Standalone Event-B models analysis relying on the EB4EB meta-theory

Peter Riviere, Neeraj Kumar Singh, Yamine Aït-Ameur, Guillaume Dupont

Standalone Event-B models analysis relying on the EB4EB meta-theory

P. Rivière[1], N. K. Singh[1], Y. Aït-Ameur[1], and G. Dupont[1] *

INPT-ENSEEIHT/IRIT, University of Toulouse, France
{peter.riviere, nsingh, yamine, guillaume.dupont}@enseeiht.fr

Abstract. Event-B is a state-based correct-by-construction system design formal method relying on proof and refinement where system models are expressed using set theory and First Order Logic (FOL). Through the generation and discharging of proof obligations (POs), Event-B natively supports the establishment of properties such as safety invariant, convergence and refinement. Other properties, relevant to system verification, may be studied as well, but need to be explicitly formalised by the designer, or expressed in another formal method. This process compromises reusability and is error-prone, especially on larger systems. Recently, the reflexive EB4EB framework has been proposed for formalising Event-B concepts as first-class objects. It allows manipulating these concepts using FOL and set theory in Event-B. In this paper, we propose a rigorous methodology for extending the EB4EB framework, to support new system analysis mechanisms associated to properties that are not natively present in core Event-B. Thanks to the reflexive nature of this framework, new generic and reusable system properties and their associated POs are expressed once and for all, and for any refinement level. For specific systems, designers instantiate these properties and the associated POs are automatically generated and submitted to Event-B's provers. This methodology is used to define three analyses: deadlock-freeness, invariant weakness analysis and reachability, all of which are demonstrated on a case study.

Keywords: Reflection · Refinement and Proof · Meta-theory · Reachability · Deadlock-Freeness · Invariant weakness · EB4EB framework · Event-B.

1 Introduction

Context. The refinement and proof state-based Event-B formal method [1] supports complex system development using a correct-by-construction approach. It is based on set theory and First Order Logic (FOL) for describing state transition systems. It relies on an inductive proof process to discharge a set of proof obligations (POs) expressing various properties. Basically, core Event-B offers built-in modelling constructs to express invariants, event convergence, simulation, guard

* The authors thank the ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project.

strengthening and event feasibility. POs associated to these constructs are automatically generated and are discharged using automatic and interactive provers.

In order to enrich the method's expressiveness, Event-B has been extended with the ability to define new algebraic data-types resulting in a richer type system [2, 11], through the introduction of *Theories*. This extension allows the formalisation of complex systems at a higher level of abstraction.

Motivation. Event-B theories make it possible to formalise new data types, but *they do not allow the definition of new POs* that correspond to properties other than the usual ones (i.e., invariants preservation, event convergence, etc.).

Indeed, when properties such as deadlock-freeness, event scheduling, liveness, and so on need to be proved, they are explicitly formalised by the designer, or expressed in another formal method. This process compromises reusability and is error-prone, especially on large systems. The designer shall formalise each desired property for each system under design using the native Event-B POs. This process may be cumbersome, must be repeated for each model to be analysed (not reusable) and results in formal developments scattered across multiple heterogeneous frameworks and semantics.

To incorporate such properties in Event-B once and for all and allow the automatic generation of property-specific POs, it is necessary to embed, in the Event-B engine, the POs associated to these new properties. Such embedding requires the manipulation, in Event-B, of Event-B concepts as first-order objects (i.e., through a reflexive meta-model). We have recently proposed a reflexive EB4EB framework [29,30] that formalises Event-B concepts as first-class objects in Event-B. It allows manipulating these concepts in Event-B using first-order logic and set theory. It is built on an algebraic meta-theory formalised as an Event-B theory, where each Event-B feature can be handled at the meta-model level, as first-class citizen. This framework also formalises Event-B's trace-based semantics and offers constructs for machines, states, and events together with a set of operators for manipulating them. Consequently, the EB4EB framework makes it possible to formally express, at any abstraction level (i.e. in the refinement chain), new reusable and automatically generated POs and high-level constructs, easing the development of complex systems with specific properties or semantics. Furthermore, it opens the door to formally embed Event-B's semantics in other formal methods and exploit their respective strengths.

Objective of this paper. This paper extends and enriches our previously developed EB4EB framework [29, 30] to support new analysis mechanisms (possibly non-intrusive), formalised as logic properties not available in native Event-B nor in its base PO generator. It extends the EB4EB Event-B meta-theory with new operators formalising such new properties. The POs associated to each operator are automatically generated. Adding the desired property, corresponding to a specific analysis, to an Event-B model is performed by invoking an operator. Designers do not need to formalise this property explicitly in the model.

