From System Events to Software Operations for Refinement-based Modeling of Hybrid Systems *

Zheng Cheng, Dominique Méry

From System Events to Software Operations for Refinement-based Modeling of Hybrid Systems*

Zheng Cheng† Dominique Méry

August 28, 2023

Abstract

Hybrid systems are widely used in a variety of safety-critical applications. Therefore, it is vital to provide a high-level safety guarantee for their implementations. Based on the action system and the refinement methodology, researchers show how to model a safe time-triggered design in the Event-B language. It consists of system events that monitor the system state periodically and make appropriate actuation decisions for the system's safety. However, two crucial types of system events are missing in such design, i.e. sensing and actuation, which hinders the development of robust hybrid systems. In addition, Event-B is specialized in high-level system modeling. Its primitives are not expressive enough to naturally support refining system events into low-level software implementations. In this work, we propose a way to refine (by decomposition) the time-triggered design to introduce the missing system events, without complex high-level system modeling. Based on the decomposition, we identify which system events are implementable. Then, we define a translational approach to the B language. Our translational approach: 1) systematically modularizes a specified Event-B software operation from associated system events. 2) defines a verified translation to obtain a semantically equivalent correspondent in the B language for each modularized Event-B software operation. This translational approach allows us to reuse the primitives of B for refining system events down to implementations, and reuse the predicate transformers defined on the B primitives to reason for the correctness of refinements. It also ensures that the behaviors of the implementations obtained via refinements in B do not divert from the corresponding system events specified in Event-B.

Keywords: hybrid system, system modeling, program development, refinement, Event-B, B method, code generation

*Supported by the ANR DISCONT Project ANR-17-CE25-0005.

†Zheng Cheng contributed to this work when he was a postdoctoral fellow in the DISCONT project. He now holds another position.

Contents

1 Introduction **3**
- 1.1 Context of the Problem and Sketch of Ideas . . . 3
- 1.2 To B(e), or not to B(e), that is the question:. . . 3
- 1.3 From Hybrid Modelling to Correct-by-Construction Controller . . . 4
- 1.4 Summary . . . 5

2 Overview **5**

3 Summary on the B-software language and the Event-B language **8**

4 Running Example **10**

5 Refining Time-triggered Design **13**
- 5.1 $Sense$ Event . . . 15
- 5.2 $Control$ Event . . . 15
- 5.3 $Actuate$ Event . . . 16
- 5.4 $Plant$ Event . . . 17
- 5.5 Methodological Summary . . . 17

6 A Translational Approach for Code Generation of System Events **17**
- 6.1 Modularization of Software Operators . . . 18
  - 6.1.1 Predicate Extraction . . . 18
  - 6.1.2 Encapsulation of Software Operators . . . 18
  - 6.1.3 Merging of System Events . . . 19
- 6.2 Verified Translation to B . . . 19
  - 6.2.1 Syntax and Semantics for the Event-B and B Constructs . . . 19
  - 6.2.2 Translation and Verification . . . 20

7 Discussion **20**

8 Related Work **21**

9 Conclusion **23**

1 Introduction

1.1 Context of the Problem and Sketch of Ideas

Whenever continuous dynamics and discrete control interact, hybrid systems arise. This is especially the case in embedded and distributed systems where decision-making logic is combined with physical continuous processes. Nowadays, hybrid systems are becoming increasingly complex and autonomous in building applications such as self-driving vehicles, and robotics. Therefore, it is vital to provide a high-level safety guarantee for their implementations. Various tools and formalisms are proposed to provide high-level safety guarantees for hybrid systems at the design level [**?**, **?**, 6]. Recently, their extensions are developed for safety guarantees at the implementation level [10, 42], which maximizes state-of-the-art verification technology to meet real-world's needs without error-prone work of implementing design models by hand. Event-B [2] is an evolution of the B language [1], specialized in high-level system design and analysis. Its development is supported by the Rodin toolbox [3], which provides effective features for stepwise refinement and mathematical proofs. It provides a subset of primitives (generalized substitutions) of B for modeling the behaviors of system events and developing reactive systems. However, it is not the focus of Event-B to use these primitives to keep refining the system events into low-level software implementations.

This last remark is important, and is based on the difference between a model corresponding to software and a model corresponding to a system that may include software. A software is a system, but a system is not necessarily a software. These elements serve to warn the reader who might confuse these two notions, expressed in very similar languages such as B [1]'s generalized substitution language with abstract machines. For this paper, an (abstract) system is modeled by a set of *events* that *observe* changes of states described by state variables; a software or a program also describes state transformations defined as *operations* that are *conditionally called*. Moreover, we aim to provide ,a general technique for integrating Event-B models and B models and for illustrating a complete chain for developing correct-by-construction controllers. A controller is a small program which interacts with a plant through sensors and actuators and it will be a C program in our current paper.

1.2 To B(e), or not to B(e), that is the question:...

Confusion may arise in the mind of readers unfamiliar with the dialects using the B notation. The B-Software language [1] is dedicated to *software modeling* and uses the wp calculus to generate verification conditions called *proof obligations;* it is supported by the Atelier-B toolbox [17]. The Event-B/Rodin language (Event-B) [2] or the Event-B/Atelier-B language (B-System) [**?**] are dedicated to modeling systems which may include software; the B-System language is supported by the Atelier-B toolbox and the Event-B language is supported by the Rodin toolbox [3]. A first point is that B-Software and B-System are both supported by the same platform namely Atelier-B; however, the bridge between, the two kinds of modelling is not defined and supported by the current toolbox. Our paper defines, analyzes and checks the translation between the Event-B models and the B-Software models. We have chosen the Rodin platform, which is open and offers a more pedagogical proof assistant. However, we wanted to take advantage of the Atelier-B to C translators to get a complete development chain. Another point should be stated in our so called Event-B models, we are dealing with hybrid systems and we aim to express very complex mathematical properties as Banach spaces or differential equations. Thanks to Back's action systems [6] which are extending the scope of classical action systems and which are extending Event-B models by applying same ideas in a first attempt for

modelling hybrid systems [40]. Technical details will be given later and we hope that this short introduction will provide a clear view of what we did and what we did not. A second point is related to the possibility to extend Event-B models in Rodin using the Theory plugin [**?**, **?**]. Finally, the choice of B notation and these two dialects, Event-B and B-Software, is justified by the tools available. Naturally, the question arises of using another Formal IDE (FIDE) [8, 36] and we can consider the use of Keymaera [**?**], which implements the dL logic with the HP (Hybrid Programs) language equipped with the
leq refinement. There are worksIsabelle/HOL [**?**, **?**] on using dL together with Isabelle/HOL but with no refinement strategy. They need the translation of a validated model into a form amenable to verification in a proof environment such as Isabelle/HOL. HHL [**?**] is another possible candidate for deriving a correct controller, but without refinement. A point which is not addressed when using those tools as well as the B-related tools is the certification of proofs and it is not considered by our approach ,and is left for further works.

1.3 From Hybrid Modelling to Correct-by-Construction Controller

There are many works on designing hybrid systems w.r.t. given safety properties using Event-B [4, 7, 13, 19–23, 31, 39, 40]. However, we do not aware of existing works on extending them for implementing designed Event-B models. Compared to existing works on code generation of Event-B for discrete algorithms [12, 16, 25, 32–34, 41], deriving implementations from hybrid system designs in Event-B requires two new properties: 1) identifying system events that need to be generated. 2) categorizing and modularizing system events to facilitate implementations.

Our proposal is based on [13], where the input is a *time-triggered* design that contains *system* events for the *plant* and the *controller*. The design *periodically* monitors the system state and specifies that, when certain conditions are met, how the system should be actuated to behave safely (Example shown in Section 2).

