A Refinement Strategy for Hybrid System Design with Safety Constraints

Zheng Cheng, Dominique Méry

A Refinement Strategy for Hybrid System Design with Safety Constraints *

Zheng Cheng and Dominique Méry
Université de Lorraine, LORIA UMR CNRS 7503
Campus scientifique - BP 239
54506 Vandœuvre-lès-Nancy, France.
email:firstname.secondname@loria.fr

July 9, 2020

Abstract

Whenever continuous dynamics and discrete control interact, hybrid systems arise. As hybrid systems become ubiquitous and more and more complex, analysis and synthesis techniques are in high demand to design safe hybrid systems. This is however challenging due to the nature of hybrid systems and their designs, and the question of how to formulate and reason their safety problems. Previous work has demonstrated how to extend discrete modelling language Event-B with continuous supports to integrate traditional refinement in hybrid system design. In the same spirit, we extend previous work by proposing a strategy that can coherently refine an abstract hybrid system design with safety constraints down to the concrete one with implementable discrete control that can behave safely. Our proposal is validated on the design of a smart heating system, and we share with our experience.

*This work was supported by grant ANR-17-CE25-0005 (The DISCONT Project `http://discont.loria.fr`) from the Agence Nationale de la Recherche (ANR).

Contents

1 Introduction

Whenever continuous dynamics and discrete control interact, hybrid systems arise. This is especially the case in embedding or distributed systems where logic decision-making are combined with physical continuous processes.

As hybrid systems are becoming increasingly complex, engineers usually start with different mathematical models (e.g. differential equations, timed automata) that are abstracted from systems. Then, the collection of analysis and synthesis techniques based on these models forms the research area of hybrid systems theory (e.g. [1, 2, 3, 4, 5]). They play an important role in the multi-disciplinary design of many technological systems that surround us.

In this work, we focus on the problem of how to design discrete control such that with its integration, the hybrid system can behave safely (i.e. without specified undesired outcome). This is intrinsically difficult mainly due to potential complex continuous behaviors of hybrid systems. However, even with simple dynamics, design implementable discrete control for hybrid system requires a significant amount of interleaving domain-specific knowledge [6, 7]. The problem becomes more challenging when we try to specify the safe behaviors of a hybrid system, and reason whether it can be achieved with the designed discrete control.

Our contribution is that we propose a strategy that can coherently refine an abstract hybrid system design with safety constraints down to the concrete one with implementable discrete control that can behave safely. In the process, we break down a set of interleaving domain-specific knowledge for implementing discrete control into *refinement* steps. Each step aims to introduce a specific kind of implementation detail, and is coherently reasoned whether the discrete control design at the current step can still ensure the safe behaviors of the designed hybrid system. Such strategy makes both the design and reasoning more modular.

Our proposed refinement strategy is inspired by the work of [8], which extends the discrete modelling language Event-B with continuous supports (e.g. continuous types such as real numbers) to integrate traditional refinement in hybrid system modelling. Under the same theoretic framework and tool setting, our proposal, however, focuses on how to engineer implementable discrete control, hence yield a different refinement strategy compared to the work of [8]. In the development of this strategy, we also strength continuous supports in Event-B to ease verification complexity. As the work is achieved in the ANR DISCONT project, we benefit from experience of our partners especially the Toulouse group [9]. We have instantiated our refinement strategy to design a small smart heating system that regulates the room temperature between upper and lower bounds, which shows the feasibility of our proposal.

The paper is structured as follows. Section 2 discusses related research on modelling hybrid systems. Section 3 illustrates the Event-B language and its dialects. Section 4 introduces our structural refinement strategy to model and analyze the safety of hybrid systems. In Section 5, we detail how to instantiate our refinement strategy to design a smart heating system, and discuss the result

in Section 6. Section 7 draws conclusions and lines for future work.

2 Modelling Hybrid Systems

In this section, we review 3 categories of methods for hybrid system design.

Classical engineering methods. Classical engineering methods (e.g. control theories[5].) are based on mathematics to propose metrics, equations, theories or tools that facilitate the design of hybrid systems, or ensuring its correctness.

Matlab, based on model-based development, is one way to design complex hybrid systems. In model-based development, domain-knowledge are encapsulated in models, and rely on automated code generation to produce other artifacts. The benefits are saving time and avoiding the introduction of manually coded errors.

Fitzgerald et al. propose a hybrid and collaborative approach to develop hybrid systems [10]. Co-modeling and co-simulation are collaboratively used.

In our opinion, classical engineering methods are based on a collection of collaborative tools, which do not necessarily have formal documents on their semantics[11, 12]. This hinders the possibility to certify the translations among these tools, and poses questions for the soundness of their hybrid systems design pipe-lines.

Formal languages. Formal languages can also be used to aid the design of hybrid systems. The general idea is to formulate the design of hybrid systems as logical statements, and checked them deductively (e.g. theorem proving) or algorithmically (e.g. model checking).

Platzer designs KeYmaera tool for deductive verification of hybrid systems [2]. Differential dynamic logic (dL) is designed as the back-end to support the logical reasoning of KeYmaera, which is a real-valued first-order dynamic logic for hybrid programs. The general idea is to abstract the safety of a hybrid system as a hybrid program in dL. Then, pre-defined deduction rules are applied to reason its validity. Some of them are designed to ease the complexity of the reasoning. For example, differential invariant is an unique deduction rule in dL. The idea is to design the invariant such that when system state changes, the changing rate of invariant and vector field can be compared and reasoned. The feature is particularly useful for hybrid systems that without unique-analytic solutions.

Hybrid Hoare logic (HHL) has been proposed by Liu et al. for a duration calculus based on hybrid communicating sequential processes [13]. In HHL, the safety of a hybrid system is encoded as a Hoare-triple with history expressions. HHL and dL are similar in the high-level concept of deductive reasoning. However, the verification process is very different, because of the way they models the message communication.

These approaches are *posteriori* verification process, where both the models and specifications need to be presented at the validation phase.

Stepwise Refinement. Stepwise refinement initially models the problem at an abstract level, and depending on the domain of developed system, developer can

decide a refinement strategy about how to make specific part of this abstraction more and more concrete.

Su et al. extends the idea of action systems [14], and proposes a development of hybrid systems in the Event-B language with the Rodin Platform [8]. It is essentially based on refining discrete systems into continuous systems. The continuous features are gradually added to the hybrid system construction while maintaining its consistency w.r.t. the concerned safety property. When they represent continuous features such as continuous variables, they reuse the idea of time-dependent functions, in order to represent continuous variables as time series. It allows them to then use a time pointer *now* to access the system state by indexing *now* in the corresponding time series. To model mathematical concepts (e.g. real numbers, continuity) in Event-B, Su et al. uses the Theory plugin [15], which makes such engineering efforts work harmoniously in Event-B (which was designed for developing discrete systems).

