t0,x0,vx,vy,cc ·
        DR ⊆ RReal ∧
        UF ⊆ RReal×RReal ∧
        t0 ∈ DR ∧ x0 ∈ (RReal×RReal)×(RReal×RReal) ∧
        vx↦vy ∈ UF ∧
        cc ∈ RReal ⇒
          SolvableWith(DR,SecondOrder2DimensionSystem(cc,UF×(RReal×RReal),t0
              ,x0),PointwiseSlopedControl(DR,vx,vy,t0))
END
```

B.5.2 Abstract Robot

Context

Listings B.16: Abstract robot context

```
CONTEXT
  Robot_0_Ctx
EXTENDS
  GenericCtx
CONSTANTS
  SpeedLimit
  CloseEnough
  px0
  py0
  CriticalDistance
AXIOMS
  axm1: SpeedLimit ∈ RReal
  axm2: Rzero ↦ SpeedLimit ∈ lt
  axm3: px0 ∈ RReal
  axm4: py0 ∈ RReal
```

```
    axm5 :  $CloseEnough \in RReal$
    axm6 :  $Rzero \mapsto CloseEnough \in lt$
    axm7 :  $CriticalDistance \in RReal$
    axm8 :  $Rzero \mapsto CriticalDistance \in lt$
    axm9 :  $CloseEnough \mapsto CriticalDistance \in lt$
    axm10 :  $plannar\_distance(px0 \mapsto py0, Rzero \mapsto Rzero) \mapsto CriticalDistance \in lt$
END
```

Machine

Listings B.17: Abstract robot machine

```
MACHINE
    Robot_0
REFINES
    Generic
SEES
    Robot_0_Ctx
VARIABLES   t , pA , Target , Direction
INVARIANTS
    inv1 :  $pA \in RReal \nrightarrow S$
    inv3 :  $Closed2Closed(Rzero, t) \subseteq dom(pA)$
    inv5 :  $x\_p = pA$
    inv6 :  $Target \in RReal \times RReal$
    inv7 :  $Direction \in RReal \times RReal$
    inv8 :  $DeltaNeighborhood(SpeedLimit, Rzero \mapsto Rzero, Direction)$
    inv9 :  $x\_s = (Target \mapsto Direction)$
    inv10 :  $plannar\_distance(Target, Rzero \mapsto Rzero) \mapsto minus(CriticalDistance \mapsto CloseEnough) \in lt$
    inv11 :  $\forall t\_ \cdot t\_ \in Closed2Closed(Rzero, t) \Rightarrow$
        $plannar\_distance(pA(t\_), Rzero \mapsto Rzero) \mapsto CriticalDistance \in lt$
EVENTS
    INITIALISATION
    WITH
        $x\_s'$ :  $x\_s' = Target' \mapsto Direction'$
        $x\_p'$ :  $x\_p' = pA'$
    THEN
        act1 :  $t := Rzero$
        act2 :  $pA := \{Rzero \mapsto (px0 \mapsto py0)\}$
        act3 :  $Target :\mid Target' \in RReal \times RReal \wedge$
            $plannar\_distance(Target', Rzero \mapsto Rzero) \mapsto minus(CriticalDistance \mapsto CloseEnough) \in lt$
        act4 :  $Direction := Rzero \mapsto Rzero$
    END

    Behave
    REFINES  Behave
    ANY   e , tp
    WHERE
        grd1 :  $e \in DE(S)$
        grd2 :  $Solvable(Closed2Closed(t, tp), e)$
        grd3 :  $plannar\_distance(Target, pA(t)) \mapsto CloseEnough \in gt$
    grd4 :  $tp \in RRealPlus$
    grd5 :  $t \mapsto tp \in lt$
    grd6 :
        $CBAPsolutionOfFIS(t, tp, pA, e,$
            $\{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt$
                $\land plannar\_distance(Rzero \mapsto Rzero, px\_ \mapsto py\_) \mapsto CriticalDistance \in lt\})$
WITH
    $x\_p'$ :  $x\_p' = pA'$
    $Inv$ :  $Inv = \{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt\}$
THEN
    act1 :
        $t, pA :\mid$
            $pA' \in RReal \mathrel{+\!\!\!\rightarrow} S \land t' = tp \land$
            $Closed2Closed(Rzero, t') \subseteq dom(pA') \land$
            $CBAPsolutionOf(t, t', pA, pA', e,$
                $\{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt$
                    $\land plannar\_distance(Rzero \mapsto Rzero, px\_ \mapsto py\_) \mapsto CriticalDistance \in lt\})$
END
```

sense__close__enough

```
REFINES  $Sense$
ANY  $next\_direction$ ,  $next\_target$
WHERE
    grd1 :  $next\_direction \in RReal \times RReal$
    grd2 :  $next\_target \in RReal \times RReal$
    grd3 :  $plannar\_distance(Target, pA(t)) \mapsto CloseEnough \in leq$
    grd4 :  $DeltaNeighborhood(SpeedLimit, Rzero \mapsto Rzero, next\_direction)$
    grd5 :  $plannar\_distance(next\_target, Rzero \mapsto Rzero) \mapsto$
        $minus(CriticalDistance \mapsto CloseEnough) \in lt$
WITH
    $s$ :  $s = \{next\_target \mapsto next\_direction\}$
    $p$ :  $p = \{Target \mapsto Direction\} \times \{t\} \times \{px\_ \mapsto py\_ \mid$
        $plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in leq\}$
    $x\_s'$ :  $x\_s' = next\_target \mapsto next\_direction$
THEN
    act1 :  $Direction := next\_direction$
    act2 :  $Target := next\_target$
END
```

transition__change__direction

```
REFINES  $Transition$
ANY  $new\_direction$
WHERE
    grd1 :  $new\_direction \in RReal \times RReal$
    grd2 :  $DeltaNeighborhood(SpeedLimit, Rzero \mapsto Rzero, new\_direction)$
WITH
    $s$ :  $s = \{Target\} \times \{new\_direction\}$
    $x\_s'$ :  $x\_s' = Target \mapsto new\_direction$
THEN
    act1 :  $Direction := new\_direction$
```