First, inspired by [22], we propose to refine the time-triggered design to introduce additional system events for sensing and actuation (Section 3). The goal is to model the limitations of the real world in the corresponding system events, and to construct more robust implementations.

Next, we categorize all system events of a time-triggered design into the physical component (plant) and the software-based components (sensor, controller, actuator). The physical component is modeled by the law of physics using for example differential equations. It is given for developing time-triggered designs and should be fixed. We can only *observe* its behaviors and actuate accordingly, but we cannot *program* it to change its behaviors to obtain what we want. The software-based components on the other hand are implementable.

To soundly refine the software-based components down to low-level software implementations, in Section 4, we define a translational approach to the B language [1]. The essential idea is to systematically modularize a specified Event-B software operator from associated system events of each software-based component. Then, by defining a verified translation, we can obtain a semantically equivalent correspondent in the classical B language for each modularized Event-B software operator. This translational approach allows us to reuse the primitives of the B notation for refining system events down to implementations, and reuse the predicate transformers defined on the B primitives to reason about the correctness of refinements. It also ensures that the behaviors of the implementations obtained via refinements in B-Software do not divert from the corresponding system events specified in Event-B.

In addition, the source of our certified translation is the specification of modularized software operation, this greatly reduces the certification effort, and makes our proposal practical.

1.4 Summary

The paper is organized as follows: Section 2 motivates our approach by an example. Section 3 details how we refine the time-triggered design of [13] to introduce the missing system events for sensing and actuation. Then, in Section 4, we define a translational approach to the B language for code generation of system events. We discuss lessons learned in Section 5, and compare our approach to the state-of-the-art in Section 6. Finally, we summarize our work and lines for future work in Section 7.

2 Overview

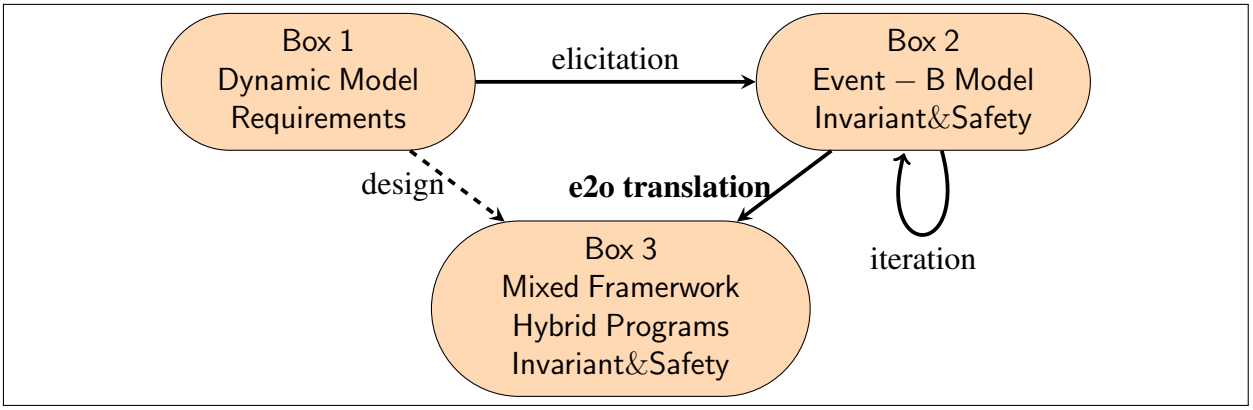


Figure 1: Events to Operations

Development methods based on refinement make it possible to develop software systems that are correct by construction. The approach is progressive and follows a series of *checks* that ensure that a system model is maintained without *errors* or *dust* as emphasized by a fairly old technique developed for software engineering and known as *Cleanroom Model* [**?**, **?**, **?**, **?**] but with a notion of refinement less well taken into account contrary to the case of the B method [1] which paces the activity by checking verification conditions called *proof obligations* implementing an *inductive* theory allowing to derive safety properties or weak invariants: B-software [1] for software design, Event-B [2,3] or B-System [17] for system modelling, constitute instances of the *Cleanroom Model* method, which rely on verification tools, in particular proof assistants. The development process is thus organized as a progressive transformation, controlled by the proof tools, of the model of the system to be built, which is enriched by different aspects or elements indicated by the specifications. The general idea is therefore to progressively add elements to a ¡model in order to obtain an instance as concrete and complete as possible, allowing *in fine* to produce a logical system preserving model properties. This step of passing from a concrete model of the system to the software model amounts to transforming pairs of (condition, action) called *events* in Event-B (or B-System) into (precondition, action) called *operations* in B-Software. Intuitively, an *event* observes a transformation of the system and an *operation* is *called* or executed by a computer. The general process can be enriched by validation of different intermediate models of the refinement chain. It is important to note that the system model integrates both events modeling software transformations and physical transformations such as temperature, force, mass, etc. On the semantic level, an operation is *called* and an event is *observed*. Thus, in an Event-B or B-System or *event-based* model, we identify events that allow us to produce software operations and other events are not translated because they concern

the environment or elements that have a life of their own. We cite a short example on the link between plant and computer.

Example 2.1. *(Plant and Software) An event-based model for a thermostat integrates events modeling the control and decision and events modeling the process that increases or decreases the temperature. It is clear that only the control and decision events are managed by the computer and the other events are only there to express the physical process and are therefore not programmable. We do not reboot a pressure cooker but we reboot the system that controls this pressure cooker. This means that the physical process associated with the temperature involves energy and that this energy is in fact continuously produced and that one can only interact via actuators.*

More generally, this work considers a very specific step of the design process (see Fig: **??**) starting from specification and requirements, then iterating a refinement process until an event-based model concretizes expected requirements with an incremental verification and a validation using Rodin [3] tools when using Event-B, but also tools like simulation [**?**]. The event-based model produced is thus correct by construction, but it is then necessary to extract a controller in our case and identify the events concerned. This last step has to be checked and the translation is relatively systematic. The new derived model is in fact expressed in the form of a B-Software operation which corresponds at the time of its call to the observation of one of the events of the event model. The interest is that this new model is then automatically translated into a programming language such as C, C++ or ADA according to the translators available in the support environment of the B method, Atelier-B [17]. We thus obtain a refinement chain of models whose transformations are verified.

This work highlights the difference between a software operation that is called and an event that is observed. It is based on two very similar environments Rodin for Event-B and Atelier-B for B-Software. It illustrates the feasibility of such an approach from requirements to controller code. It is possible also to use Atelier-B for B-System Atelier-B for B-Software for B-Software but the main advantage of Rodin remains the clever interface of the proof tools making proofs easier to build. We have summarized the main steps in the global process (see Fig: **??**).

Figure **??** shows the different steps of the proposed and sketched methodology. To a certain extent, we extend the Cleanroom Model of software engineering by filling the holes that exist between the different formalisms or languages when considering CPS systems and thus hybrid models. The proof effort is probably the most critical point in these development activities. For each box, we can associate a language or formal model and thus justify the proposed work. Box 1 is the one where requirements are described and a sketch of the hybrid system to be designed is given and this box 1 is for example what is classically described by a set of differential equations and safety properties (temperature is always between two values). MathWork manages these elements and we can also note that hybrid automata [**?**, **?**] undoubtedly constitute a clear and rigorous framework of expressions to model a hybrid system described by linear differential equations and safety properties. This Box 1 is thus occupied by hybrid automata and one can thus use the tools of these models to have a first expression of the problem to be solved as a hybrid automaton satisfying a set of properties. In particular, SpaceEx [**?**] offers possibilities that we have explored. The transition from Box 1 to Box 2 is possible by a translation that we have studied and defined but which will be the subject of a separate document. The interest of this transformation is to allow then a use of the refinement in Box 2 which is in fact the Event-B laboratory where we can play with the abstractions.