Banach et al. design Hybrid Event-B modelling language to address the design of hybrid systems [3]. The language is also based on stepwise refinement and extending the Event-B language. Their extensions have implicit management (proof obligations) of time which aims to ease user efforts.

Dupont et al. propose and develop a general closed-loop pattern for hybrid systems [9]. The pattern captures several general activities in the closed-loop model. A specific hybrid system should be able to instantiate the designed pattern for addressing its safety concerns.

While these approaches clearly demonstrate the feasibility of stepwise refinement in hybrid system designs. We think a general refinement strategy that aims to generate implementable discrete control for hybrid systems is still missing.

3 Event-B Dialect for Hybrid systems Modelling

First, we explain the main points of the Event-B discrete modelling language. Then, we discuss how to extend it to model continuous features of hybrid systems.

3.1 The Event-B Modelling Language

Event-B is a modelling language, originated from the classical B method [16]. By integrating set-theoretical notations, first-order predicate calculus into the language, it offers a general framework for incrementally developing systems by means of refinement: a system is initially modelled at an abstract level, and depending on the domain of developed system, developer can decide how to make specific part of this abstraction more concrete by refining it into more concrete level. As refinement level goes deeper and deeper, ideally, the final refinement is concrete enough to be directly ported to executable languages [17, 18]. In addition, Event-B also allows developer to specify functional behaviors, safety constraints or other critical properties on the system, and then generates proof obligations to be proved at each refinement. In this way, developers can make

sure the refinements is kept consistent to the abstract initial system that they originally want to develop, which is known as correct by construction development. In what follows, we give a brief introduction of the Event-B language, and we refer [16] for more detailed explanation.

Context c_1 **Extends** c_2
Sets S
Constants C
Axioms A_c
Theorems T_c
End

Listing 1: Abstract syntax of Event-B contexts

Machine m_1 **Refines** m_2
Sees c_1
Variables V
Invariants I
Events E
End

Listing 2: Abstract syntax of Event-B machines

Event e
Any P
Where G
Then A
End

Listing 3: Abstract syntax of Event-B events

Specifically, when we model a system in Event-B, it usually consists of 2 parts: contexts and machines. Each context (abstract syntax shown in Listing 1) gives static properties of the system at a particular refinement level, in terms of user-defined types (specified under $Sets$), static objects (specified under $Constants$), presumed properties (specified under $Axioms$), and derived properties (specified under $Theorems$). It can be extended by other contexts for reuse (specified under $Extends$).

Each machine (abstract syntax shown in Listing 2) gives dynamic behaviors (of the system at a particular refinement level, which allows to access contexts that specified under $Sees$). A machine can be refined by another one to make its specific part of dynamic behaviors more concrete (specified under $Refines$), e.g. changing data structure (data refinement) or add complexity (guard strengthening, superposition refinement). Each machine describes the observation of a reactive system modifying a finite list of state variables (specified under $Variables$) satisfying invariant properties (specified under $Invariants$). State variables are only modifiable by means of events (specified under $Events$).

Each event (abstract syntax shown in Listing 3) is parametrized (specified under Any), and defines under which guards (specified under $Where$) the state variables are changed by actions (specified under $Then$). Each action can be non-deterministic or deterministic. Non-deterministic actions take the general form of a *before-after* predicate $v : |BA(C, S, P_e, v, v')$, i.e. state variable is updated such that the post-state v' and its pre-state v have the relation stated by the predicate BA. It is the user's responsibility to ensure the feasibility of each non-deterministic actions. Deterministic actions take the formal of general assignment, which deterministically assigns values to state variables.

From a methodological point of view, classical refinements are generally used to make abstract specification implementable. Therefore, in this setting, actions need to be gradually refined to make them more suitable for implementation (e.g. non-deterministic actions being refined into deterministic ones). However, some state variables can be model variables, i.e. their corresponding updates facilitate proofs, but do not contribute to the final implementation [19].

The Rodin platform is an Eclipse-based IDE for Event-B. It provides ef-

fective support for refinement and mathematical proof. The platform is open source, and can be further extended by various plug-ins. Among existing ones, the Theory plug-in contributes to Rodin by providing facilities to define mathematical extensions. Its functionality are quite similar to contexts. However, unlike contexts that are only visible within the developed Event-B system, Theory plug-in allows users to introduce ones that can be reused across different systems modelled in Event-B. Moreover, it provides mechanism to guide how the prover should use the defined mathematical extensions.

3.2 Event-B Dialect for Hybrid Systems Modelling

Traditionally, Event-B models are specified by discrete variables (whose types are in countable discrete domains, e.g. integers) and discrete events (who only update discrete variables). However, a lot of features in hybrid systems are continuous and hence uncountable, e.g. the history of their physical states. To support system modelling of this kind, Su et al. extend Event-B [8] with: 1) continuous variables, and 2) continuous events. Both of them are developed on top of native Event-B language constructs described in Section 3.1.

Continuous variables. By means of continuous variables, their types are in uncountable domains, e.g. real numbers. For example, the physical states of a hybrid system can be usually defined by a time series $x_c \in \mathbb{R}^+ \to D$, which continuously maps ($\to$) from time domain of positive reals ($\mathbb{R}^+$) to abstract domain of the hybrid system under consideration (D which is generally dense). The key to be able to define continuous variables, is to be able to introduce continuous types, such as $\mathbb{R}^+$. Butler and Abrial initiate the development of a theory for real numbers using the Theory plug-in[1]. We extend this theory in this work with Y additional operations and X axioms to strength continuous supports in Event-B for hybrid systems modelling and verification.

Continuous events. By means of continuous events, they contain actions that update continuous variables. For example, $x_c := x_c \Leftarrow ([0, 1] \lhd f)$ can be an action in a continuous event that update the continuous variable x_c in such way that: the new state of x_c agrees with its old state, except on a closed interval "$[0, 1]$" it agrees with some other continuous variable f.

In this work, all of our continuous events are expressed by actions that update continuous variables exclusively using native Event-B operators on functions and relations. However, we can imaging users might want to encapsulate their methodologies/operators for updating continuous variables using theory plug-in for reuse and readability.

[1] A theory for real numbers. `https://sourceforge.net/projects/rodin-b-sharp/files/Theory_StdLib/`

4 General Pattern for Correct-by-Construction Controler

In this section, we propose a refinement strategy to design discrete control such that with its integration, the hybrid system can behave safely.