```
END

transition_change_target
REFINES Transition
ANY next_target
WHERE
    grd1 : $next\_target \in RReal \times RReal$
    grd2 : $plannar\_distance(next\_target, Rzero \mapsto Rzero) \mapsto$
        $minus(CriticalDistance \mapsto CloseEnough) \in lt$
WITH
    s : $s = \{next\_target\} \times \{Direction\}$
    x_s' : $x\_s' = next\_target \mapsto Direction$
THEN
    act1 : $Target := next\_target$
END

actuate_movement
REFINES Actuate
ANY tp
WHERE
    grd3 : $plannar\_distance(Target, pA(t)) \mapsto CloseEnough \in gt$
    grd7 : $tp \in RRealPlus$
    grd8 : $t \mapsto tp \in lt$
    grd9 :
        $CBAPsolutionOfFIS(t, tp, pA,$
            $withControl($
                $Closed2Closed(t, tp),$
                $FirstOrder2DimensionSystem(DeltaNeighborhoodSet(SpeedLimit, Rzero \mapsto Rzero), t, pA(t)),$
                $PointwiseControl(Closed2Closed(t, tp), prj1(Direction), prj2(Direction), t)$
            $)$
        $, \{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt$
                $\land plannar\_distance(Rzero \mapsto Rzero, px\_ \mapsto py\_) \mapsto CriticalDistance \in lt\})$
WITH
    x_p' : $x\_p' = pA'$
    Inv : $Inv = \{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt\}$
    s : $s = \{Target\} \times \{Direction\}$
    e :
        $e = withControl($
            $Closed2Closed(t, tp),$
            $FirstOrder2DimensionSystem(DeltaNeighborhoodSet(SpeedLimit, Rzero \mapsto Rzero), t, pA(t)),$
            $PointwiseControl(Closed2Closed(t, tp), prj1(Direction), prj2(Direction), t)$
        $)$
THEN
    act1 :
        $t, pA :|$
            $pA' \in RReal \twoheadrightarrow S \land t' = tp \land$
            $Closed2Closed(Rzero, t') \subseteq dom(pA') \land$
            $CBAPsolutionOf($
                $t, t',$
                $pA, pA',$
                withControl(
                  Closed2Closed(t, t'),
                  FirstOrder2DimensionSystem(DeltaNeighborhoodSet(SpeedLimit, Rzero ↦ Rzero), t, pA(t)),
                  PointwiseControl(Closed2Closed(t, t'), prj1(Direction), prj2(Direction), t)
                ),
                {px_ ↦ py_ | plannar_distance(Target, px_ ↦ py_) ↦ CloseEnough ∈ gt
                        ∧plannar_distance(Rzero ↦ Rzero, px_ ↦ py_) ↦ CriticalDistance ∈ lt}
              )
  END

END
```

B.5.3 Concrete Approximated Robot

Context

Listings B.18: Concrete robot context

```
CONTEXT
   Robot_1_Ctx
EXTENDS
   Robot_0_Ctx
CONSTANTS
   ControllerSpeedLimit
   ControlCoefficient
   S2
   AppDelta
AXIOMS
   axm1 : ControllerSpeedLimit ∈ RReal
   axm2 : Rzero ↦ ControllerSpeedLimit ∈ lt
   axm3 : ControlCoefficient ∈ RRealPlus
   axm4 : NeighborhoodSimulationCondition(ControlCoefficient, SpeedLimit, ControllerSpeedLimit)
   axm5 : S2 = (RReal × RReal) × (RReal × RReal)
   axm6 : ControllerSpeedLimit ↦ SpeedLimit ∈ leq
   axm7 : AppDelta = times(Rtwo ↦ ControllerSpeedLimit)
   axm8 : Rzero ↦ AppDelta ∈ lt
END
```

Machine

Listings B.19: Concrete robot machine

```
MACHINE
   Robot_1
REFINES
   Robot_0
SEES
   Robot_1_Ctx
VARIABLES   t ,  Target ,  DirectionControl ,  vC ,  pC
INVARIANTS
```

```
  inv1 :  $DirectionControl \in RReal \times RReal$
  inv2 :  $DeltaNeighborhood(ControllerSpeedLimit, DirectionControl, Rzero \mapsto Rzero)$
  inv3 :  $pC \in RReal \pfun S$
  inv6 :  $Closed2Closed(Rzero, t) \subseteq dom(pC)$
  inv7 :  $vC \in RReal \pfun S$
  inv8 :  $Closed2Closed(Rzero, t) \subseteq dom(vC)$
  inv11 :  $Direction = DirectionControl$
  inv12 :  $DeltaApproximation(Closed2Closed(Rzero, t), AppDelta, pA, pC)$
  inv13 :  $\forall t\_ \cdot t\_ \in Closed2Closed(Rzero, t) \Rightarrow$
      $plannar\_distance(pC(t\_), Rzero \mapsto Rzero) \mapsto minus(CriticalDistance \mapsto AppDelta) \in lt$
EVENTS
  INITIALISATION
  THEN
    act1 :  $t := Rzero$
    act3 :  $pC := \{Rzero \mapsto (px0 \mapsto py0)\}$
    act4 :  $vC := \{Rzero \mapsto (Rzero \mapsto Rzero)\}$
    act5 :  $Target :\mid Target' \in RReal \times RReal \land$
        $plannar\_distance(Target', Rzero \mapsto Rzero) \mapsto minus(CriticalDistance \mapsto CloseEnough) \in lt$
    act6 :  $DirectionControl := Rzero \mapsto Rzero$
  END