Box 2 is thus the laboratory allowing the development of hybrid models written in Event-B and expressed quite directly in the Event-B formalism which is in fact initially defined on

discrete structures but which can be extended by the use of continuous variables [6, 40]. This box is developed in a more operational way by the recent works on the integration of continuous mathematics in the Rodin environment thanks to the Theory plugin [19–23, 39]. The Event-B/Rodin framework provides a way to simulate models under development as it was proposed by [**?**]. The same is true of the work [**?**, 4] which followed the path of an extension of the set of variables by continuous variables and a partial integration of the work of Platzer d$\mathcal{L}$ [**?**]. However, the modeling is particular and takes up the work of [19] by preserving a modeling independent of the time which evolves (dt or time progression event in Event-B model [**?**, 4]) but not at the same time as the state function f of the system (df or plant progression event in Event-B [**?**, 4])) and that leads to an original modeling but far from the standards of physics where the function f and t evolve according to the quantity df/dt. This means that we will use event models [22] with no explicit time progression events in accordance with the physical realist assumptions of the Hybrid Event-B [7] extension. Back [6] had proposed an extension of action systems by interpreting action systems on continuous time-dependent variables and the notation $x : |R(x, x')$ where x is a discrete variable is extended to $y : -e$ or $y := y/now/e$ meaning $y/now/e = \mathsf{if}\ t < now\ \mathsf{then}\ y(t)\ \ \mathsf{else}\ e(t)\ \ \mathsf{fi}$. Finally, Back defines $\dot{y} : -f$ by $\{y : -z | z(now) = y(now) \wedge \dot{z} = f(z), z \geq now\}$. The Event-B generalized substitution can thus be quite simply extended to continuous variables. To complete the box 2, we must point out the importance of the refinement relation defined initially on discrete structures but extensible to continuous variables and to events handling differential equations. In general, this relation is established between an event of an abstract model, possibly *skip*, and a concrete event but this relation can be more specific and a concrete event can refine several events by the merge construction and this possibility will be useful for our work.

The Box 3 is called *Mixed Framework* to indicate that it can contain several techniques of quite different nature. Examples of frameworks are Mathworks [**?**], Keymaera-X [**?**], or Aterlier-B [17],. To some extent, these frameworks are very different but we can use some features offered by Atelier-B for example which is the translator of implementation B machines into C, C++ or ADA code. This functionality is available and we will use it in our work considering it as reliable. Atelier-B is used for the discrete part of the model from Box 2. Of course, classical tools such as Matlab or Simulink allow to understand the hybrid systems under construction. However, the Keymaera-X framework and the associated tools offer more concrete possibilities thanks to the diiferential dynamic logic and to the proof techniques for the design of hybrid programs. To a certain extent this tool is complementary to what we use and we could consider applying Keymaera to validate the program produced by our transformation but this requires a specific study and an adaptation of our transformation.

Our contribution concerns the transformation e2o which transforms a part of the events of a time-triggered event-driven model into a software operation corresponding to the controller. The problem is that event-driven models are all event-driven models and it is clear that the move to a discrete implementation necessarily induces assumptions to be verified and models to be evolved to obtain a model that is time-triggered but also interacts well with the external world of physics, in particular the management of sensor information and the relationship to actuators. To a certain extent, the event-driven models must be sufficiently concrete and time-triggered concrete and time sensitive to be refined into a single operation modeling the controller call.

We wanted to set the scene and situate this work in a larger whole. In the next section, we will specify some elements about the example used to explain this transformation from a suitable concrete model which is a model allowing to merge events to get an operation in a time-triggered view.

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

3 Summary on the B-software language and the Event-B language

Event-B [2] is an evolution of the B language [1], specialized in high-level system modeling and analysis. When we model a reactive system in Event-B, it usually consists of 2 parts: contexts and machines. Each context (abstract syntax shown in Listing **??**) 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 extends other contexts for reuse (specified under **Extends**).

Each machine (abstract syntax shown in Listing **??**) gives dynamic behaviors of the system at a particular refinement level, which allows to access the contexts 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 adding 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 through events (specified under **Events**).

Each event (abstract syntax shown in Listing **??**) can be parametrized (specified under **Any**). It defines under which guards (specified under **Where**) the state variables are changed by actions (specified under **Then**). Each action can be either deterministic or non-deterministic.

- Deterministic actions take the form of general assignment, which deterministically assigns values to state variables.
- Non-deterministic actions take the general form of a before-after predicate $x :| P(x, x')$, i.e. the state variable named x is updated such that the post-state x' and its pre-state x

have the relation stated by the predicate P[1].

It is the user's responsibility to ensure the feasibility of each action.

```
Event e
Any P
Where G
Then A
End
```

Listing 3: Abstract syntax of Event-B events

From a methodological point of view, classical refinements are generally used to make abstract specifications 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.

```
Operator op
Parameters ins
Returns outs
Axioms axioms
End
```

Listing 4: Abstract syntax of Event-B operators developed by the Theory plugin

The Rodin platform is an Eclipse-based IDE for Event-B. It provides effective supports for stepwise refinement and mathematical proofs. The platform is open-source and can be extended by various plugins. Among existing ones, the Theory plugin contributes to Rodin by providing facilities to define mathematical extensions [11]. Its functionality is quite similar to contexts. However, unlike contexts that are only visible within the developed Event-B system, the Theory plugin allows users to introduce ones that can be reused across different systems modeled in Event-B. Moreover, it provides a mechanism to guide how the prover should use the defined mathematical extensions.

A theory developed by the Theory plugin consists of a list of interpreted and uninterpreted operators. The interpreted operators must define their function body. The semantics of uninterpreted operators are axiomatized, which takes the form shown in Listing **??**. Each operator needs to define its inputs and outputs (specified under **Parameters** and **Returns** respectively). It can have a list of axioms to define its semantics (specified under **Axioms**).

We have briefly introduced the Event-B language which is supported by Rodin [3], an open toolset for modelling and reasoning in Event-B, and for completeness, we should also recall that the B-System language is associated with the Atelier-B environment [17]. Event-B and B-System are two dialects for modeling reactive systems and are based on the same assumptions. Historically, Atelier-B was used to develop software and the B Book [1] is the first book presenting the B language. J.-R. Abrial and his collaborators have evolved the language to allow a so-called system approach. Therefore, it is important to remember that there is a separation between the B models and the Event-B models. The passage from an Event-B model to a B model requires the identification of events corresponding to software operations. Thus, the B language provides a way to produce code from an implementation B-machine via a process of

[1]The P predicate of an event can only refer to the constants and sets in the context it allows to **Sees**, and the event parameters, and the variables in its machine.

refinement of operations. Also, in the case of a B model, operations are called and in the case of Event-B, events are observed. The distinction is fundamental and is the focus of this paper. We recall some elements of B.

A B abstract machine AM is defined by variables x that should satisfy an invariant I(x) and by a list of operations which should preserve the invariant when they are called.

The next concept is to state what is the proof obligation of an operation O of the machine AM O is defined as follows:

- $O \stackrel{def}{=}$

  $out \longleftarrow \mathsf{NAMEOP}(in)$
  PRE
  $C(in, x)$
  THEN
  $out, x :\mid (R(in, x, x', out'))$
  END

  where in is an input parameter and out is an output parameter.

- $I(x) \wedge C(in, x) \wedge \mathsf{trm}(R)(in, x) \Rightarrow [O](I(x))$ where $\mathsf{trm}(R)(in, x)$ expresses the termination condition for R.

An operation O is called and the precondition should be true when calling it and it allows to communicate values as in the interaction with sensors and actuators. The key transformation is the merging of events into one event, which is in fact corresponding to an operation. Finally, using events-based modelling, namely Event-B, makes possible the expression of events by operations as long as it corresponds to a software activity.

4 Running Example

As a sample hybrid system, we consider a car position tracking system [37]: while the car is driving down the lane, the controller must choose when to begin decelerating so that it stops at or before a stop sign. The following informal information is given for the modeling.