In our view, a hybrid system consists of a list of un-dividable sub-systems. Each of these sub-systems is with its own safety constraints. The input of our strategy is thus an un-dividable hybrid system with its own safety constraints. The output is a concrete system with implementable discrete control that can behave safely. Consequently, these sub-systems with concrete designs can be orchestrated to ensure the overall safety of the given hybrid system. However, how to divide a hybrid system into un-dividable sub-ones or orchestra sub-systems are classical modularization problem (discussed in Section 6), which are not our focus in this work.

The intuition of our refinement strategy is that we think the problem of design discrete control for the safe behaviours of a given un-dividable hybrid system is equivalent to the problem of constructing its controlled safe time series. Moreover, we believe that interleaving domain-specific knowledge for implementing discrete control can be broken down into refinement steps, to make the construction of controlled safe time series more and more concrete.

Our refinement strategy can be visualized as in Figure 1. We start by a machine M_specification, which consists of a variable x_a that represents the time series of a generic hybrid system. Such variable is a model variable that only means for proofs. Moreover, since M_specification is generic, we do not know which kind hybrid system is concerned, nor which domain it resides in. We therefore type time series x_a as a partial function that maps from time domain R^+ to an abstract domain D. Similarly, we encode a generic safety property $P_a(d)$, where $d \in D$, to generalize the main safety property that the system to-be-constructed needs to respect.

The next machine M_safety refines M_specification by considering the safety of the input hybrid system. It consists of a model variable x that refines x_a in M_specification by concretizing its type (which now maps time domain to the concrete domain of the input system), and a predicate P that concretely specifies the safety property. All events in this machine yield a "big-step" semantics for the input hybrid system, i.e. there exists some time series f that satisfies P at all times, which can be assigned to x in one go. How to construct such f will be the goal of the following refinements.

Next, the machine M_cycle refines M_safety to make the construction of f more precise by specifying that it needs to be inductively constructed by a piece-wise safe time series. To achieve this, we introduce cycles, where we assume that there is a model variable *now* to help tracking cycles in our proofs. Each cycle is an interval between *now* and $now + \delta$, where δ is a constant to model periodic cycle. At each periodic cycle n, the event in M_cycle aims to specify that there will exist of some function f_n that satisfies P for this cycle. In this way, assuming that we have a time series which up until *now* is safe (i.e. cycles up until $n-1$

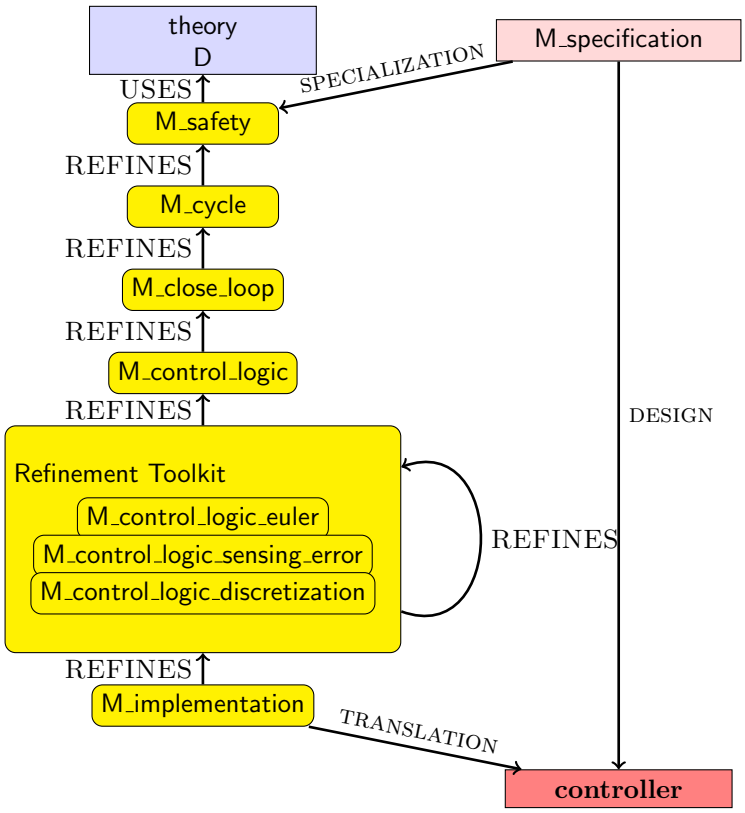


Figure 1: Overview of our proposed structural refinement strategy

is safe), by appending a safe trajectory f_n for an additional cycle n, then it is intuitive that cycles up until n is safe. The cycles recurs towards time infinity to inductively construct the time series f that is promised in the M_safety.

Then, we refine the inductive construction step in M_closed_loop, specifying that each piece in the safe time series to be constructed experiences the full cycle of a simplified closed-loop architecture: at each cycle, the discrete control to be designed, as a blackbox, first predict a safe trajectory f_p (w.r.t. safety property P) over the next δ seconds. Then, the discrete control passes the initiative to the continuous system, where predicted trajectory f_p is assigned to the dynamics of the current cycle f_n to model system progression. Finally, continuous system passes initiative back to discrete control for the next cycle and close the loop. The intuition of this refinement is that, by the closed-loop architecture, we delegate the construction of safe trajectory f_n at each cycle to the discrete control, which is intended to be implemented on a computer and can be refined to be more implementable in the following steps.

In the refinement of M_control_logic, we aim to open the blackbox of discrete control, and turn this blackbox into concrete control logic. The control logic

consists of cases, where each case is designed by 3 consecutive components: 1) sensing. where system state is read. 2) decision, where system state is judged. 3) actuation, where commands (corresponding to the decision) that alternate system state are made and predicted safe trajectory f_p is constructed (w.r.t. the safety property P).

Upon this point, while the discrete control is looming from the closed-loop architecture and the design of control logic cases, it is not yet concrete enough to be implemented. Thus, we develop the discrete control refinement toolkit, which contains a list of domain-specific refinement steps (M_control_logic_euler - M_discretization). These steps stem from our preliminary result that breaks down interleaving domain-specific knowledge for implementing discrete control. Each step modularly introduces a specific kind of implementation detail, and is coherently reasoned by whether the hybrid system still behaves safely. Consequently, the users can choose and apply freely these refinement steps from the toolkit, plus in any order that they desired for engineering implementable discrete control.

In our discrete control refinement toolkit, M_control_logic_euler concretizes the "decision" component in each of the designed control logic cases. The concretization is due to the fact that system state might not only judged by the current state, but also future ones where exact predictions might be involved. Such exact prediction is straightforward and accurate when system dynamics have analytical solutions and initial conditions are known. However, real-life systems rarely meet these criteria, which makes exact state prediction more of an approximation. The problem is that when we use any approximation methods in place of exact mathematical procedures, truncation error occurs. Therefore, if we want to adapt control logic cases by approximations, we need to consider truncation errors carefully.