  Behave
  REFINES  Behave
  ANY   $e2$ ,  $tp$
  WHERE
    grd1 :  $e2 \in DE(S2)$
    grd2 :  $Solvable(Closed2Closed(t, tp), e2)$
    grd3 :  $plannar\_distance(Target, pC(t)) \mapsto plus(CloseEnough \mapsto AppDelta) \in gt$
    grd4 :  $tp \in RRealPlus$
    grd5 :  $t \mapsto tp \in lt$
    grd6 :  $CBAPsolutionOfFIS(t, tp, bind(vC, pC), e2,$
        $\{(vx\_ \mapsto vy\_) \mapsto (px\_ \mapsto py\_) \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt\})$
  WITH
    $e$ :
      $e \in DE(S) \land$
      $Solvable(Closed2Closed(t, t'), e) \land ($
        $\forall etaA, etaC \cdot$
          $etaA \in RRealPlus \pfun S \land etaC \in RRealPlus \pfun S2 \land$
          $Closed2Closed(Rzero, t') \subseteq dom(etaA) \land$
          $Closed2Closed(Rzero, t') \subseteq dom(etaC) \land$
          $solutionOf(Closed2Closed(t, t'), etaA, e) \land$
          $solutionOf(Closed2Closed(t, t'), etaC, e2) \Rightarrow$
            $DeltaApproximation(Closed2Closed(t, t'), AppDelta, etaA, fproj2(etaC))$
      $)$
    $pA'$ :
      $pA' \in RReal \pfun S \land Closed2Closed(Rzero, t') \subseteq dom(pA') \land$
      $DeltaApproximation(Closed2Closed(Rzero, t'), AppDelta, pA', pC')$
  THEN
    act1 :
      $t, pC, vC :\mid$
```

```
        $pC' \in RReal \rightarrow\!\!\!\!\!\!+ \; S \land$
        $vC' \in RReal \rightarrow\!\!\!\!\!\!+ \; S \land$
        $t' = tp \land$
        $Closed2Closed(Rzero, t') \subseteq dom(pC') \land$
        $Closed2Closed(Rzero, t') \subseteq dom(vC') \land$
        $CBAPsolutionOf(t, t', bind(vC, pC), bind(vC', pC'), e2, \{(vx\_ \mapsto vy\_) \mapsto (px\_ \mapsto py\_) \mid$
          $plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt\})$
END

sense__close__enough
REFINES  sense__close__enough
ANY  next__direction__ctrl ,  next__target
WHERE
   grd1 :  $next\_direction\_ctrl \in RReal \times RReal$
   grd2 :  $next\_target \in RReal \times RReal$
   grd3 :  $plannar\_distance(Target, pC(t)) \mapsto minus(CloseEnough \mapsto AppDelta) \in leq$
   grd4 :  $DeltaNeighborhood(ControllerSpeedLimit, Rzero \mapsto Rzero, next\_direction\_ctrl)$
   grd5 :  $plannar\_distance(next\_target, Rzero \mapsto Rzero) \mapsto$
      $minus(CriticalDistance \mapsto CloseEnough) \in lt$
WITH
   next__direction :  $next\_direction = next\_direction\_ctrl$
THEN
   act1 :  $DirectionControl := next\_direction\_ctrl$
   act2 :  $Target := next\_target$
END

transition__change__direction
REFINES  transition__change__direction
ANY  new__direction__ctrl
WHERE
   grd1 :  $new\_direction\_ctrl \in RReal \times RReal$
   grd2 :  $DeltaNeighborhood(ControllerSpeedLimit, Rzero \mapsto Rzero, new\_direction\_ctrl)$
WITH
   new__direction :  $new\_direction = new\_direction\_ctrl$
THEN
   act1 :  $DirectionControl := new\_direction\_ctrl$
END

transition__change__target
REFINES  transition__change__target
END

actuate__movement
REFINES  actuate__movement
ANY  tp
WHERE
   grd1 :  $plannar\_distance(Target, pC(t)) \mapsto plus(CloseEnough \mapsto AppDelta) \in gt$
   grd7 :  $tp \in RRealPlus$
   grd8 :  $t \mapsto tp \in lt$
   grd9 :
        $CBAPsolutionOfFIS(t, tp, bind(vC, pC),$
            $withControl($
                $Closed2Closed(t, tp),$
                $SecondOrder2DimensionSystem($
                    $ControlCoefficient,$
                    $DeltaNeighborhoodSet(ControllerSpeedLimit, Rzero \mapsto Rzero) \times (RReal \times RReal),$
                    $t,$
                    $vC(t) \mapsto pC(t)$
                $),$
                $PointwiseSlopedControl($
                    $Closed2Closed(t, tp),$
                    $prj1(DirectionControl), prj2(DirectionControl),$
                    $t$
                $)$
            $)$
        $, \{(vC\_ \mapsto pC\_) \mid plannar\_distance(Target, pC\_) \mapsto plus(CloseEnough \mapsto AppDelta) \in gt\})$
WITH
    $pA' :$
        $pA' \in RReal \rightarrow\!\!\!\!\!+ S \wedge$
        $Closed2Closed(Rzero, t') \subseteq dom(pA') \wedge$
        $CBAPsolutionOf($
            $t, t',$
            $pA, pA',$
            $withControl($
                $Closed2Closed(t, t'),$
                $FirstOrder2DimensionSystem(DeltaNeighborhoodSet(SpeedLimit, Rzero \mapsto Rzero), t, pA(t)),$
                $PointwiseControl(Closed2Closed(t, t'), prj1(Direction), prj2(Direction), t)$
            $),$
            $\{px\_ \mapsto py\_ \mid plannar\_distance(Target, px\_ \mapsto py\_) \mapsto CloseEnough \in gt\}$
        $) \wedge$
        $DeltaApproximation(Closed2Closed(Rzero, t'), AppDelta, pA', pC')$
THEN
    act1 :
        $t, pC, vC :|$
            $pC' \in RReal \rightarrow\!\!\!\!\!+ S \wedge$
            $vC' \in RReal \rightarrow\!\!\!\!\!+ S \wedge$
            $t' = tp \wedge$
            $Closed2Closed(Rzero, t') \subseteq dom(pC') \wedge$
            $Closed2Closed(Rzero, t') \subseteq dom(vC') \wedge$
            $CBAPsolutionOf($
                $t, t',$
                $bind(vC, pC),$
                $bind(vC', pC'),$
                $withControl($
                    $Closed2Closed(t, t'),$
                    $SecondOrder2DimensionSystem($
                        $ControlCoefficient,$
                        $DeltaNeighborhoodSet(ControllerSpeedLimit, Rzero \mapsto Rzero) \times (RReal \times RReal),$
                        $t,$
                        $vC(t) \mapsto pC(t)$
                ),
                PointwiseSlopedControl(
                    Closed2Closed(t, t′),
                    prj1(DirectionControl), prj2(DirectionControl),
                    t
                )
            ),
            {(vC_ ↦ pC_) | plannar_distance(Target, pC_) ↦ plus(CloseEnough ↦ AppDelta) ∈ gt}
        )
  END