- the hybrid system has two state variables p for the car position, and v for the car velocity. The direction towards the stop sign is the positive direction.
- the possible behavior of the hybrid system is given by the differential equation $\dot{p} = v, \dot{v} = u$.
- the user could alternate the behaviors of the hybrid system every δ seconds, by actuating the acceleration u in the differential equation.
- the acceleration can be chosen from 3 simple actuation commands: it may cause the car to accelerate with the rate $A > 0$, maintain velocity by choosing acceleration 0, or brake with rate $-B < 0$.
- the safety constraint of the hybrid system is that assuming the stop sign is at the position S, the car position should always satisfy $p \leq S$. Moreover, in reality, when a car brakes, after its velocity reaches 0, it is not possible to drive a car backward by braking. To model this physical limitation of the hybrid system, we have a property that restricts $v \geq 0$ always.

Cheng and Méry [13] develop a refinement strategy to model such hybrid systems in Event-B (we refer to the complete paper [13]). The outcome of their strategy is a time-triggered design that is proven to ensure the given hybrid system behaves safely w.r.t. the given safety property. It can be abstracted as shown in Listing 1.1. It has variables:

- x to represent the system state, which is of type $\mathbb{R}^+ \rightarrow D$ to represent a time series that maps from the time domain (positive real numbers) to the system state domain D. In our example, we thus have $p, v \in \mathbb{R}^+ \nrightarrow \mathbb{R}$ to represent the car position and velocity respectively; both p and v are partial functions.
- now for indexing the time series, e.g. $x(now)$ represents the system state x at time now. System events use now to explicitly model the time progression of the hybrid system according to the relationship $\Delta f = f' * \Delta t$. Domains of p and v are $0..now$.
- s to distinguish between two system modes: 1) discrete control ($DECISION$), and 2) continuous progression (RUN).
- u and t_u to keep track of the chosen actuation command and the safety time envelope under the chosen actuation command. The two variables are shared between the discrete control and the continuous progression modes.

The safety property of the hybrid system is declared as an Event-B invariant, i.e. up until now the system state is safe w.r.t. to the safety property $Safe$.

Then, the behaviors of the hybrid system are modeled by system events, which forms a simplified closed-loop architecture. At the beginning of each cycle, the discrete control will be executed first. It is modeled by a set of $Prediction_i$ system events. Each $Prediction_i$ event predicts an actuation command u_i from all possible commands (grd_3), such that under certain conditions $C_i(x)$, its corresponding trajectory x_{u_i} is verified to progress safely for the next sampling time δ from now (thm_1). For example, the car example yields 3 $Prediction$ events[2]:

Machine M_TIME_TRIGGERED
Variables x now s u t_u
Invariants ...
$safe_x$: $\forall t \cdot t \in [0, now] \Rightarrow Safe(x(t))$
$safe_{run}$: $s = RUN \Rightarrow$
$\Big(\forall t \cdot t \in (now, now + t_u] \Rightarrow Safe(x_u(t))\Big)$
Events ...
Event $Prediction_i \ \widehat{=}$
Where ...
grd_1: $C_i(x)$
grd_2: $s = DECISION$
grd_3: $u_i \in U$
Theorem
thm_1: $\forall t \cdot t \in (now, now + \delta] \Rightarrow Safe(x_{u_i}(t))$
Then ...
act_1: $u, t_u := u_i, \delta$
act_2: $s := RUN$

[2]For each event with the actuation command u_i, p_{u_i}, v_{u_i} are the analytical solutions of the differential equations $\dot{p} = v, \dot{v} = u$, where $p_{u_i}(t) = p(now) + v(now)(t - now) + \frac{1}{2}u_i(t - now)^2$, and $v_{u_i}(t) = v(now) + u_i(t - now)$.

End
Event $Progression \ \widehat{=}$
Any t_r
Where ...
grd_1: $t_r \in [0, t_u]$
grd_2: $s = RUN$
Then
act_1: $x := x \Leftarrow ((now, now + t_r] \lhd x_u)$
act_2: $now := now + t_r$
act_3: $s := DECISION$
End
End

Listing 5: Time-triggered design from [13]

- under $C_i(p, v) \ \widehat{=} \ p_A(now + \delta) + \frac{v_A(now+\delta)^2}{2B} \leq S$, u_i is set to accelerate at the rate of A.
- under $C_i(p, v) \ \widehat{=} \ p_A(now+\delta)+\frac{v_A(now+\delta)^2}{2B} > S \wedge v = 0$, u_i is set to maintain acceleration at the rate of 0.
- under $C_i(p, v) \ \widehat{=} \ p_A(now + \delta) + \frac{v_A(now+\delta)^2}{2B} > S \wedge v \neq 0$, u_i is set to brake at the rate of $-B$.

The predicted actuation command is then stored by u and the safety envelope is stored by t_u (act_1). As thm_1 is proven on all the $Prediction$ events, we can deduce that the $safe_{run}$ is an invariant of the system: i.e. the system can follow the trajectory x_u (resulting from taking the predicted command u) to progress safely for the next t_u seconds.

Next, the $Progression$ event will be observed and it models the continuous progression of the hybrid system. The progression is modeled to take t_r seconds, which is between 0 and t_u (grd_1). Because of the invariant $safe_{run}$, we can ensure that when the system experiences the full predicted cycle ($t_r = t_u$), or exits prematurely ($t_r \in [0, t_u)$), it will behave safely. Thus, the $Progression$ event will then update the system state x to follow the trajectory x_u (resulted from taking the predicted command u) for the next t_r time from now (act_1)[3]. now is simultaneously updated to $now + t_r$ to model the duration of this continuous progression (act_2). Finally, the system alternates back to the discrete control mode to predict the next cycle (act_3). Such time-triggered designs allow the controller to specify system events that monitor the system state periodically and make appropriate actuation decisions for the system to behave safely.

In this work, we aim to extend [13] for implementing designed Event-B models. Firstly, our proposal is inspired by the work of Dupont et al. to introduce additional system events for sensing and actuation [22]. They allow us to model the limitations of the real world, and to construct more robust implementations. A small difference of our proposal compared to [22] is that for a modular refinement strategy, we refine the initial design to introduce the additional events, instead of directly adding them to the initial design. For example, when we refine the time-triggered design of the car example, we introduce a $Sense$ event to model the process that digitizes the true system state $p(now)$ and $v(now)$ into the observed system state ps and vs respectively. Then, it allows us to introduce bounded sensor error between the true state and the observed state, and propagate the error via invariants. Next, the recipients of the sensor data can

[3]Event-B is based on the set theory. $\Leftarrow$ here means relational override, and $\lhd$ means domain restriction.

Operator $car\_ctrl \mathrel{\widehat{=}}$
Parameters ps, vs
Returns u_d, t_u
Axioms
$(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} \leq S \Rightarrow$
$u_d = ACC \wedge t_u = \delta) \wedge$
$(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} > S \wedge v = 0 \Rightarrow$
$u_d = STOP \wedge t_u = \delta) \wedge$
$(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} > S \wedge v \neq 0 \Rightarrow$
$u_d = BRAKE \wedge t_u = \delta)$
End

Listing 6: A specified Event-B software operator for the discrete controller of the car position tracking example

consider the bounded error for a more robust design (i.e. the system remains to behave safely in presence of sensor error).

Secondly, to soundly refine the system events of Event-B down to low-level software implementations, we define a translational approach from Event-B to the B language. We exemplify our approach to the controller development of the car system. We first systematically modularize the $Prediction$ events into a specified Event-B software operator called *car_ctrl* (Listing 1.2).