Truncation errors is highly associated by the chosen approximation method. In this work, we consider one of the simplest approximation methods, i.e. Forward-Euler method: $f_e(n+\delta) = f(n) + \dot{f}(n, f(n)) * \delta$ which simply evaluates the system dynamics ($\dot{f}$) at the current state ($f(n)$), and crudely follows the evaluation to predict future state ($f_e(n+\delta)$) with the given step size (δ). The difference between predicted and exact future state is the introduced truncation error. Feasibility to bound truncation error depends on various factors such as complexity of system dynamics. However, our responsibility in the refinement of M_control_logic_euler is to show that assuming truncation errors are boundable, we can refine control logic cases that are based on judgements of exact predictions to the ones in terms of approximations.

M_control_logic_sensing_error concretizes the "sensing" component in each of the designed control logic cases. When system state is passed from continuous system to discrete control via sensors, error might occur. This error might due to converting from analog signal to digital values (i.e. round-off errors), or defects of the sensors, or noise in the environment. Our responsibility in the refinement of M_control_logic_sensing_error is to show that assuming sensor errors are boundable, then to refine control logic cases by taking into consideration of possible sensor errors.

Towards generating implementation, M_discretization aims to have a discrete view of time series x in this refinement, namely x_d. x_d agrees with x on the system state at precisely the beginning of each cycle, but does not keep track system state between cycles (hence discrete). The benefit is that the discrete nature of x_d makes its persistence on a computer feasible. This offers an extension point for users for more complex designs (e.g. integrals).

Our discrete control refinement toolkit is intended to be extended, and we discuss possible extension points in Section 6.

Once the users are comfortable with implementation details for the designed discrete control, the final refinement M_implementation simply aims to merge designed discrete control cases together to be shipped for implementation.

5 Case Study

In this section, we evaluate our refinement strategy proposed in Section 4 by applying it to design a hybrid smart heating system, which regulates the temperature of a house.

5.1 A Hybrid Smart Heating System with Safety Constraints

The hybrid smart heating system that we consider can operate in two discrete modes: "on" and "off". In each mode, the evolution of the continuous variable, i.e. temperature T, can be described by a differential equation (which we simplified for illustration purpose): when the mode of heating system is "on", the value of temperature follows: $\dot{T} = 1$; when the mode of "off", the value of temperature follows: $\dot{T} = -1$.

Every δ (which is a constant, and greater than zero) seconds, the room temperature T is sampled by a sensor, and sends to a thermostat controller that controls mode switching. If the controller decides to switch mode, there is t_{act} (which is a constant and greater or equal to zero, but strictly smaller than δ) seconds of inertia, e.g. when switching from "on" to "off", the evolution of temperature first follows the differential equation of "on" mode for t_{act} seconds before following the dynamics of the "off" mode.

The safety property that we are interested for this heating system is: the room temperature T must always be greater or equal to T_{min}, and less or equal to T_{max} (where T_{min} strictly less than T_{max}). In what follows, we discuss the development of the heating system by using our proposed refinement strategy.

5.2 Applying Refinement Strategy

5.2.1 M_specification

In the initial hybrid system modeling M_specification (Listing 4), it mainly consists of a variable d that represents the time series of a generic hybrid system,

which maps from time domain R^+ to an abstract domain D (as in $type_d$). In addition, a generic safety property P_a is formulated to generalize the main safety property that any system to-be-constructed needs to respect (as in $safety_d$). Last but not least, an event $Update$ models a "big-step" semantics for any generic hybrid system, i.e. there exists some function f that satisfies safety property at all times, which is assigned to d in one shot.

Machine M_specification
Variables d
Invariants
  $type_d$: d$\in R^+ \to D$
  $safety_d$: $\forall$t$\cdot$t$\in R^+ \Rightarrow P_a$(d(t))
Events
  ...
  Event $Update \mathrel{\widehat{=}}$
    Any f
    Where
      grd_1: f$\in R^+ \to$D
          $\wedge \forall$t$\cdot$t$\in R^+ \Rightarrow P_a$(f(t))
    Then
      act_1 d := f
  End
End

Listing 4: M_specification

Machine M_safety **Refines** M_specification
Variables Ta
Invariants
  $type_{Ta}$: Ta$\in R^+ \to R$
  $safety_{Ta}$: $\forall$t$\cdot$t$\in R^+ \Rightarrow$Ta(t)$\in [T_{min}, T_{max}]$
Events
  ...
  Event $Update \mathrel{\widehat{=}}$
    Refines Update
    Any f
    Where
      grd_1: f$\in R^+ \to$R
          $\wedge \forall$t$\cdot$t$\in R^+ \Rightarrow$f(t)$\in [T_{min}, T_{max}]$
    Then
      act_1 Ta := f
  End
End

Listing 5: M_safety

5.2.2 M_safety

The next machine M_safety (Listing 5) refines M_specification by specifically considering the safety of our hybrid smart heating system. Notably, it specifies that a model variable Ta that abstractly represents the time series of the room (as in $type_{Ta}$). It refines the variable d in M_specification, by concretizing the abstract domain D of M_specification using the concrete domain of real numbers. Moreover, the safety property is also refined w.r.t. our smart heating system, i.e. the room temperature should always be between two constant references T_{min} and T_{max} (as in $safety_{Ta}$). Finally, the $Update$ event is refined such that there exists some function f that satisfies the refined safety property at all times, which is assigned to Ta in one go.

5.2.3 M_cycle

Next, the machine M_cycle (Listing 6) refines M_safety to make the construction of f more precise by specifying that it needs to be inductively constructed by a safe piece-wise time series. To achieve this, we introduce a variable now, initialized at 0, to help tracking of cycles. now partitions Ta into two parts: the past cycles up till now (inclusive), and the future cycles that from now beyond. In addition, we explicitly record the former part as T in this machine, where the

following property bridges between T and Ta: $\forall t \cdot t \in [0, now] \Rightarrow T(t) = Ta(t)$, i.e. the time series Ta and T agree on temperature up till now.

T is safe if the room temperature is bounded within the safe range up until now (as in the invariant $safety_T$). We need to inductively specify that when now progresses, T remains safe. That is why we introduce an inductive event $Prophecy$: at the beginning of each cycle, the $Prophecy$ event assumes the existence of a function f_n that can safely progress until the beginning of the next cycle (as in $safe_{fn}$). Under such assumption, we model the progression of the heating system: 1) time progresses for a sampling period of δ seconds (act_1), and T will follows f_n for the next δ seconds (act_2).

The main proof in this refinement is to establish that the $Prophecy$ event preserves the safety invariant $safety_T$, which can be proved by induction on T and now.

Consequently, in this refinement, now progresses towards time infinity, while T is safely built simultaneously. We therefore can construct a safe time series Ta as promised in M_safety.