END
```

B.6 Inverted Pendulum

This section presents the models for the inverted pendulum case study, detailed in Section 7.5.2. The domain theory of inverted pendulums is first given. The non-linear version of the pendulum is then defined, followed by the linearised version.

B.6.1 Inverted Pendulum Theory

Listings B.20: Inverted pendulum theory

```
THEORY
  IMPORT THEORY Approximation
  OPERATORS
    PendulumRawFun expression (omega0: RReal)
      direct definition
        (λ t_↦(x1_↦x2_)↦u_ ·
          t_ ∈ RRealPlus ∧ x1_ ∈ RReal ∧ x2_ ∈ RReal ∧ u_ ∈ RReal
          | x2_ ↦ (plus(times(u_ ↦ cos(x1_)) ↦ times(times(omega0↦omega0)↦
              sin(x1_))))
        )
    PendulumRaw expression (omega0: RReal, x0: RReal×RReal, t0: RRealPlus)
      direct definition
        code(PendulumRawFun(omega0), x0, t0)
    PendulumLinFun expression (omega0: RReal)
      direct definition
        (λ t_↦(x1_↦x2_)↦u_ ·
          t_ ∈ RRealPlus ∧ x1_ ∈ RReal ∧ x2_ ∈ RReal ∧ u_ ∈ RReal
          | x2_ ↦ (plus(u_ ↦ times(times(omega0↦omega0)↦x1_)))
        )
    PendulumLin expression (omega0: RReal, x0: RReal×RReal, t0: RRealPlus)
      direct definition
        code(PendulumLinFun(omega0), x0, t0)
  AXIOMATIC DEFINITIONS
   pendulum_solvability:
     OPERATORS
       theta_max expression (omega0: RReal) : RReal
    AXIOMS
      theta_max_bounds:
        ∀ omega0 · omega0 ∈ RReal ⇒ Rzero ↦ theta_max(omega0) ∈ leq ∧
            theta_max(omega0) ↦ divide(pi↦Rtwo) ∈ leq
      pendulum_raw_controllability:
        ∀ omega0,theta0,thetap0,t0 ·
          omega0 ∈ RReal ∧
          theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
          thetap0 ∈ RReal ∧
          t0 ∈ RRealPlus
            ⇒ (
          ∃ t1 · t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
            Controllable(Closed2Closed(t0,t1),PendulumRaw(omega0,(theta0↦
                thetap0),t0))
        )
      pendulum_lin_controllability:
        ∀ omega0,theta0,thetap0,t0 ·
          omega0 ∈ RReal ∧
          theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
          thetap0 ∈ RReal ∧
          t0 ∈ RRealPlus
            ⇒ (
          ∃ t1 · t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
            Controllable(Closed2Closed(t0,t1),PendulumLin(omega0,(theta0↦
                thetap0),t0))
        )
  pendulum_approx:
    OPERATORS
      PendulumApproxWD predicate (delta: RReal,omega0: RReal,theta_bound:
          RReal,ctrl_bound: RReal,ctrl_bound_lin: RReal,ctrl_delta: RReal,t0:
           RRealPlus,t1: RRealPlus) :
        well−definedness Rzero↦delta ∈ lt,Rzero↦theta_bound ∈ lt,
            theta_bound ↦ theta_max(omega0) ∈ lt,Rzero↦ctrl_bound ∈ lt,Rzero
            ↦ctrl_bound_lin ∈ lt,Rzero↦ctrl_delta ∈ lt,t0 ↦ t1 ∈ lt
    AXIOMS
      PAWD_Approximation:
        ∀ delta,omega0,theta_bound,ctrl_bound,ctrl_bound_lin,ctrl_delta,t0,t1
            ,x0_raw,x0_lin,u_raw,u_lin ·
          delta ∈ RReal ∧ Rzero ↦ delta ∈ lt ∧
          omega0 ∈ RReal ∧
          theta_bound ∈ RReal ∧ Rzero ↦ theta_bound ∈ lt ∧ theta_bound ↦
              theta_max(omega0) ∈ lt ∧
          ctrl_bound ∈ RReal ∧ Rzero ↦ ctrl_bound ∈ lt ∧
          ctrl_bound_lin ∈ RReal ∧ Rzero ↦ ctrl_bound_lin ∈ lt ∧
          ctrl_delta ∈ RReal ∧ Rzero ↦ ctrl_delta ∈ lt ∧
          t0 ∈ RRealPlus ∧ t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
          PendulumApproxWD(delta,omega0,theta_bound,ctrl_bound,ctrl_bound_lin
              ,ctrl_delta,t0,t1) ∧
          u_raw ∈ RReal ⇸ RReal ∧ Closed2Closed(t0,t1) ⊆ dom(u_raw) ∧ (∀ t_
              · t_ ∈ Closed2Closed(t0,t1) ⇒ abs(u_raw(t_)) ↦ ctrl_bound ∈ lt
              ) ∧
          u_lin ∈ RReal ⇸ RReal ∧ Closed2Closed(t0,t1) ⊆ dom(u_lin) ∧ (∀ t_
              · t_ ∈ Closed2Closed(t0,t1) ⇒ abs(u_lin(t_)) ↦ ctrl_bound_lin
              ∈ lt) ∧
          DeltaApproximation(Closed2Closed(t0,t1),ctrl_delta,u_raw,u_lin) ∧
          x0_raw ∈ RReal×RReal ∧ x0_lin ∈ RReal×RReal ∧ DeltaNeighborhood(
              delta,x0_raw,x0_lin)
            ⇒
              DeltaApproximationEq(Closed2Closed(t0,t1),delta,
                withControl(Closed2Closed(t0,t1),PendulumRaw(omega0,x0_raw,t0
                    ),u_raw),
                withControl(Closed2Closed(t0,t1),PendulumLin(omega0,x0_lin,t0
                    ),u_lin)
              )
  pendulum_control:
    OPERATORS
      PendulumRawControl expression (omega0: RReal,theta0: RReal,thetap0:
          RReal,t0: RRealPlus) : ℙ(RReal×RReal)
        well−definedness abs(theta0) ↦ theta_max(omega0) ∈ lt
      PendulumLinControl expression (omega0: RReal,theta0: RReal,thetap0:
          RReal,t0: RRealPlus) : ℙ(RReal×RReal)
        well−definedness abs(theta0) ↦ theta_max(omega0) ∈ lt
      PC_raw_bound expression () : RReal
      PC_lin_bound expression () : RReal
      PendulumControlDelta expression (omega0: RReal,delta: RReal) : RReal
        well−definedness Rzero ↦ delta ∈ lt
    AXIOMS
      pendulum_raw_control_type:
        ∀ omega0,theta0,thetap0,t0 ·
          omega0 ∈ RReal ∧
          theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
          thetap0 ∈ RReal ∧
          t0 ∈ RRealPlus
            ⇒ (
              PendulumRawControl(omega0,theta0,thetap0,t0) ∈ RReal ⇸ RReal ∧
              Closed2Infinity(t0) ⊆ dom(PendulumRawControl(omega0,theta0,
                  thetap0,t0))
        )
      pendulum_lin_control_type:
        ∀ omega0,theta0,thetap0,t0 ·
          omega0 ∈ RReal ∧
          theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
          thetap0 ∈ RReal ∧
          t0 ∈ RRealPlus
            ⇒ (
              PendulumLinControl(omega0,theta0,thetap0,t0) ∈ RReal ⇸ RReal ∧
              Closed2Infinity(t0) ⊆ dom(PendulumLinControl(omega0,theta0,
                  thetap0,t0))
        )
      pendulum_raw_control_bound_def:
```