It encapsulates the input parameters, returned outputs, and the specification of the discrete controller that we want to implement. We then translate the specified Event-B software operator into a B operation (Listing 1.3)[4]. The translated B operation is initially given by a *preconditioned substitution* of B (**Pre**...**Then**...): the type information of translated inputs and outputs is given in the **Pre** section, and the **Then** section uses a before-after predicate of the B language to encode the translated operator specification[5].

We verify that the specification of the Event-B software operator and its translation in B are semantically equivalent. This verification is to ensure that the behaviors of the implementations obtained via refinements in B do not divert from the corresponding system events specified in Event-B. Next, we reuse the programming primitives of B (e.g. *local variable*, *if* and *sequence* substitutions) to refine the translated B operation for producing software implementation (Listing 1.4). The refinement correctness is verified on the Atelier-B platform [27], which implements the existing predicate transformers defined on the B primitives [1].

5 Refining Time-triggered Design

From the time-triggered design developed in [13] (Listing 1.1), our proposal starts by refining it into: 1) physical component (physical plant). 2) software-based components (sensor, controller, actuator). We cannot directly program the differential equations of a physical component to

[4]By an Event-B software operator, we refer to an axiomatic operator introduced by the Theory plugin of Rodin for mathematical extensions of Event-B [11]. By a B(-software) operation, we refer to the standard operation construct in the B(-software) language. A B(-software) operation intends to model a software call and is supported by the B-software language using Atelier-B [27].

[5]In Event-B, x' refers to the post-state of x after the updating. In B, $x\$0$ is the pre-value of x and x is the next value of x. The before-after predicate in B takes the format of $x : (P(x\$0, x))$, which means the variable x is updated in a way that satisfies the predicate P. The notation is changed to $x :| P(x, x')$ in Event-B.

```
u_d, t_u ← car_ctrl(ps, vs) =
Pre
  ps ∈ Real ∧ vs ∈ Real ∧ u_d ∈ U_d ∧ t_u ∈ Real
Then
  u_d, t_u :
    (ps + vsδ + ½Aδ² + (vs+Aδ)²/2B ≤ S ⇒
      u_d = ACC ∧ t_u = δ) ∧
    (ps + vsδ + ½Aδ² + (vs+Aδ)²/2B > S ∧ v = 0 ⇒
      u_d = STOP ∧ t_u = δ) ∧
    (ps + vsδ + ½Aδ² + (vs+Aδ)²/2B > S ∧ v ≠ 0 ⇒
      u_d = BRAKE ∧ t_u = δ)
End
```

Listing 7: The translated B operation of Listing 1.2

```
u_d, t_u ← car_ctrl(ps, vs) =
Var
  v_1, v_2
In
  v_1 := ps + vsδ + ½Aδ² + (vs+Aδ)²/2B ;
  v_2 := S ;
  If v_1 ≤ v_2 Then
    u_d, t_u := ACC, δ
  Else
    If vs = 0 Then
      u_d, t_u := STOP, δ
    Else
      u_d, t_u := BRAKE, δ
    End
  End
End
```

Listing 8: The implemented B operation of Listing 1.3

change its behaviors. However, we can program software-based components to do that.

These components together form an improved closed-loop architecture: 1) at the start, the system state is observed via sensors (the $Sense$ event). 2) Then, the observed data is passed to the controller to make a control decision (the $Control$ events). 3) Next, the control decision is received by the actuator to generate the actuation command (the $Actuate$ event). 4) Finally, the generated actuation command is sent to the plant. The plant reacts to the command, and produces a new system state for observation, thereby closing the loop (the $Plant$ event).

Compared to [13], we introduce two missing system events for sensing and actuation. This allows us to model the limitations of the real world in the corresponding system events, and allow us to construct a more robust design. To establish its refinement relationship with the time-triggered design introduced in [13], we introduce a new mode variable $ss \in \{SENSE, CONTROL, ACTUATE, PLANT\}$, and the following glue invariants with the mode variable $s \in \{DECISION, RUN\}$ in the time-triggered design:

- $ss = SENSE \Rightarrow s = DECISION$
- $ss = CONTROL \Rightarrow s = DECISION$
- $ss = ACTUATE \Rightarrow s = RUN$
- $ss = PLANT \Rightarrow s = RUN$

In what follows, we detail our development for such an improved closed-loop architecture.

5.1 $Sense$ Event

In reality, sensors are used to measure the system state and to convert measurements into digital data to be processed on a computer. However, the measurements could be inaccurate, and the conversion inevitably loses precision. Thus, we introduce the $Sense$ event (Listing 1.5) to model the transformation from the true system state to the digital data, i.e. the state observed by the controller.

The semantics of the $Sense$ event is that it takes the full system state x as the input (after continuous progression), and it outputs the observed state x_s. The types of x and x_s can be different.

Through external engineering efforts (such as reading the specification for the chosen sensors, and/or via observability analysis from the control theory), the user needs to determine a specification R_s that establishes the relationship between x and x_s (act_1).

Then, the specification of the $Sense$ event is propagated via a glue invariant inv_{xs}, and is used for designs of other software-based components.

Example. In the car position tracking example, the $Sense$ event should model the translation from the true system state p and v to the observed state ps and vs. One scenario could be that the state observation does not introduce any noise. Thus, we introduce an invariant to establish their equalities, i.e. $R_s(ps, vs, p, v) \mathrel{\widehat{=}} ps = p \wedge vs = v$. A different scenario could be that the state observation introduces bounded errors ϵ_p and ϵ_v for the state p and v respectively. Then, we can introduce an invariant, i.e. $R_s(ps, vs, p, v) \mathrel{\widehat{=}} |ps - p| \leq \epsilon_p \wedge |vs - v| \leq \epsilon_v$ to propagate this information to other software-based components.

Machine M_IMPL
Refines
M_TIME_TRIGGERED
Variables x x_s ...
Invariants
$inv_{xs} : R_s(x_s, x)$
Events
Event $Sense \ \widehat{=}$
Where
$grd_2 : ss = SENSE$
Then
$act_1 : x_s :| \ R_s(x'_s, x)$
$act_2 : ss := CONTROL$
End
...
End

Listing 9: The $Sense$ event

Event $Control_i \ \widehat{=}$
Refines $Prediction_i$
Where
$grd_1 : CC_i(x_s)$
$grd_2 : ss = CONTROL$
Then
$act_1 : u_d, t_u :|$
$u'_d = u_{d_i} \wedge t'_u = \delta$
$act_2 : ss := ACTUATE$
End

Listing 10: The $Control$ event

Machine M_IMPL
Refines
M_TIME_TRIGGERED
Invariants
$inv_{ud} : R_a(u_d, u)$
Events
Event $Actuate \ \widehat{=}$
Where
$grd_1 : ss = ACTUATE$
Then
$act_1 : u :| \ R_a(u_d, u')$
$act_2 : ss := PLANT$
End
...
End

Listing 11: The $Actuate$ event

5.2 $Control$ Event

Each $Control$ event refines a $Prediction$ event of the time-triggered design. It still models the discrete control, However, its semantics is slightly changed in this refinement (Listing 1.6): the guard of the $Prediction$ event C_i that predicts the true system state is refined into CC_i that predicts the observed state, where CC_i needs to be as strong as C_i (i.e. $CC_i(x_s) \Rightarrow C_i(x)$). Moreover, instead of directly producing the desired actuation command u, we introduce a discrete variable u_d. This allows us to encapsulate actuation-related matters into a separate system event called $Actuate$.

Example. In the car position tracking example, we refine each $Prediction_i$ event to a $Control_i$ event to model that the observed state is received by the controller to generate a discrete actuation command u_d.