Structure of the paper. The paper is organised as follows. Section 2 describes the Event-B method and the Theory mathematical extension. Section 3 introduces the EB4EB framework and its Event-B meta-theory, as well as the case study

Context	Machine	Theory
CONTEXT Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G_i(x,\alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**	**THEORY** Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATATYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well—definedness** $WD(p_1, ...)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well—definedness** $WD(p_1, ...)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **PROOF RULES** R_1, ... **END**
(a)	(b)	(c)

Table 1: Global structure of Event-B Contexts, Machines and Theories

(1) Ctx Theorems (ThmCtx)	$A(s,c) \Rightarrow T_{ctx}$ `(For contexts)`
(2) Mch Theorems (ThmMch)	$A(s,c) \wedge I(x) \Rightarrow T_{mch}(x)$ `(For machines)`
(3) Initialisation (Init)	$A(s,c) \wedge BAP(x') \Rightarrow I(x')$
(4) Invariant preservation (Inv)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow I(x')$
(5) Event feasibility (Fis)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \Rightarrow \exists x' \cdot BAP(x,\alpha,x')$
(6) Variant progress (Var)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow V(x') < V(x)$

Table 2: Relevant POs for Event-B contexts and machines

used throughout this paper. Three externally defined Event-B analyses and POs are introduced in Section 4 and applied to the case study. The positioning of this work with respect to the state of the art and its advantages are discussed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 Event-B

Event-B [1] is based on set theory and FOL. It relies on an expressive state-based modelling language where a set of events models state changes.

2.1 Contexts and machines (Tables 1.a and 1.b)

A `Context` (Table 1.a) describes the static part of a model. It introduces *carrier sets* s and *constants* c, and their properties using *axioms* A and *theorems* T_{ctx}. A `Machine` (Table 1.b) describes the model behaviour as a transition system. A set of *events* evt, possibly guarded by G and/or parameterized by α, is used to modify a set of state variables x using Before-After Predicates (BAP) to record state changes. A machine may define *invariants* $I(x)$, *theorems* $T_{mch}(x)$ and *variants* $V(x)$ to capture particular properties (e.g., safety and convergence). Model consistency is ensured via a set of generated POs, given in Table 2.

Refinements. Refinement decomposes a *machine* into a less abstract one with more design decisions (refined states and events) moving from an abstract level to

a less abstract one (simulation relationship). Gluing invariants relating abstract and concrete variables ensure property preservation.

Core Well-definedness (WD). In addition to machine-related POs, each operator is associated to a *WD*, that must be established for expressions to be meaningful. Once proved, these WD conditions are used as hypotheses to prove further POs.

2.2 Event-B extensions with Theories

To handle more complex and abstract concepts beyond set theory and FOL, an Event-B extension for externally defined mathematical objects has been proposed [2, 11]. It introduces user data types with new types, operators, theorems and associated rewrite and inference rules, all bundled in so-called *theories*. Close to proof assistants like Coq [5], Isabelle/HOL [25] or PVS [26], this capability is convenient to model, as data types, *concepts unavailable in core Event-B*.

Theory description (See Table 1.c). Theories define new data types, operators, and theorems. Data types (`DATATYPES` clause) define *constructors* to build inhabitants of the defined type. It may define various *operators* further used in Event-B expressions as FOL *predicates* or *expressions* producing actual values (<nature> tag). Operators may be used in theories, contexts and machines.

Operators may be defined explicitly in the `DIRECT DEFINITION` clause (constructive definition), or axiomatically in the `AXIOMATIC DEFINITIONS` clause (a set of axioms). Last, a theory defines a set of axioms, completing the definitions, as well as theorems and proof rules. Theorems and proof rules are proved from the definitions and axioms used by the proof system. Many theories have been defined for sequences, lists, groups, reals, differential equations, etc.

Well-definedness (WD) in Theories. An important feature provided by Event-B theories is the possibility to define *Well-Definedness* (WD) conditions (close to Type-Correctness Condition (TCC) conditions in PVS [26]). TCC must be discharged before the corresponding theory types correctly. Similarly, in Event-B theories, each defined operator (thus partially defined) is associated with a user-defined condition ensuring its well-formedness. Note that, when an operator is applied, it automatically invokes its WD condition and generates a PO requiring to establish that this condition holds, i.e., the operator is used correctly and that its parameters belong to its definition domain.

Event-B proof system and its IDE Rodin. Rodin is an open source IDE for modelling in Event-B. It offers resources for model editing, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. The Event-B theories extension is available as a plug-in. Theories are tightly integrated in the proof process. Depending on their definition (direct or axiomatic), operator definitions are expanded either using their direct definition (if available) or by enriching the set of axioms (hypotheses in proof sequents) using their axiomatic definition. Theorems may be imported as hypotheses and used in proofs like other theorems. Many provers for first-order logic as well as SMT solvers are plugged to Rodin for helping the proof process.