5.2.4 M_closed_loop

We then refine the event $Prophecy$ of M_cycle, modeling that the safe function f_n is constructed by experiencing the full cycle of a simplified closed-loop architecture. Specifically, as shown in Listing 7, we first introduce a system mode variable s (as in $type_s$) to distinguish $DECISION$ and RUN modes in the closed-loop architecture. The $DECSION$ mode corresponds to discrete control, and is modelled by the $Prediction$ event. Its semantics is that, it will predict a safe function f_n to progress within the next cycle (as in $safe_{fn}$), and assign the prediction as a candidate for the heating system to progress. How the prediction is done is a blackbox in this refinement, and will be refined in the next refinement. Once prediction finished, the mode is changed to the RUN mode. This mode corresponds to the system progression and is modelled by the $Progression$ event, whose behavior is to follow the predicted candidate for the next cycle. Then, the heating system alternates back to the $DECISION$ mode to predict for the next cycle, thereby forming a closed-loop.

```
Machine M_cycle Refines M_safety
Variables T now
Invariants
  ...
  safety_T:
  ∀t·t∈ [0,now]⇒T(t)∈[T_min,T_max]
Events
  ...
  Event Prophecy ≙
    Refines Update
    Any f_n
    Where
      ...
      safe_fn:
      ∀t·t∈ (now,now+δ]⇒
      f_n(t)∈[T_min,T_max]
    Then
      act_1: now := now+δ
      act_2: T :=
          T ⩤((now,+∞)◁ f_n)
  End
End
```

Listing 6: M_cycle

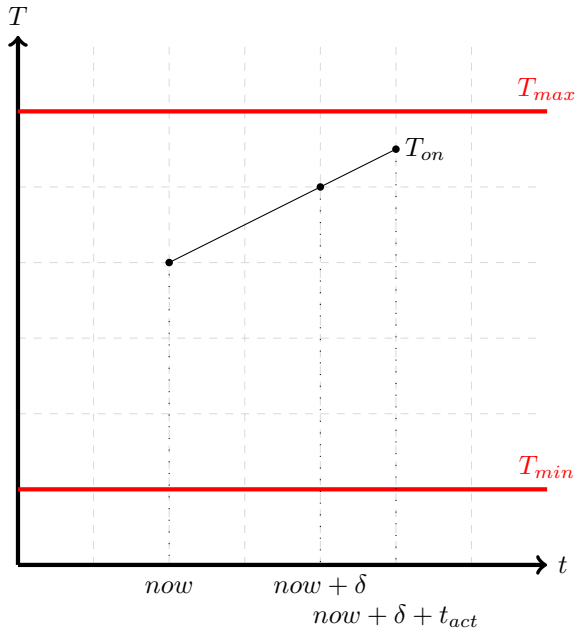
```
Machine M_closed_loop Refines M_cycle
Variables ... s fa
Invariants
  ...
  type_s: s ∈ SysMode
  safe_fa: s=RUN⇒ ∀t·t∈ (now,now+δ]⇒
          fa(t)∈[T_min,T_max]
Events
  ...
  Event Prediction ≙
    Any f_n
    Where
      ...
      safe_fn: ∀t·t∈ (now,now+δ]⇒
      f_n(t)∈[T_min,T_max]
      grd_s: s = DECISION
    Then
      act_1: fa := f_n
      act_s: s := RUN
  End
  Event Progression ≙
    Refine Prophecy
    Where
      ...
      grd_s: s = RUN
    Then
      act_1: now := now+δ
      act_2: T := T ⩤((now,+∞)◁ fa)
      act_s: s := DECISION
  End
End
```

Listing 7: M_closed_loop

5.2.5 M_Control_Logic

In the refinement of M_control_logic, we aim to open the blackbox of discrete control, and turn this blackbox into concrete control logic. By reading carefully the problem description of heating system, we deduce that:

- The two discrete modes inform us the only actuation command is to alternate between these modes.
- The simple dynamic in each mode gives us functions monotonicity.
- The sampling time tells us the cycle duration, and the control logic needs to decide at the beginning of each cycle in order to progress safely for the next full cycle.



- The actuation time warns us there is a cost of switching modes, and our controller design needs to take that into consideration.

Based on these information, we distinguish 8 cases in our control logic. Two of them are demonstrated here for illustration purpose.

Case ONE: ON mode safe

The first case is pictured as in Fig 5.2.5. The intuition of its design is to cope with inertia caused by switching mode: if we are in the ON mode that increases the temperature, and we are at somewhere that not only can safely progress for δ seconds, but also can additionally bear t_{act} seconds of inertia, then we predict that the heating system can stay at the ON mode during the next cycle, and follow the dynamics of the corresponding mode.

To encode this case, we refine the *Prediction* event of M_closed_loop to $Prediction_1$ by concretizing 3 consecutive components of the control logic (Listing 8): 1) sensing, where system mode is read to be "ON" ($sensing_m$) and room temperature is read to follow the dynamics of the ON mode at now ($sensing_T$); 2) decision, where we judge whether the temperature at $now + \delta + t_{act}$ is predicted to be less or equal to T_{max} (as in $decision_1^c$); 3) actuation, where we decide to stay at the ON mode $actuation_m$, and follow the dynamics of T_{on} for the next cycle $actuation_T$.

To ensure the refinement correctness for $Prediction_1$, the main task is to prove the theorem $safe_{fn}$ (i.e. the guard preservation of $safe_{fn}$ for the *Prediction*

event in M_closed_Loop): $\forall t \cdot t \in (now, now + \delta] \Rightarrow T_{on}(t) \in [T_{min}, T_{max}]$. We prove this theorem by proving the following invariant (which trivially implies the theorem $safe_{fn}$): $\forall t \cdot t \in (now, now + \delta + t_{act}] \Rightarrow T_{on}(t) \in [T_{min}, T_{max}]$. This invariant holds for the first case due to the monotonicity of ON mode, and the facts that the boundary value of temperature on the time interval "$[now, now + \delta + t_{act}]$"is safe.