The additional complication here is that the 3 sub-cases in the control logic of the time-triggered design are written in terms of the system state p and v, which are not available anymore and need to be rewritten in terms of the observed state ps and vs sent by the $Sense$ event. For example, assuming that there is no sensor error (i.e. using $ps = p \wedge vs = v$ as the invariant via the $Sense$ event), let us consider one of the case $C_i(p, v) \ \widehat{=} \ p_A(now + \delta) + \frac{v_A(now+\delta)^2}{2B} \leq S$, we can expand the definition of p_A and v_A, and replace the reference to p and v with ps and vs to obtain $CC_i(ps, vs) \ \widehat{=} \ ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} \leq S$, which is trivial to prove that it is as strong as $C_i(p, v)$. We can do such replacement analogously when there are bounded sensor errors to be considered. Furthermore, the new control logic will no longer directly produce the actuation command $u \in \{A, 0, -B\}$, but produce a discrete actuation command $u_d \in \{ACC, STOP, BRAKE\}$.

5.3 $Actuate$ Event

The $Actuate$ event is demonstrated by the snippets shown in Listing 1.7. It models how the discrete actuation command u_d is mapped to the desired actuation command u that affects the state of the hybrid system. The goal is to encapsulate actuation-related matters into this system event.

Through external engineering efforts (such as reading the specification for the chosen actuator), the user needs to determine a specification R_a that establishes the relationship between u_d and u. Then, the specification of the $Actuate$ event is propagated via a glue invariant inv_{ud}. Such invariant informs other components on what the $Actuate$ event is produced.

Example. In the car example, the $Actuate$ event models the translation from the discrete actuation command u_d to the desired actuation command u, where the glue invariant $R_a(u_d, u)$:

- $u_d = ACC \Rightarrow u = A$
- $u_d = STOP \Rightarrow u = 0$
- $u_d = BRAKE \Rightarrow u = -B$

5.4 $Plant$ Event

The $Plant$ event corresponds to the $Progression$ event from the time-triggered design, which has the same inputs/outputs behavior as the $Progression$ event: it still receives the desired actuation command u and produces the true system state x. Therefore, there is no adaption required on the $Progression$ event except to change its mode control from s to ss to integrate into the new closed-loop architecture.

5.5 Methodological Summary

The specifications of the $Sense$ and $Actuate$ events are explicitly defined using two predicates $R_s(x_s, x)$ and $R_a(u_d, u)$. Making *explicit* information on the sensing precision or on the actuating effectiveness is related to the domain analysis and formalization [5, 9]. The domain experts may provide a list of properties or assumptions on those predicates. One can consider that those properties are expressed in frameworks as ontologies or knowledge domains.

The $Control$ event emphasizes on concrete control logic development based on the digitized data (i.e. the observed state x_s, and the discrete actuation command u_d). The developed concrete control logic should refine the abstract control logic defined by the $Prediction$ event of the time-triggered design. To establish such refinement generically, a global mathematical theory should be defined based on the *explicit* information from domain experts, which can be developed via the theory mechanism of Event-B (supported by the Theory plugin from Rodin [11]).

6 A Translational Approach for Code Generation of System Events

By the proposed refinement in Section 3, we categorize system events of a *time-triggered* design into the physical component (plant) and the software-based components (sensor, controller, actuator). In this section, we define a translational approach to the B language. Our translational approach: 1) systematically modularize a specified Event-B software operator from associated system events of each software-based component (Section 4.1). 2) defines a verified translation

to obtain a semantically equivalent correspondent in the B language for each modularized Event-B software operator (Section 4.2).

6.1 Modularization of Software Operators

To capture the overall expected semantics of each software-based component, we systematically modularize each software-based component into a specified Event-B software operator in 3 steps.

6.1.1 Predicate Extraction

For each software-based component, we extract a predicate from each of its associated Event-B system events, which takes the form of $G_e(x) \Rightarrow BA_e(x, x')$ (where $G_e(x)$ is the guard of the associated system event e, and $BA_e(x, x')$ is the action of e represented by a before-after predicate) [6].

We then refine the action of each associated event (i.e. $BA_e(x, x')$) by the extracted predicate (i.e. $G_e(x) \Rightarrow BA_e(x, x')$).

For example, by predicate extraction, each $Control$ event of the controller component shown in Listing 1.6 is refined into the snippet shown in Listing 1.8.

Event $Control_i \mathrel{\widehat{=}}$
Refines $Control_i$
Where
$grd_1 : CC_i(x_s)$
$grd_2 : ss = CONTROL$
Then
$act_1 : u_d, t_u :|$
$\Big(CC_i(x_s) \Rightarrow$
$(u'_d = u_{d_i} \wedge t'_u = \delta)\Big)$
$act_2 : ss := ACTUATE$
End

Listing 12: The refined $Control$ event of Listing 1.6 in Event-B

Operator $f_c \mathrel{\widehat{=}}$
Parameters x_s
Returns u_d, t_u
Axioms
$\wedge_i \Big(CC_i(x_s) \Rightarrow$
$(u_d = u_{d_i} \wedge t_u = \delta)\Big)$
End

Listing 13: The template for encapsulating a spec-ified Event-B software operator for the controller component

Event $Control \mathrel{\widehat{=}}$
Refines $Control_1$,
$\ldots$, $Control_i$
Where
$grd_1 : CC_1(x_s) \vee$
$\ldots \vee CC_i(x_s)$
$grd_2 : ss = CONTROL$
Then
$act_1 : u_d$, $t_u := f_c(x_s)$
$act_2 : ss := ACTUATE$
End

Listing 14: The merged event for the controller component in Event-B

6.1.2 Encapsulation of Software Operators

For each software-based component, we then encapsulate a specified Event-B software operator via the theory mechanism of Event-B (using the Theory plugin from Rodin [11]). This encapsulation requires to provide: 1) the inputs of the operator, which are gathered from variables/constants in the guards of associated events of a component. 2) the outputs of the operator, which are gathered from variables in the actions of associated events of a component. 3) the

[6] mode variable ss is excluded since it does not contribute to the implementation logic.

specifications of the operator, which are the logical conjunction of the extracted predicates in the actions of associated events of a component.

The template for encapsulating a specified Event-B software operator for the controller component is demonstrated in Listing 1.9. We thus can instantiate the template to obtain Listing 1.2 for the controller component of the car position tracking example. The other two components are developed analogously.

6.1.3 Merging of System Events

We then refine the action of each associated system event by referring to the encapsulated operator. This is to engineer all associated events of a software-based component to: 1) take the same number/type of inputs, and 2) produce the same number/type of outputs, and 3) perform the same form of behaviors.

Then, we merge all associated events of each software-based component into a single system event of Event-B. This is done by the event merging mechanism of Event-B, where: 1) the actions of the events to be merged must be identical, and become the actions of the merged event. 2) the guards of the events to be merged form a logical $disjunction$, which becomes the guard of the merged event. We demonstrate the merging for the controller component in Listing 1.10.

6.2 Verified Translation to B

Next, we translate each specified Event-B software operator into a B operation. The translated B operation is initially given by a *preconditioned substitution* of B (**Pre**...**Then**...), where the type information of translated inputs and outputs is given in the **Pre** section, and the translated operator specification is given in the **Then** section (See Listing 1.3 for an example).

If we incorrectly translate the specifications of Event-B software operators, our development in B would start with the wrong specifications, and thus not able to obtain the implementation that we want. Thus, in this section, we verify that the specification of the Event-B software operator and its translation in B are semantically equivalent.

6.2.1 Syntax and Semantics for the Event-B and B Constructs

First, we formalize the syntax and semantics for a subset of the Event-B and B languages in the Dafny language [28]. The subsets consist of constructs for expressions and predicates, which we use to express the operator specifications in both languages. We choose Dafny because of its proof automation, but other languages with equivalent expressiveness can be used for the same task (e.g. Coq [8], Lean [35], Isabelle [36]).