```
    Rzero ↦ PC_raw_bound ∈ lt
  pendulum_raw_control_bound:
    ∀ omega0, theta0, thetap0, t0 ·
      omega0 ∈ RReal ∧
      theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
      thetap0 ∈ RReal ∧
      t0 ∈ RRealPlus
        ⇒ (
          ∀ t_ · t_ ∈ RRealPlus ∧ t0 ↦ t_ ∈ leq ⇒
            abs(PendulumRawControl(omega0, theta0, thetap0, t0)(t_)) ↦
                PC_raw_bound ∈ lt
    )
  pendulum_raw_control_acceptable:
    ∀ omega0, theta0, thetap0, t0 ·
      omega0 ∈ RReal ∧
      theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
      thetap0 ∈ RReal ∧
      t0 ∈ RRealPlus
        ⇒ (
        ∃ t1 · t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
          SolvableWith(
            Closed2Closed(t0, t1),
            PendulumRaw(omega0, (theta0↦thetap0), t0),
            PendulumRawControl(omega0, theta0, thetap0, t0)
          )
    )
  pendulum_raw_control_solution_bounded:
    ∀ omega0, theta0, thetap0, t0, t1 ·
      omega0 ∈ RReal ∧
      theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
      thetap0 ∈ RReal ∧
      t0 ∈ RRealPlus ∧ t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
      SolvableWith(Closed2Closed(t0, t1), PendulumRaw(omega0, (theta0↦
          thetap0), t0), PendulumRawControl(omega0, theta0, thetap0, t0))
      ⇒ (
        ∀ theta_, thetap_ ·
          theta_ ∈ RReal ⇸ RReal ∧ Closed2Closed(t0, t1) ⊆ dom(theta_) ∧
          thetap_ ∈ RReal ⇸ RReal ∧ Closed2Closed(t0, t1) ⊆ dom(thetap_)
              ∧
          solutionOf(
            Closed2Closed(t0, t1),
            bind(theta_, thetap_),
            withControl(
              Closed2Closed(t0, t1),
              PendulumRaw(omega0, (theta0↦thetap0), t0),
              PendulumRawControl(omega0, theta0, thetap0, t0)
            )
          )
          ⇒ (
                ∀ t_ · t_ ∈ Closed2Closed(t0,t1) ⇒ abs(theta_(t_)) ↦ theta0
                    ∈ lt
            )
        )
    pendulum_lin_control_bound_def:
        Rzero ↦ PC_lin_bound ∈ lt
    pendulum_lin_control_bound:
        ∀ omega0,theta0,thetap0,t0 ·
            omega0 ∈ RReal ∧
            theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
            thetap0 ∈ RReal ∧
            t0 ∈ RRealPlus
                ⇒ (
                    ∀ t_ · t_ ∈ RRealPlus ∧ t0 ↦ t_ ∈ leq ⇒
                        abs(PendulumLinControl(omega0,theta0,thetap0,t0)(t_)) ↦
                            PC_lin_bound ∈ lt
        )
    pendulum_lin_control_acceptable:
        ∀ omega0,theta0,thetap0,t0 ·
            omega0 ∈ RReal ∧
            theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
            thetap0 ∈ RReal ∧
            t0 ∈ RRealPlus
                ⇒ (
                ∃ t1 · t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
                    SolvableWith(
                        Closed2Closed(t0,t1),
                        PendulumLin(omega0,(theta0↦thetap0),t0),
                        PendulumLinControl(omega0,theta0,thetap0,t0)
                    )
        )
    pendulum_lin_control_solution_bounded:
        ∀ omega0,theta0,thetap0,t0,t1 ·
            omega0 ∈ RReal ∧
            theta0 ∈ RReal ∧ abs(theta0) ↦ theta_max(omega0) ∈ lt ∧
            thetap0 ∈ RReal ∧
            t0 ∈ RRealPlus ∧ t1 ∈ RRealPlus ∧ t0 ↦ t1 ∈ lt ∧
            SolvableWith(Closed2Closed(t0,t1),PendulumLin(omega0,(theta0↦
                thetap0),t0),PendulumLinControl(omega0,theta0,thetap0,t0))
            ⇒ (
                ∀ theta_,thetap_ ·
                    theta_ ∈ RReal ⇸ RReal ∧ Closed2Closed(t0,t1) ⊆ dom(theta_) ∧
                    thetap_ ∈ RReal ⇸ RReal ∧ Closed2Closed(t0,t1) ⊆ dom(thetap_)
                        ∧
                    solutionOf(
                        Closed2Closed(t0,t1),
                        bind(theta_,thetap_),
                        withControl(
                            Closed2Closed(t0,t1),
                            PendulumLin(omega0,(theta0↦thetap0),t0),
                  PendulumLinControl(omega0,theta0 ,thetap0 ,t0)
                )
              )
              ⇒ (
                ∀ t_ · t_ ∈ Closed2Closed(t0,t1) ⇒ abs(theta_(t_)) ↦ theta0
                    ∈ lt
            )
        )
      pendulum_control_delta_def:
        ∀ omega0,delta ·
          omega0 ∈ RReal ∧
          delta ∈ RReal ∧ Rzero ↦ delta ∈ lt
            ⇒
              Rzero ↦ PendulumControlDelta(omega0,delta) ∈ lt
      pendulum_control_approx:
        ∀ delta ,omega0,theta0_raw ,thetap0_raw ,theta0_lin ,thetap0_lin ,t0 ·
          delta ∈ RReal ∧ Rzero ↦ delta ∈ lt ∧
          omega0 ∈ RReal ∧
          theta0_raw ∈ RReal ∧ abs(theta0_raw) ↦ theta_max(omega0) ∈ lt ∧
          thetap0_raw ∈ RReal ∧
          theta0_lin ∈ RReal ∧ abs(theta0_lin) ↦ theta_max(omega0) ∈ lt ∧
          thetap0_lin ∈ RReal ∧
          t0 ∈ RRealPlus ∧
          DeltaNeighborhood(delta ,(theta0_raw↦thetap0_raw) ,(theta0_lin↦
              thetap0_lin))
            ⇒
              DeltaApproximation(
                Closed2Infinity(t0),
                PendulumControlDelta(omega0,delta),
                PendulumRawControl(omega0,theta0_raw,thetap0_raw,t0),
                PendulumLinControl(omega0,theta0_lin ,thetap0_lin ,t0)
              )
END
```

B.6.2 Non-Linear Inverted Pendulum

Context

Listings B.21: Non-linear pendulum context