Event $Prediction_1 \widehat{=}$
  Refines Prediction
  Where
    $sensing_m$: m = ON
    $sensing_T$: T(now)=T_{on}(now)
    $decision_1^c$:
    $T_{on}(now + \delta + t_{act}) \leq T_{max}$
    grd_s: s = DECISION
  Theorem
    $safe_{fn}$:$\forall$ t·t$\in$(now,now+δ]$\Rightarrow$
    T_{on}(t) $\in$ [T_{min},T_{max}]
  Then
    $actuation_T$: fa := T_{on}
    $actuation_m$: m := ON
    act_s: s := RUN
End

Listing 8: M_control_logic: ON mode safe case

Case TWO: ON mode unsafe

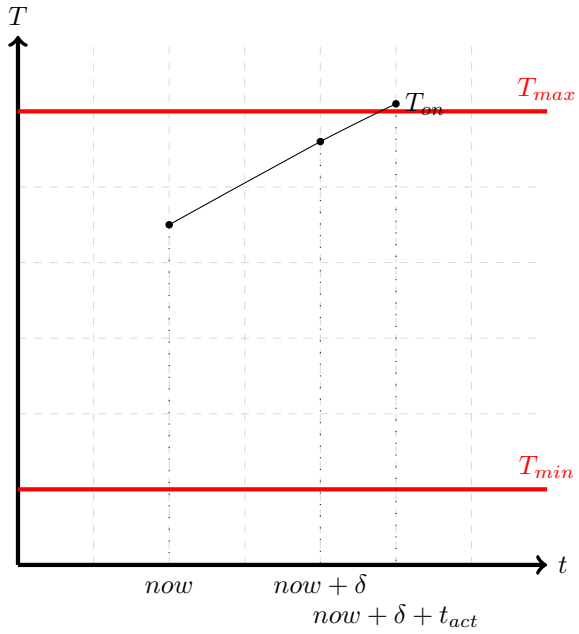
The second case is pictured as in Fig 5.2.5. The scenario captured by this case is that: when the heating system is in the discrete mode ON, and the temperature, follows the dynamics of the ON mode, at $now + \delta + t_{act}$ is greater than T_{max}. Then, we decide to switch to the OFF mode.

The dynamics for the next cycle would be a piece-wise function that follows ON mode for t_{act} seconds, and OFF mode for the rest of cycle. To ensure that following this dynamics is safe for the next cycle, we again prove something stronger:

$$\forall t \cdot t \in (now, now + \delta + t_{act}] \Rightarrow T_{onoff}(t) \in [T_{min}, T_{max}]$$

Proving this property allow us to find a missing requirement in the heating system:

$$T_{min} < T_{max} - 2 * \delta$$



This property simply requires that T_{min} and T_{max} should not be too close. Otherwise, after t_{act} seconds of inertia, it is possible to violate the safety property when continuing the dynamics of the switched mode for the rest of cycle.

In conclusion, we design 8 cases for the control logic of our heating system. The design philosophy is similar to the design of cases 1 and 2. We refer to our on-line repository for the full implementation [20].

5.2.6 M_control_logic_euler

Next, we use M_control_logic_euler from our discrete control refinement toolkit to further refine control cases down to implementation. In this refinement, we introduce the following assumptions (in the context) on truncation errors in each mode:

- (prop^{lte}_{on}) $|T_{on}(now + \delta + t_{act})$ - $Te_{on}(now + \delta + t_{act})| \leq \epsilon^{lte}_{on}$
- $(\text{prop}^{lte}_{off})$ $|T_{off}(now + \delta + t_{act})$ - $Te_{off}(now + \delta + t_{act})| \leq \epsilon^{lte}_{off}$
- $(prop_{\dot{T_{on}}})$ $Min_1 \leq \dot{T_{on}}(now, T_{on}(now)) \leq Max_1$
- $(prop_{\dot{T_{off}}})$ $Min_2 \leq \dot{T_{off}}(now, T_{off}(now)) \leq Max_2$

The first two assumptions simply state that truncation errors in each mode can be bounded by constants. The last two convey that the derivative evaluations in each mode at each cycle can be bounded to constants (since in this example, the derivative in each mode is a constant).

Event $Prediction_1 \widehat{=}$
  Refines $Prediction_1$
  Where
    ...
    $decision_1^e$:
    $T_{on}(now) + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} \leq T_{max}$
  Theorem
    $decision_1^c$: $T_{on}(now + \delta + t_{act}) \leq T_{max}$
  ...
End

Listing 9: M_control_logic_euler

Then, we concretize the "decision" component in each of the designed control logic cases. Listing 9 demonstrates how this is achieved for the first case of the smart heating system. As we can see, the key change in this refinement is the development of new control logic $decision_1^e$. Moreover, to prove the correctness of this refinement, we mainly need to prove the theorem $decision_1^c$, i.e. the guard preservation of $decision_1^c$ for the $Prediction_1$ event w.r.t. M_Control_Logic. The proof is preceded as follows:

$$\begin{aligned}
T_{on}(now + \delta + t_{act}) &\leq Te_{on}(now + \delta + t_{act}) + \epsilon_{on}^{lte} && (\text{prop}_{on}^{lte}) \\
&= T_{on}(now) + \dot{T_{on}}(now, T_{on}(now)) \cdot \delta + t_{act} + \epsilon_{on}^{lte} && (Euler) \\
&\leq T_{on}(now) + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} && (prop_{\dot{T_{on}}}) \\
&\leq T_{max} && (decision_1^e)
\end{aligned}$$

5.2.7 M_control_logic_sensing_error

In this refinement, we introduce the following assumptions (in the context) on sensor error at each cycle: $|T(now) - reading| \leq \epsilon_s$, which simply says the error between system state and sensor reading at each cycle can be bounded by a constant ϵ_s.

Then, we concretize the "sensing" component in each of the designed control logic cases. Listing 10 demonstrates how this is achieved for the first case of the smart heating system. As we can see, we first introduces an argument *reading* to the $Prediction_1$ event to abstractly represent sensor reading. In other word, the system state at each cycle $T(now)$ is masked (hence inaccessible by the control logic). Then, a guard $prop_{sensor}$ conveys our assumption about sensor reading error is bounded. Next, we replace the old control logic $decision_1^e$ by a new control logic as in $decision_1^s$. To prove the correctness of this refinement, we mainly need to prove the guard preservation, i.e. the theorem $decision_1^e$, which is easily discharged by using $prop_{sensor}$.

Event $Prediction_1 \widehat{=}$
  Refines $Prediction_1$
  Any reading **Where**
    ...
    $prop_{sensor}$: $|\mathtt{T(now)}-\mathtt{reading}| \leq \epsilon_s$
    $decision_1^s$:
    $reading + \epsilon_s + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} \leq T_{max}$
  Theorem
    $decision_1^e$:
    $T_{on}(now) + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} \leq T_{max}$
    ...
  End

Listing 10: M_control_logic_sensing_error

Event $Prediction_1 \widehat{=}$
  Refines $Prediction_1$ **Where**
    ...
    $decision_1^d$:
    $T_d(n) + \epsilon_s + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} \leq T_{max}$

    Theorem
    $decision_1^s$:
    $reading + \epsilon_s + Max_1 \cdot \delta + t_{act} + \epsilon_{on}^{lte} \leq T_{max}$
    ...
End

Listing 11: M_discretization

5.2.8 M_discretization

M_discretization refines M_control_logic_sensing_error by introducing a discrete view of time series T, namely T_d. The discrete view T_d and its corresponding view T is glued by the following properties: $\forall c \cdot 0 \leq c \leq n \rightarrow |T(c*\delta) - T_d(c)| \leq \epsilon_s$, where $now = n*\delta$. This property specifies that provided that now is the nth cycle, then the error between system state and sensor reading at each previous cycles are bounded by a constant ϵ_s.

