# 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 and Dominique Méry

Université de Lorraine, LORIA, France.

**Abstract.** To reduce error-prone work of implementing hybrid system designs in Event-B by hand, we revisit the refinement methodology to systematically identify, categorize and modularize software operators from system events. Then, by defining a verified translation, we can obtain a semantically equivalent correspondent in the B language for each modularized Event-B software operator. Thus, we can reuse the primitives of B (which are superset of Event-B) for refining system events down to implementations, and reuse the predicate transformers defined on the B primitives to reason about the correctness of refinements. The verified translation 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. We evaluate our proposal on two case studies, and discuss the lessons learned.

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

## 1 Introduction

Various tools and formalisms are proposed to provide high-level safety guarantees for hybrid systems at the design level [6, 25, 36]. 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

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

by the Rodin platform [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.

There are many works on design hybrid systems w.r.t. the given safety properties using Event-B [4, 7, 13, 18–22, 30, 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, 15, 24, 31–33, 41], deriving implementations from hybrid system designs in Event-B requires 2 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 [21], 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 asso-

ciated system events of each software-based component. Then, by defining a verified translation, we can obtain a semantically equivalent correspondent in the B language for each modularized Event-B software operator. 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 about 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.

## 2 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 $\delta$ 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.

In [13], Cheng and Méry develop a refinement strategy to model such hybrid systems in Event-B (we refer to Appendix A for a short

introduction of Event-B). 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 number) to the system state domain $D$. In our example, we thus have $p, v : \mathbb{R}^+ \rightarrow \mathbb{R}$ to represent the car position and velocity respectively.
- $now$ 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.
- $s$ 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 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 $\delta$ from $now$ ($thm_1$). For example, the car example yields 3 $Prediction$ events[1]:

[1] 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 $v_{u_i}(t) = v(now) + u_i(t - now)$.

**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** ...
$act_1$: $u, t_u := u_i, \delta$
$act_2$: $s := RUN$
**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 1.1.** 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* $(act_1)$[2]. *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 [21]. 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 [21] 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 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

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

specified Event-B software operator into a B operation (Listing 1.3)[3]. 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[4]. 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].

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

**Listing 1.2.** A specified Event-B software operator for the discrete controller of the car position tracking example

$u_d, t_u \ \leftarrow \ car\_ctrl(ps, vs) \ =$
**Pre**
$ps \in Real \ \wedge \ vs \in Real \ \wedge \ u_d \in U_d \ \wedge \ t_u \in Real$
**Then**
$u_d, t_u \ : \ ( \ \Big(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} \leq S \Rightarrow u_d = ACC \wedge t_u = \delta\Big) \wedge$
$\Big(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} > S \wedge v = 0 \Rightarrow u_d = STOP \wedge t_u = \delta\Big) \wedge$
$\Big(ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} > S \wedge v \neq 0 \Rightarrow u_d = BRAKE \wedge t_u = \delta\Big) \ )$
**End**

**Listing 1.3.** The translated B operation of Listing 1.2

[3] 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 operation, we refer to the standard operation construct in the B language.

[4] 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) =
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 1.4.** The implemented B operation of Listing 1.3

## 3 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 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 $ss \in \{SENSE, CONTROL, ACT-$

$UATE, PLANT\}$, 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.

### 3.1 *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.

**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 1.5.** 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 1.6.** 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 1.7.** The *Actuate* event

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 $\epsilon_p$ and $\epsilon_v$ 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.

### 3.2 *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 $C_i(p, v) \mathrel{\widehat{=}} p_A(now + \delta) + \frac{v_A(now+\delta)^2}{2B} \leq S$, 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 $CC_i(ps, vs) \mathrel{\widehat{=}} ps + vs\delta + \frac{1}{2}A\delta^2 + \frac{(vs+A\delta)^2}{2B} \leq S$, 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 $u_d \in \{ACC, STOP, BRAKE\}$.

### 3.3 *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 $R_a(u_d, u)$:

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

### 3.4 *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.

### 3.5 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]).

## 4 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).

### 4.1 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.

**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) [5].

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** $Control_i \ \widehat{=}$
**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 1.8.** The refined *Control* event of Listing 1.6 in Event-B

**Operator** $f_c \ \widehat{=}$
**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 1.9.** The template for encapsulating a specified Event-B software operator for the controller component

**Event** $Control \ \widehat{=}$
**Refines** $Control_1$ , ... , $Control_i$
**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 1.10.** The merged event for the controller component in Event-B

**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

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

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 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.

**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.

### 4.2 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.

**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 [34], Isabelle [35]).

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, 16, 23].

**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].

## 5 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 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 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.

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.

## 6 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, 18–22, 39, 40]. They are broadly under the framework of the continuous action system 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.

There are also several works on generating implementations from Event-B models (e.g. [12, 15, 24, 31–33, 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 hy-

brid 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 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.

## 7 Conclusion

In this work, we aim to extend [13] for implementing designed time-triggered Event-B models. 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 language.

Our future works would focus on generating artifacts to validate the implementation produced by our translational approach, e.g., code for Frama-C [17] 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.