```
CONTEXT
  PendulumCtx
EXTENDS
  GenericCtx
CONSTANTS
  omega0
  thetamax
  theta0
  control
AXIOMS
```

```
    axm1 : $omega0 \in RReal$
    axm2 : $omega0 \neq Rzero$
    axm3 : $thetamax = theta\_max(omega0)$
    axm4 : $theta0 \in RReal$
    axm5 : $abs(theta0) \mapsto thetamax \in lt$
    axm6 : $partition(STATES, \{control\})$
END
```

Machine

Listings B.22: Non-linear pendulum machine

```
MACHINE
    Pendulum
REFINES
    Generic
SEES
    PendulumCtx
VARIABLES  t , theta , thetap , t_sense , theta_sense , thetap_sense , control_fun
INVARIANTS
    inv1 : $theta \in RReal \pfun RReal$
    inv2 : $thetap \in RReal \pfun RReal$
    inv3 : $Closed2Closed(Rzero, t) \subseteq dom(theta)$
    inv4 : $Closed2Closed(Rzero, t) \subseteq dom(thetap)$
    inv5 : $x\_p = bind(theta, thetap)$
    inv6 : $\forall t\_ \cdot t\_ \in Closed2Closed(Rzero, t) \Rightarrow abs(theta(t\_)) \mapsto thetamax \in lt$
    inv7 : $x\_s = control$
    inv8 : $t\_sense \in RRealPlus$
    inv9 : $theta\_sense \in RReal$
    inv10 : $thetap\_sense \in RReal$
    inv11 : $control\_fun \in RReal \pfun RReal$
    inv12 : $Closed2Infinity(t\_sense) \subseteq dom(control\_fun)$
    inv13 : $abs(theta\_sense) \mapsto thetamax \in leq$
EVENTS
    INITIALISATION
    WITH
        $x\_p'$ : $x\_p' = \{Rzero \mapsto (theta0 \mapsto Rzero)\}$
        $x\_s'$ : $x\_s' = control$
    THEN
        act1 : $t := Rzero$
        act2 : $theta := \{Rzero \mapsto theta0\}$
        act3 : $thetap := \{Rzero \mapsto Rzero\}$
        act4 : $t\_sense := Rzero$
        act5 : $theta\_sense, thetap\_sense := theta0, Rzero$
        act6 : $control\_fun := PendulumRawControl(omega0, theta0, Rzero, Rzero)$
    END