We first inductively define the syntax of the Event-B and B expressions. For example, the following Dafny code defines the integer expression for Event-B, which can be constructed from either a literal expression, an identifier, an unary expression (negation), or a binary expression (addition, subtraction, multiplication, and division):

```
datatype ebIntExpr =
  | ebIntLitExpr(n: int)
  | ebIntIdExpr(v: string)
  | ebIntUnExpr(op: ebIntUnOp, e: ebIntExpr)
  | ebIntBinExpr
    (op: ebIntBinOp, e1: ebIntExpr, e2: ebIntExpr)
```

We construct a similar inductive type in Dafny for defining the syntax of the real expressions.

Based on the syntax of the Event-B and B expressions, we then define the syntax for their predicates. For example, the following Dafny code defines the predicate for Event-B, which consists of: boolean literal predicate (true, false), unary predicate (not), binary predicate (logical and, or, implication and equivalence), and relational predicate for integers and reals (arithmetic equality, and comparisons):

```
datatype ebPred =
| ebLitPred(v:ebLitPredVal)
| ebUnPred(op:ebUnPredOp,p:ebPred)
| ebBinPred(bop:ebBinPredOp,p1:ebPred,p2:ebPred)
| ebIntRelPred
  (op:ebIntRelOp,e1:ebIntExpr,e2:ebIntExpr)
| ebRealRelPred
  (op:ebRealRelOp,e1:ebRealExpr,e2:ebRealExpr)
```

Then, we model the semantics for expressions and predicates of both languages in Dafny. For this task, we first define the evaluation context as a simple mapping from the identifier string to its value (either an integer or a real number):

```
datatype value = vint(i: int) | vreal(r: real)
type Context = map<string, value>
```

We then define an evaluation function in Dafny to encode how the Event-B and B expressions and predicates are interpreted given an evaluation context. For example, we can look up the value of an Event-B identifier expression in the given context by (which returns 0 if not found or not an integer identifier expression):

```
function method interp_ebIntExpr
  (e:ebIntExpr,ctx: Context):int
  decreases e
{ match e { ...
   case ebIntIdExpr(v) ⇒
    if v in ctx.Keys then
     if ctx[v].vint? then ctx[v].i else 0
    else 0
... }}
```

As we use an intermediate language to encode the semantics of source and target languages, we need to pay attention to potential semantic differences, and we rely on three language references to reduce the potential errors in this process [2, 17, 24].

6.2.2 Translation and Verification

The translation from the Event-B expressions/predicates to the B expressions/predicates is defined based on their corresponding syntax. It is a straightforward one-to-one mapping between the constructs of the two languages. We then verify that the specification of the Event-B software operator and its translation in B are semantically equivalent. The semantic equivalence is defined as:

```
function method compilePred(p:ebPred):abPred
  decreases p
  ensures ∀ ctx:Context •
    interp_abPred(compilePred(p),ctx)=
    interp_ebPred(p,ctx)
```

That is by giving the same context, the interpretation of the compiled B predicate yields the same result as the interpretation of the input Event-B predicate. It can be verified by case analysis on the input Event-B predicate, which is automatically proved in Dafny due to its automated induction capability [29].

7 Discussion

In addition to the car position tracking system, we also validate our approach to the design of a smart heating system that is used by [13]. The artifacts of this work are publicly and anonimously available on [14]. While the example shows the feasibility of our proposal, we identify the following points that are not yet addressed in our proposal for modeling hybrid systems.

Currently, we assume that the actuation command is sent from the control logic to the physical plant without disturbance. To ensure the system would behave correctly under actuation command interference, one needs to model that the designed control logic could produce a range of actuation commands (constrained by user-defined predicates). Then, it needs to prove that under any of these commands, the system should still be able to progress safely for the next cycle. Such a model for handling actuation command interference has a larger set of behaviors than the time-triggered design of [13], which requires modifying the initial abstraction of [13] to include it in the refinement.

When we design control logic in Event-B based on exact future predictions, it is straightforward and accurate when system dynamics have analytical solutions and initial conditions are known (like in the car position example). However, systems in real life rarely meet these criteria, which makes such an exact prediction more of an approximation. The problem is that when we use any approximation method in place of exact mathematical procedures, the truncation error occurs. Therefore, if we want to adapt control logic cases by approximations, we need to consider truncation errors carefully. The feasibility of bounding truncation error depends on various factors such as the complexity of system dynamics, or the chosen approximation method. Assuming that truncation errors are boundable, we need to introduce new refinement to the Event-B model defined in Section 3, which rewrites the designed control logic in terms of bounded error and the approximation result.

Atelier-B [27] defines a subset of B, called the B0 language. When a B component is encoded in this subset, it can be automatically translated to C programs using the Atelier-B code generator. One of the main restrictions of B0 is that it requires its program to only use integers. We think that we could introduce data-type conversion predicates in Event-B (e.g. real to integers), and bound the conversion error. This allows us to rewrite control logic in a similar way that deals with truncation errors. An alternative is to introduce float-point arithmetic in Event-B, and rewrite control logic based on that. Consequently, we can extend our verified translation to target B0 for reusing its code generator.

We use real numbers for modeling system state, which are not finitely representable on a computer. Languages/frameworks generally restrict their usage in the implementation. For example, Atelier-B requires its program logic to be defined in integers for using its code generator. To facilitate the implementation of software-based components, we should introduce data-type conversion predicates (e.g. real to integers), and bound the conversion error. This allows us to rewrite control logic in a similar way that deals with truncation errors. An alternative is to introduce float-point arithmetic, and rewrite control logic based on that.

To simplify our modeling, we assume that sensing/computation/actuation takes no time. However, most realistic systems take time for these tasks, and the time taken might not be neglectable. Thanks to the refinement introduced in Section 3, we could introduce delays in the corresponding system events for modeling the time taken for sensing/computation/actuation.

8 Related Work

Researchers have developed various techniques to concretize models for hybrid system designs. One approach is to write verified translations to generate control software directly from safe hybrid models [10, 42]. Its main difficulty is that the source and target languages might have quite different semantics. In this work, we adopt refinements to generate control software from safe hybrid models in two steps: 1) control software specification from safe hybrid models, 2) control software implementation from its specification. The first step involves several refinement steps to reduce the semantic gap between the source and target models, which simplifies the verification of their translation.

Various works design hybrid systems using Event-B [4, 7, 19–23, 39, 40]. They are broadly under the framework of the continuous action systems developed by Back et al. [6]. Our current work is highly influenced by these works. However, we do not aware of existing works on extending them for implementing designed Event-B models as we did in this work and we remain in the philosophy of the initial B method starting by an abstract machine which is progressively refined into an implementation machine. It plays the correct by construction paradigm but in a software perspective. The complete paper [15] gives a better description of the Event-B modelling language and of the B-software modelling language.