# References


1. Abrial, J.R.: The B book - Assigning Programs to Meanings. Cambridge university press (1996)
2. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: 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. Afendi, M., Laleau, R., Mammar, A.: Modelling hybrid programs with Event-B. In: 7th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 139–154. Springer (2020)
5. Ameur, Y.A., Méry, D.: Making explicit domain knowledge in formal system development. Science of Computer Programming 121, 100–127 (2016)
6. Back, R.J., Petre, L., Porres, I.: Continuous action systems as a model for hybrid systems. Nordic Journal of Computing 8(1), 2–21 (2001)
7. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Science of Computer Programming 105, 92–123 (2015)
8. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer (2013)
9. Bjørner, D.: Domain Science and Engineering - A Foundation for Software Development. Springer (2021)
10. Bohrer, R., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: Veriphy: verified controller executables from verified cyber-physical system models. In: 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 617–630. ACM (2018)
11. Butler, M., Maamria, I.: Mathematical extension in Event-B through the Rodin theory component (2010)
12. Catano, N., Rivera, V.: Eventb2java: A code generator for event-b. In: NASA Formal Methods Symposium. pp. 166–171. Springer (2016)
13. Cheng, Z., Méry, D.: A refinement strategy for hybrid system design with safety constraints. In: 10th International Conference on Model and Data Engineering. pp. 3–17. Springer (2021)
14. Cheng, Z., Méry, D.: From system events to software operations for refinement-based modeling of hybrid systems(online) (2022), https://doi.org/10.5281/zenodo.7740747
15. Cheng, Z., Méry, D., Monahan, R.: 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. pp. 821–838. Springer (2016)
16. ClearSy: B Language reference manual ver.1.8.10 (2022)
17. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: 10th International conference on software engineering and formal methods. pp. 233–247. Springer (2012)
18. Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K.: Hybrid systems and Event-B: a formal approach to signalised left-turn assist. In: 8th International Conference on Model and Data Engineering. pp. 153–158. Springer (2018)
19. Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K.: 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. pp. 155–170. Springer (2018)

20. Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K.: Handling refinement of continuous behaviors: a proof based approach with Event-B. In: 13th International Symposium on Theoretical Aspects of Software Engineering. pp. 9–16. IEEE (2019)
21. Dupont, G., Ait-Ameur, Y., Singh, N.K., Pantel, M.: Formally verified architectural patterns of hybrid systems using proof and refinement with Event-B. Science of Computer Programming 216 (2022)
22. Dupont, G., Ameur, Y.A., Pantel, M., Singh, N.K.: Handling refinement of continuous behaviors: A proof based approach with Event-B. In: 13th International Symposium on Theoretical Aspects of Software Engineering. pp. 9–16. IEEE (2019)
23. Ford, R.L., Leino, K.R.M.: Dafny Reference Manual (2017)
24. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: International Conference on Integrated Formal Methods. pp. 323–338. Springer (2014)
25. J., H.: From CSP to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare. pp. 171—-189. Prentice Hall (1994)
26. Koenig, J., Leino, R.: Programming language features for refinement. Tech. rep., arXiv preprint (2016)
27. Lecomte, T.: Atelier-B. Formal Methods Applied to Complex Systems: Implementation of the B Method pp. 35–46 (2014)
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: 17th International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
29. Leino, K.R.M.: Automating induction with an SMT solver. In: 13th International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 315–331. Springer (2012)
30. Mammar, A., Afendi, M., Laleau, R.: Modeling and proving hybrid programs with Event-B: An approach by generalization and instantiation. Science of Computer Programming p. 102856 (2022)
31. Méry, D.: Playing with state-based models for designing better algorithms. Future Generation Computer Systems 68, 445–455 (2017)
32. Méry, D., Monahan, R.: Transforming event B models into verified C# implementations. In: 1st International Workshop on Verification and Program Transformation. EPiC Series in Computing (2013)
33. Méry, D., Singh, N.K.: Automatic code generation from event-b models. In: 2nd symposium on information and communication technology. pp. 179–188 (2011)
34. Moura, L.d., Kong, S., Avigad, J., Doorn, F.v., Raumer, J.v.: The Lean theorem prover (system description). In: 25th International Conference on Automated Deduction. pp. 378–388. Springer (2015)
35. Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)
36. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer (2018)
37. Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A.: 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. Sall, B.D., Peschanski, F., Chailloux, E.: A mechanized theory of program refinement. In: 21st International Conference on Formal Engineering Methods. pp. 305–321. Springer (2019)
39. Stankaitis, P., Dupont, G., Singh, N.K., Ait-Ameur, Y., Iliasov, A., Romanovsky, A.: Modelling hybrid train speed controller using proof and refinement. In: 24th International conference on engineering of complex computer systems. pp. 107–113. IEEE (2019)

40. Su, W., Abrial, J.R., Zhu, H.: Formalizing hybrid systems with Event-B and the Rodin platform. Science of Computer Programming 94, 164–202 (2014)
41. Tounsi, M., Mosbah, M., Méry, D.: From Event-B specifications to programs for distributed algorithms. In: 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. pp. 104–109. IEEE (2013)
42. Yan, G., Jiao, L., Wang, S., Wang, L., Zhan, N.: Automatically generating SystemC code from HCSP formal models. ACM Transactions on Software Engineering and Methodology 29(1), 1–39 (2020)

## Appendix A: Introduction to Event-B

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 1.11) 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**).

```
Context c1
Extends c2
Sets S
Constants C
Axioms Ac
Theorems Tc
End
```

**Listing 1.11.** Abstract syntax of Event-B contexts

```
Machine m1
Refines m2
Sees c1
Variables V
Invariants I
Events E
End
```

**Listing 1.12.** Abstract syntax of Event-B machines

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

**Listing 1.13.** Abstract syntax of Event-B events

Each machine (abstract syntax shown in Listing 1.12) 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 un-

der **Invariants**). State variables are only modifiable through events (specified under **Events**).

Each event (abstract syntax shown in Listing 1.13) 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 $x :\mid P(x, x')$, 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 $P$[6].

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

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.

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 1.14.

[6] 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.

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

**Listing 1.14.** Abstract syntax of Event-B operators developed by the Theory plugin

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**).