Now, we can use this discrete view in our control logic. For example, the control logic of our first case is shown in Listing 11. It looks like a simple replacement of previous abstract argument *reading* by the abstract view at current cycle $T_d(n)$. However, the insight of this refinement is to make persistent of discrete values feasible (e.g. cycles tracking counter n, or discrete view T_d) for flexible control logic implementation (since for example, storing a continuous view of the system T is impractical).

5.2.9 M_implementation

The last refinement M_implementation will merge all developed control cases together into a single event. Event-B language make this process rigorous and simple. A snippet of the result for this merging is shown in Listing 12.

As we can see in the merged event *Prediction*, each of its guard (whose name with the prefix *case*) is a conjunction of predicates that summarize a particular control case (including sensing, decision and actuation). In order to ensure this refinement is correct, we need to prove that the guards in *Prediction* implies guards of $Prediction_1$ to $Prediction_8$ w.r.t. M_discretization, which is trivially true by rewriting in our case.

The full development of this case study developed in Event-B can be found in [20].

Event $Prediction \mathrel{\widehat{=}}$
  Refines $Prediction_1$ ... $Prediction_8$
  Any a b
  Where
    $case_1$: m=ON $\wedge$ T(now)=T_{on}(now) $\wedge$ $decision_1^d$ $\wedge$ a=T_{on} $\wedge$ b=ON
    ...
    $case_8$: m=OFF $\wedge$ T(now)=T_{off}(now) $\wedge$ $decision_8^d$ $\wedge$ a=T_{off} $\wedge$ b=OFF
  Then
    $actuation_T$: fa := a
    $actuation_m$: m := b
    ...
End

Listing 12: M_implementation

6 Discussions

While our case study shows the feasibility of our approach, we also learn several lessons that we discuss in this section.

Modelling. In this work, we show one way to refine an abstract hybrid system design with safety constraints down to the concrete one with implementable discrete control that can behave safely. However, we do not claim it is the best and only way, and do hope it can be useful for improving or extensions. Here are some possibilities that we can think of:

We assume that time is absolute, a real number t, consistently visible everywhere, and advancing uniformly (classical Newtonian ideal). Only plant can advance time, but cannot change time to go backwards. Our assumption on time has effects on code generation, since an infinite real number has to be represent differently (e.g. float) on a computer. In this case, we think the refinement strategy is similar to what we present in this work, i.e. we need to consider how to justify the possible error introduced when converting real to float. In addition, we find that some mathematical concepts are difficult to encode consistently under this assumption, e.g. limits. Therefore, we plan to investigate the feasibility and usability of other time models (e.g. hyperreals) in our approach.

To simplify our modelling, we assume that sensing and computation take no time. Such assumption do not necessarily hold in reality. For example, control logic in a smart-grid might take some time for a heavy computation to make a actuation decision. However, it is easy to liberate this assumption by considering involved delays in the discrete control design.

We currently do not explicitly model how the system pass its state to the controller, which is usually done by the sensors. To model this behaviour, we can extend our M_Close_Loop with the participation of sensors. We anticipate this could make the following refinement more modular, e.g. the refinement responsibilities in M_Control_Logic_Sensing_Error can be localized.

We currently assume that controller receives the true state of the system

without external helps (e.g. state estimators). If this is not possible, we think that it can be benefit from developing another refinement step in our kit, where the essential point is that true state can be refined by estimated state plus an invariant that captures the relationship between them.

We currently assume that the system behaves normal without disturbance. We think our refinement approach can be directly used to model the system under disturbance, since such case is only a special case of a un-dividable sub-system in normal behavior but different in the system's dynamics. Then, the problem becomes how to coordinate sub-systems of a hybrid system to ensure its overall safety. We think that this is a classical problem of modular verification: we have to show that by design a high level automata, it can orchestra sub-systems to achieve some tasks while ensuring the global safety of the hybrid system. We sketch the development of such high level automata in Appendix A. The key is to be able to derive specifications for sub-systems, and allow them to be used when verifying the high level automata. In our experience, refinement starts with an abstract specification for the designed (sub-)system (in our case M_specification), which make this derivation simpler and thus ideal for such kind of modular verification.

Differential equations. We are currently and indirectly using differential equations in our modelling (e.g. using its analytical solution, approximating by evaluating differential equations). We do not encounter any issues in our modelling or verification so far because of this. However, we do agree that natively expressing them in the Event-B can enhance readability and maintainability. There is already work using the Theory plug-in to develop differential equations support for Event-B models [9].

Soundness of verification. Our reasoning of hybrid system safety is essentially based on axiomatization of hybrid system behaviors, and mathematical theories (e.g. real numbers). We assume that our axiomatization are consistent. To check axioms consistency, we could use the realization mechanism by Why3 [21]: to ensure the theories in the Why3 framework are consistent, the developers clearly separates a small core of axioms, then build lemmas on top of the core. They also use realization mechanism to instantiate uninterpreted functions, then ensuring axioms are provable by instantiation.

Completeness of verification. Verifying the validity of algebraic safety assertions is a fundamental problem in hybrid systems. To ensure the completeness of assertions validity verification, one of the key research is to be able to find appropriate algebraic invariants that quantify the progression of hybrid system states and is useful to prove assertions of interest at the same time. In our current work, we have not considered how to guide user to find these appropriate algebraic invariants. However, we believe that this task to existing frameworks or approaches would be more beneficial, e.g. Pegasus [22], HHL [13].

User experience. The Rodin platform has been designed for Event-B program development. Various plug-ins have been implemented on top of Rodin to make development or proving easier. We have used the theory plug-in to develop our domain theories. We also have used SMT solver plug-in for more automated reasoning. The two plug-ins do not interact natively. Thus, we currently seek

for possibilities that allow SMT solver to draw on domain knowledge for more automated reasoning.

7 Conclusion

The main contribution of this paper is a strategy that can coherently refine an abstract hybrid system design with safety constraints down to the concrete one with implementable discrete control that can behave safely. In the process, we break down a set of interleaving domain-specific knowledge for implementing discrete control into refinement steps. Each step aims to modularly introduce a specific kind of implementation detail, and is coherently reasoned whether the discrete control design at the current step can still ensure the safe behaviors of the designed hybrid system. Our proposed strategy is validated on a case study of smart heating system that regulates the room temperature between upper and lower bounds, which shows its feasibility.