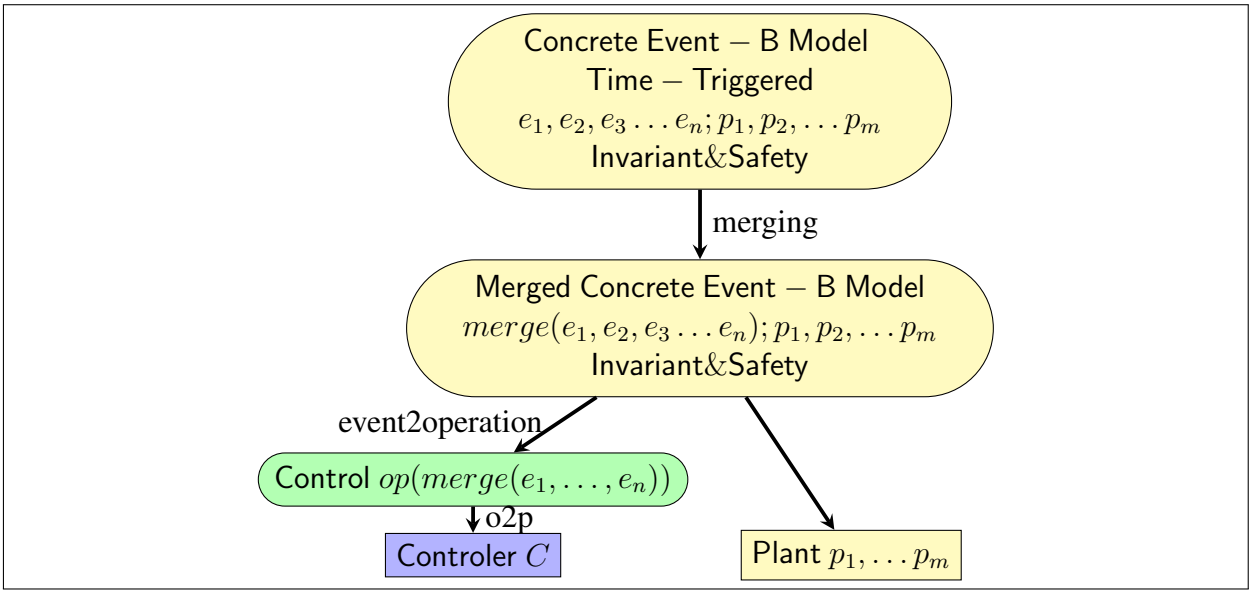
There are also several works on generating implementations from Event-B models (e.g. [12, 16, 25, 32–34, 41]), focused on different aspects of generating implementations for discrete algorithms, such as scheduling and arithmetic overflow. While we are investigating how these aspects can be integrated into our proposal, we think that hybrid programs offer at least 2 new properties to be considered in the generation process: 1) not all the events participate in the process. 2) events should be categorized and modularized for facilitating implementations.

Our proposal is based on a translational approach to the B-software language for its capabilities of refinement-based program construction and analysis. Koenig and Leino introduce programming language features for refinement in the Dafny language [26]. Sall et al. introduce a formalized theory for stepwise refinement of imperative programs in the Coq proof assistant [38]. Thus, these works potentially allow our translational approach to extend to more target languages.

9 Conclusion

In this work, we aim to extend [13] abstract Event-B models for implementing designed time-triggered Event-B models. The main steps of the transformation are sketched in the figure 1. First, we propose to refine a given time-triggered design with additional system events for sensing and actuation, which allow us to model the limitations of the real world in the corresponding system events and to construct more robust implementations. Then, we categorize all system events of a time-triggered design into the physical component and the software-based components. In addition, to soundly refine the software-based components down to low-level software implementations, we define a translational approach to the B-software language.

Our future works would focus on generating artifacts to validate the implementation produced by our translational approach, e.g., code for Frama-C [18] and Polyspace to check against certain industry code standards (e.g. absence of runtime error), or simulation models for Simulink and Stateflow to give a holistic view of the developed hybrid system.



Yellow nodes are Event-B models, Green nodes are B software models, Blue nodes are codes.

Figure 2: Merging Events into Operation

References

[1] Jean-Raymond Abrial. *The B book - Assigning Programs to Meanings*. Cambridge university press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[4] Meryem Afendi, Régine Laleau, and Amel Mammar. Modelling hybrid programs with Event-B. In *7th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 139–154. Springer, 2020.

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

[6] 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.

[7] 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.

[8] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2013.

[9] Dines Bjørner. *Domain Science and Engineering - A Foundation for Software Development*. Springer, 2021.

[10] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: verified controller executables from verified cyber-physical system models. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 617–630. ACM, 2018.

[11] Michael Butler and Issam Maamria. Mathematical extension in Event-B through the Rodin theory component, 2010.

[12] Néstor Catano and Víctor Rivera. Eventb2java: A code generator for event-b. In *NASA Formal Methods Symposium*, pages 166–171. Springer, 2016.

[13] Zheng Cheng and Dominique Méry. A refinement strategy for hybrid system design with safety constraints. In *10th International Conference on Model and Data Engineering*, pages 3–17. Springer, 2021.

[14] Zheng Cheng and Dominique Méry. From system events to software operations for refinement-based modeling of hybrid systems(online), 2022. DOI address: https://doi.org/10.5281/zenodo.7740748.

[15] Zheng Cheng and Dominique Méry. From System Events to Software Operations for Refinement-based Modeling of Hybrid Systems. working paper or preprint, March 2023.

[16] 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. Springer, 2016.

[17] ClearSy. *B Language reference manual ver.1.8.10*, 2022.

[18] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *10th International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[19] Guillaume Dupont, Yamine Aït-Ameur, Marc Pantel, and Neeraj K Singh. Hybrid systems and Event-B: a formal approach to signalised left-turn assist. In *8th International Conference on Model and Data Engineering*, pages 153–158. Springer, 2018.

[20] Guillaume Dupont, Yamine Aït-Ameur, Marc Pantel, and Neeraj Kumar Singh. Proof-based approach to hybrid systems development: dynamic logic and Event-B. In *6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 155–170. Springer, 2018.

[21] 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. IEEE, 2019.

[22] Guillaume Dupont, Yamine Ait-Ameur, Neeraj Kumar Singh, and Marc Pantel. Formally verified architectural patterns of hybrid systems using proof and refinement with Event-B. *Science of Computer Programming*, 216, 2022.

[23] 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. IEEE, 2019.

[24] Richard L. Ford and K. Rustan M. Leino. *Dafny Reference Manual*, 2017.

[25] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code generation for Event-B. In *International Conference on Integrated Formal Methods*, pages 323–338. Springer, 2014.

[26] Jason Koenig and Rustan Leino. Programming language features for refinement. Technical report, arXiv preprint, 2016.

[27] Thierry Lecomte. Atelier-B. *Formal Methods Applied to Complex Systems: Implementation of the B Method*, pages 35–46, 2014.

[28] 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. Springer, 2010.

[29] K Rustan M Leino. Automating induction with an SMT solver. In *13th International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[30] Sarah M Loos and André Platzer. Differential refinement logic. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 505–514. ACM, 2016.

[31] Amel Mammar, Meryem Afendi, and Régine Laleau. Modeling and proving hybrid programs with Event-B: An approach by generalization and instantiation. *Science of Computer Programming*, page 102856, 2022.

[32] Dominique Méry. Playing with state-based models for designing better algorithms. *Future Generation Computer Systems*, 68:445–455, 2017.

[33] Dominique Méry and Rosemary Monahan. Transforming event B models into verified C# implementations. In *1st International Workshop on Verification and Program Transformation*. EPiC Series in Computing, 2013.

[34] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *2nd symposium on information and communication technology*, pages 179–188, 2011.

[35] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *25th International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[36] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[37] Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *International Journal on Software Tools for Technology Transfer*, 18(1):67–91, 2016.

[38] Boubacar Demba Sall, Frédéric Peschanski, and Emmanuel Chailloux. A mechanized theory of program refinement. In *21st International Conference on Formal Engineering Methods*, pages 305–321. Springer, 2019.

[39] Paulius Stankaitis, Guillaume Dupont, Neeraj Kumar Singh, Yamine Ait-Ameur, Alexei Iliasov, and Alexander Romanovsky. Modelling hybrid train speed controller using proof and refinement. In *24th International conference on engineering of complex computer systems*, pages 107–113. IEEE, 2019.

[40] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing hybrid systems with event-b and the rodin platform. *Sci. Comput. Program.*, 94:164–202, 2014.

[41] Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. From Event-B specifications to programs for distributed algorithms. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 104–109. IEEE, 2013.

[42] Gaogao Yan, Li Jiao, Shuling Wang, Lingtai Wang, and Naijun Zhan. Automatically generating SystemC code from HCSP formal models. *ACM Transactions on Software Engineering and Methodology*, 29(1):1–39, 2020.