    Behave
    REFINES  Behave
    ANY   e , tp
```

```
WHERE
    grd1 :  $e \in DE(S)$
    grd2 :  $Solvable(Closed2Closed(t, tp), e)$
    grd4 :  $theta(t) \mapsto thetamax \in lt$
    grd5 :  $tp \in RRealPlus$
    grd6 :  $t \mapsto tp \in lt$
    grd7 :
        $CBAPsolutionOfFIS(t, tp, bind(theta, thetap), e,$
            $\{theta\_\_, thetap\_\_ \cdot theta\_\_ \in RReal \wedge thetap\_\_ \in RReal \wedge$
                    $theta\_\_ \mapsto thetamax \in lt$
                $\mid theta\_\_ \mapsto thetap\_\_\})$
WITH
    $Inv$ :
        $Inv = \{theta\_\_, thetap\_\_ \cdot theta\_\_ \in RReal \wedge thetap\_\_ \in RReal \wedge$
                    $theta\_\_ \mapsto thetamax \in lt$
                $\mid theta\_\_ \mapsto thetap\_\_\}$
    $x\_p'$ :  $x\_p' = bind(theta', thetap')$
THEN
    act1 :
        $t, theta, thetap :\mid$
            $t' = tp \wedge$
            $theta' \in RReal \mathrel{+\!\!\!\rightarrow} RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(theta') \wedge$
            $thetap' \in RReal \mathrel{+\!\!\!\rightarrow} RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(thetap') \wedge$
            $CBAPsolutionOf(t, t', bind(theta, thetap), bind(theta', thetap'), e,$
                $\{theta\_\_, thetap\_\_ \cdot theta\_\_ \in RReal \wedge thetap\_\_ \in RReal \wedge$
                    $theta\_\_ \mapsto thetamax \in lt$
                $\mid theta\_\_ \mapsto thetap\_\_\})$
END

sense__angle
REFINES  Sense
WHERE
    grd1 :  $Rzero \mapsto abs(theta(t)) \in lt$
WITH
    $x\_s'$ :  $x\_s' = control$
    $s$ :  $s = \{control\}$
    $p$ :
        $p = \{control\} \times RReal \times \{theta\_\_, thetap\_\_ \cdot theta\_\_ \in RReal \wedge thetap\_\_ \in RReal \wedge$
                              $abs(theta\_\_) \mapsto thetamax \in geq$
                        $\mid theta\_\_ \mapsto thetap\_\_\}$
THEN
    act1 :  $t\_sense, theta\_sense, thetap\_sense := t, theta(t), thetap(t)$
END

transition__calculate__control
REFINES  Transition
WITH
    $x\_s'$ :  $x\_s' = control$
    $s$ :  $s = \{control\}$
THEN
```

```
    act1 :  $control\_fun := PendulumRawControl(omega0, theta\_sense, thetap\_sense, t\_sense)$
  END

  actuate_balance
  REFINES  $Actuate$
  ANY   $tp$
  WHERE
    grd0 :  $tp \in RRealPlus \wedge t \mapsto tp \in lt$
    grd2 :
      $SolvableWith($
        $Closed2Closed(t, tp),$
        $PendulumRaw(omega0, (theta(t) \mapsto thetap(t)), t),$
          $control\_fun$
      $)$
    grd4 :  $theta(t) \mapsto thetamax \in lt$
  WITH
    $e$ :
      $e = withControl($
        $Closed2Closed(t, t'),$
        $PendulumRaw(omega0, (theta(t) \mapsto thetap(t)), t),$
        $control\_fun$
      $)$
    $Inv$ :
      $Inv = \{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
            $abs(theta\_) \mapsto thetamax \in lt$
          $\mid theta\_ \mapsto thetap\_\}$
    $x\_p'$ :  $x\_p' = bind(theta', thetap')$
    $s$ :  $s = \{control\}$
  THEN
    act1 :
      $t, theta, thetap :\mid$
        $t' = tp \wedge$
        $theta' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(theta') \wedge$
        $thetap' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(thetap') \wedge$
        $CBAPsolutionOf($
          $t, t',$
          $bind(theta, thetap),$
          $bind(theta', thetap'),$
          $withControl($
            $Closed2Closed(t, t'),$
            $PendulumRaw(omega0, (theta(t) \mapsto thetap(t)), t),$
            $control\_fun$
          $),$
          $\{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
            $abs(theta\_) \mapsto thetamax \in lt$
          $\mid theta\_ \mapsto thetap\_\}$
        $)$
  END

END
```

B.6.3 Linearised Inverted Pendulum

Context

Listings B.23: Linear pendulum context

```
CONTEXT
   PendulumLinCtx
EXTENDS
   PendulumCtx
CONSTANTS
   delta
   control_delta
   thetabound
AXIOMS
   axm1:  delta ∈ RReal
   axm2:  Rzero ↦ delta ∈ lt
   axm3:  control_delta = PendulumControlDelta(omega0, delta)
   axm4:  thetabound ∈ RReal
   axm5:  Rzero ↦ thetabound ∈ lt
   axm6:  thetabound ↦ thetamax ∈ lt
   axm7:  delta ↦ thetabound ∈ lt
END
```

Machine

Listings B.24: Linear pendulum machine

```
MACHINE
   PendulumLin
REFINES
   Pendulum
SEES
   PendulumLinCtx
VARIABLES   t , theta , thetap , t_sense , control_fun , theta_sense , thetap_sense ,
      control_fun_lin , theta_lin , thetap_lin , theta_lin_sense , thetap_lin_sense
INVARIANTS
   inv1:  theta_lin ∈ RReal ⇸ RReal
   inv2:  thetap_lin ∈ RReal ⇸ RReal
   inv3:  Closed2Closed(Rzero, t) ⊆ dom(theta_lin)
   inv4:  Closed2Closed(Rzero, t) ⊆ dom(thetap_lin)
   inv5:  DeltaApproximation(Closed2Closed(Rzero, t),
      delta, bind(theta, thetap), bind(theta_lin, thetap_lin))
   inv6:
      ∀t_ · t_ ∈ Closed2Closed(Rzero, t) ⇒
         abs(theta(t_)) ↦ thetabound ∈ lt∧
         abs(theta_lin(t_)) ↦ minus(thetabound ↦ delta) ∈ lt
   inv7:  control_fun_lin ∈ RReal ⇸ RReal
   inv8:  DeltaApproximation(Closed2Infinity(t), control_delta, control_fun, control_fun_lin)
   inv9:  theta_lin_sense ∈ RReal
   inv10:  thetap_lin_sense ∈ RReal
```