Our future work will focus on developing more case studies that using proposed refinement strategy. We have specifically interest in systems with complex system dynamics that can be linearized. It would be interesting to investigate how the domain knowledge of linearization (e.g. dynamics validity around equilibrium points) can be encoded, and affects our overall strategy. We think that existing works in developing differential equations support for Event-B models can help us to achieve that goal [9]. We are also interested in technology transfer to other programming languages/frameworks for cross-validation. Moreover, to improve user-experience in proofs, we are co-operating with SMT-solvers to generate meaning counter-instances that are useful for domain experts of of hybrid systems.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995.

[2] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[3] Richard Banach, Michael Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Science of Computer Programming*, 105:92 – 123, 2015.

[4] Naijun Zhan, Shuling Wang, and Hengjun Zhao. *Formal Verification of Simulink/Stateflow Diagrams - A Deductive Approach*. Springer, 2017.

[5] Iaon D. Landau and Gianluca Zito. *Digital Control Systems Design Identification and Implementation*. Springer, 2010.

[6] Yamine Aït Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Science of Computer Programming*, 121:100–127, 2016.

[7] Dines Bjørner. Domain analysis and description principles, techniques, and modelling languages. *ACM Transactions on Software Engineering and Methodology*, 28(2):8:1–8:67, 2019.

[8] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing hybrid systems with Event-B and the Rodin platform. *Science of Computer Programming*, 94:164–202, 2014.

[9] Guillaume Dupont, Yamine Aït Ameur, Marc Pantel, and Neeraj Kumar Singh. Handling refinement of continuous behaviors: A proof based approach with Event-B. In *13th International Symposium on Theoretical Aspects of Software Engineering*, pages 9–16, Guilin, China, 2019. IEEE.

[10] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems Co-modelling and Co-simulation*. Springer, 2014.

[11] Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Carl Gamble, Richard Payne, and Kenneth Pierce. Features of integrated model-based co-modelling and co-simulation technology. In *17th International Conference on Software Engineering and Formal Methods*, pages 377–390, Trento, Italy, 2017. Springer.

[12] Mario Gleirscher, Simon Foster, and Jim Woodcock. New opportunities for integrated formal methods. *ACM Computing Surveys*, 52(6):1–36, 2019.

[13] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid CSP. In *8th Asian Symposium on Programming Languages and Systems*, pages 1–15, Shanghai, China, 2010. Springer.

[14] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Continuous action systems as a model for hybrid systems. *Nordic Journal of Computing*, 8(1):2–21, 2001.

[15] Michael Butler and Issam Maamria. Mathematical extension in event-b through the rodin theory component. 2010.

[16] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[17] Neeraj Kumar Singh. Eb2all: an automatic code generation tool. In *Using Event-B for Critical Device Software Systems*, pages 105–141. Springer, 2013.

[18] Zheng Cheng, Dominique Méry, and Rosemary Monahan. On two friends for getting correct programs - automatically translating Event-B specifications to recursive algorithms in rodin. In *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 821–838, Corfu, Greece, 2016. Springer.

[19] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *17th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370, Yogyakarta, Indonesia, 2010. Springer.

[20] Zheng Cheng and Dominique Méry. The full development of smart heating system case study in Event-B. `https://gitlab.inria.fr/mery/discont/-/tree/dev`, 2020.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *22nd European symposium on programming*, pages 125–128, Rome, Italy, 2013. Springer.

[22] Andrew Sogokon, Stefan Mitsch, Yong Kiam Tan, Katherine Cordwell, and André Platzer. Pegasus: A framework for sound continuous invariant generation. In *23rd International Symposium on Formal Methods*, pages 138–157. Springer, 2019.

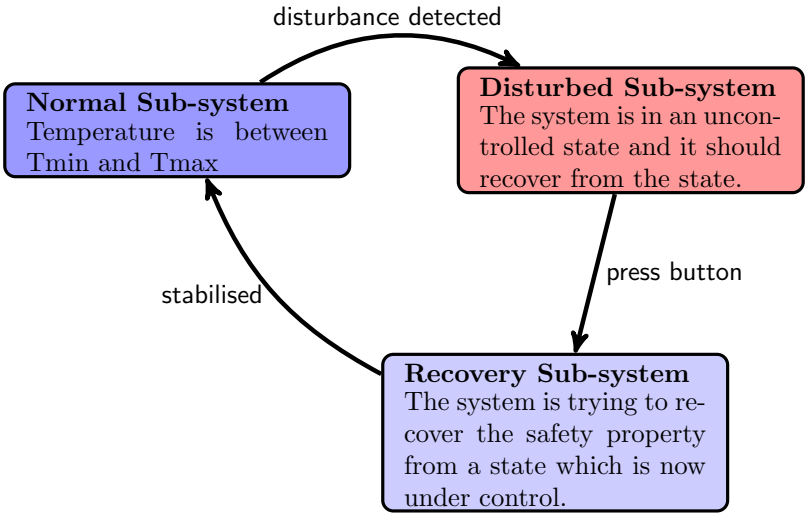


Figure 2: High level automata of the smart heating system

A Modular Verification of Hybrid Systems

In this section, we show that how to use a high level automata, that coordinates different behaviors of the hybrid system under concern, to achieve some tasks while ensuring the global safety.

We illustrate by using a variation of the smart heating system given in Section 5. The variation stems from the fact that now the system might expose to one kind of external disturbance, i.e. momentary white noise. Then, we want to stabilize the system using customized dynamics (e.g. faster recovery).

To model the behaviors of the new heating system, we distinguish 3 of its sub-systems (Fig 2):

- A normal sub-system that maintains the room temperature between given two references T_{min} and T_{max}.
- A disturbed sub-system that detects the abnormal room temperature which outside of given temperature references.
- A recovery sub-system that recovers the room temperature from abnormal back to normal reference temperature.

Intuitively, the heating system is initialized at the normal sub-system, when abnormal cases are detected, the system transits to the disturbed sub-system for alerting. The disturbed sub-system remains unchanged until user press the recovery button. The it transits to the recovery sub-system until the temperature back to normal.

Each sub-systems has its own safety constraints:

- The normal sub-system should always maintains the room temperature between given two references T_{min} and T_{max}.
- The disturbed sub-system should ensure the alert message is issued.
- The recovery sub-system should recovers the room temperature from abnormal back to normal reference temperature within k seconds.

Each of these sub-systems with its corresponding safety constraints can be developed modularly using the refinement strategy we proposed in Section 4. Then, we can verify that by orchestrating the sub-systems as shown in Figure 2, the overall system, when it is not disturbed, can maintain the room temperature between two references to be safe, which require the safety constraints of each sub-system to prove.