```
    inv11 :  $abs(theta\_lin\_sense) \mapsto minus(thetabound \mapsto delta) \in leq$
    inv12 :  $DeltaNeighborhood(delta, theta\_sense \mapsto thetap\_sense, theta\_lin\_sense \mapsto thetap\_lin\_sense)$
EVENTS
    INITIALISATION
    THEN
        act1 :  $t := Rzero$
        act2 :  $theta := \{Rzero \mapsto theta0\}$
        act3 :  $thetap := \{Rzero \mapsto Rzero\}$
        act4 :  $t\_sense := Rzero$
        act5 :  $theta\_sense, thetap\_sense := theta0, Rzero$
        act6 :  $control\_fun := PendulumRawControl(omega0, theta0, Rzero, Rzero)$
        act7 :  $control\_fun\_lin := PendulumLinControl(omega0, theta0, Rzero, Rzero)$
        act8 :  $theta\_lin := \{Rzero \mapsto theta0\}$
        act9 :  $thetap\_lin := \{Rzero \mapsto Rzero\}$
        act10 :  $theta\_lin\_sense, thetap\_lin\_sense := theta0, Rzero$
    END

    Behave
    REFINES  Behave
    ANY   $e$ ,  $tp$
    WHERE
        grd1 :  $e \in DE(S)$
        grd2 :  $Solvable(Closed2Closed(t, tp), e)$
        grd4 :  $theta(t) \mapsto thetabound \in lt$
        grd5 :  $tp \in RRealPlus$
        grd6 :  $t \mapsto tp \in lt$
        grd7 :
            $CBAPsolutionOfFIS(t, tp, bind(theta, thetap), e,$
                $\{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
                        $abs(theta\_) \mapsto thetabound \in lt$
                    $\mid theta\_ \mapsto thetap\_\})$
        grd8 :
            $CBAPsolutionOfFIS(t, tp, bind(theta\_lin, thetap\_lin), e,$
                $\{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
                        $abs(theta\_) \mapsto thetabound \in lt$
                    $\mid theta\_ \mapsto thetap\_\})$
    THEN
        act1 :
            $t, theta, thetap, theta\_lin, thetap\_lin :\mid$
                $t' = tp \wedge$
                $theta' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(theta') \wedge$
                $thetap' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(thetap') \wedge$
                $CBAPsolutionOf(t, t', bind(theta, thetap), bind(theta', thetap'), e,$
                    $\{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
                        $abs(theta\_) \mapsto thetabound \in lt$
                    $\mid theta\_ \mapsto thetap\_\}) \wedge$
                $theta\_lin' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(theta\_lin') \wedge$
                $thetap\_lin' \in RReal \nrightarrow RReal \wedge Closed2Closed(Rzero, t') \subseteq dom(thetap\_lin') \wedge$
                $CBAPsolutionOf(t, t', bind(theta\_lin, thetap\_lin), bind(theta\_lin', thetap\_lin'), e,$
                    $\{theta\_, thetap\_ \cdot theta\_ \in RReal \wedge thetap\_ \in RReal \wedge$
                $abs(theta\_) \mapsto thetabound \in lt$
            $\mid theta\_ \mapsto thetap\_\})$
END

sense_angle
REFINES  $sense\_angle$
WHERE
    grd1:  $Rzero \mapsto abs(theta\_lin(t)) \in lt$
THEN
    act1:  $t\_sense, theta\_sense, thetap\_sense := t, theta(t), thetap(t)$
    act2:  $theta\_lin\_sense, thetap\_lin\_sense := theta\_lin(t), thetap\_lin(t)$
END

transition_calculate_control
REFINES  $transition\_calculate\_control$
THEN
    act1:  $control\_fun := PendulumRawControl(omega0, theta\_sense, thetap\_sense, t\_sense)$
    act2:  $control\_fun\_lin :=$
        $PendulumLinControl(omega0, theta\_lin\_sense, thetap\_lin\_sense, t\_sense)$
END

actuate_balance
REFINES  $actuate\_balance$
ANY  $tp$
WHERE
    grd0:  $tp \in RRealPlus \land t \mapsto tp \in lt$
    grd2:
        $SolvableWith($
            $Closed2Closed(t, tp),$
            $PendulumRaw(omega0, (theta(t) \mapsto thetap(t)), t),$
                $control\_fun$
        $)$
    grd3:
        $SolvableWith($
            $Closed2Closed(t, tp),$
            $PendulumLin(omega0, (theta\_lin(t) \mapsto thetap\_lin(t)), t),$
            $control\_fun\_lin$
        $)$
    grd4:  $abs(theta(t)) \mapsto thetabound \in lt$
    grd5:  $abs(theta\_lin(t)) \mapsto minus(thetabound \mapsto delta) \in lt$
THEN
    act1:
        $t, theta, thetap, theta\_lin, thetap\_lin :\mid$
            $t' = tp \land$
            $theta' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(theta') \land$
            $thetap' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(thetap') \land$
            $CBAPsolutionOf($
                $t, t',$
                $bind(theta, thetap),$
                $bind(theta', thetap'),$
        $withControl($
            $Closed2Closed(t, t')$,
            $PendulumRaw(omega0, (theta(t) \mapsto thetap(t)), t)$,
            $control\_fun$
        $)$,
        $\{theta\_, thetap\_ \cdot theta\_ \in RReal \land thetap\_ \in RReal \land$
            $abs(theta\_) \mapsto thetabound \in lt$
        $\mid theta\_ \mapsto thetap\_\}$
    $)\land$
    $theta\_lin' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(theta\_lin')\land$
    $thetap\_lin' \in RReal \nrightarrow RReal \land Closed2Closed(Rzero, t') \subseteq dom(thetap\_lin')\land$
    $CBAPsolutionOf($
        $t, t'$,
        $bind(theta\_lin, thetap\_lin)$,
        $bind(theta\_lin', thetap\_lin')$,
        $withControl($
            $Closed2Closed(t, t')$,
            $PendulumLin(omega0, (theta\_lin(t) \mapsto thetap\_lin(t)), t)$,
            $control\_fun\_lin$
        $)$,
        $\{theta\_, thetap\_ \cdot theta\_ \in RReal \land thetap\_ \in RReal \land$
            $abs(theta\_) \mapsto minus(thetabound \mapsto delta) \in lt$
        $\mid theta\_ \mapsto thetap\_\}$
    $)$
 END

END
```