3 The EB4EB framework

The main objective of the EB4EB reflexive framework [29, 30] is to provide explicit manipulation of Event-B components as first-class objects, making it possible to reason on these objects and define new Event-B analyses. For this purpose, the concept of Event-B machine is formalised as a data-type in a theory (a meta-theory), together with a set of operators that guarantee the correctness, relative to Event-B semantics, of instances of this data-type. The meta-theory formalises the semantics of Event-B, as described in the Event-B Book [1], i.e. a set of states and guarded events defined as a relation between states. In addition, the meta-theory is equipped with relevant proved (once and for all) theorems useful for discharging the generated POs. These additional theorems are available to help users reduce proof efforts and aid in system development and analysis.

Event-B machines (models) are defined using the meta-theory mentioned above, by instantiating the machine data-type and providing appropriate values for each of its fields: states, events, guards, before-after predicates, invariants, variant and so on. At instantiation, operators of the meta-theory are used in theorems; the related POs ensure the defined machine's consistency, including invariant preservation, event feasibility, variant progress, etc.

As previously stated, the goal of this paper is to demonstrate that the meta-theory can be extended with new operators for manipulating machine elements of the meta-theory, in order to define so-called *analyses*, expressed with new POs. Based on the work presented in [3], such analyses allow the system designer to check new properties, obtain feedback about their behaviour, enrich model design phases and check new properties that are not available in core Event-B.

This section summarises the main features of the Event-B meta-theory (Listings 1, 2 and 3), and presents the case study used to illustrate our approach throughout this paper.

3.1 The Event-B Meta-theory

Machine structure. Listing 1 shows the `Machine` data-type, defined using type parameters for abstracting event labels (`EVENTS`) and states (`STATES`). It is built using the `Cons_machine` single constructor with a parameter for each machine component, and defines a state-transition system with state *State* (constrained by invariants *Inv* and theorems *Thm*) and a set of, possibly parameterised, events (*Event*), with an initialisation event *Init* and progress events *Progress*, split into ordinary *Ordinary* and convergent *Convergent* events. State changes are

```
THEORY EvtBTheo
TYPE PARAMETERS STATE, EVENT
DATATYPES
  Machine(STATE, EVENT)
 CONSTRUCTORS
   Cons_machine(
     Event : ℙ(EVENT),
     State : ℙ(STATE),
     Init : EVENT,
     Progress : ℙ(EVENT)
     AP : ℙ(STATE),
     Grd : ℙ(EVENT × STATE),
     BAP : ℙ(EVENT × (STATE × STATE)),
     Inv : ℙ(STATE)
     Thm : ℙ(STATE),
     Variant : ℙ(STATE × ℤ),
     Ordinary : ℙ(EVENT),
     Convergent : ℙ(EVENT))
```

Listing 1: Machine Data-type

recorded using an *after-predicate* (AP) for initialisation and a set of *before-after predicates* (BAP) associated to progress events, possibly guarded (Grd). Finally, integer variants for event convergence are introduced as well ($Variant$).

Well-Constructed machines. To ensure machines are structurally well-defined, the meta-theory introduces several predicate operators (Listing 2): `BAP_WellCons` to check that each progress event is associated to a BAP, `Grd_WellCons` to check that progress events are possibly guarded, and `Event_WellCons` to check that machine events are composed of an initialisation (`Init`) and progress (`Progress`) events. The `Machine_WellCons` predicate operator, defined as a conjunction of the previous operators (and others), ensures that a machine is well-structured (static semantics).

BAP_WellCons <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition $dom(BAP(m)) = Progress(m)$
Grd_WellCons <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition $dom(Grd(m)) = Progress(m)$
Event_WellCons <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition $partition(Event(m), \{Init(m)\}, Progress(m))$
  ...
Machine_WellCons <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $BAP\_WellCons(m) \land Grd\_WellCons(m) \land Event\_WellCons(m) \land \ldots$

Listing 2: Operators to check well-defined data-type (static semantics)

Machine POs (behavioural semantics). The `Machine` data-type offers operators to access and handle its components. In addition to structural consistency, machine correctness is also encoded, through its behavioural semantics and correctness criteria. Formally, this is done by providing an operator for each PO of Event-B (see Table 2), as shown in Listing 3. Such operators are usually defined inductively on the structure of a machine (for initialisation and progress events).

Mch_THM <*predicate*> ...
Mch_INV_Init <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition $AP(m) \subseteq Inv(m)$
Mch_INV_One_Ev <*predicate*> $(m : Machine(STATE, EVENT), e : EVENT)$
  well−definedness $e \in Progress(m)$
  direct definition $BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$
Mch_INV <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $Mch\_INV\_Init(m) \land (\forall e \cdot e \in Progress(m) \Rightarrow Mch\_INV\_One\_Ev(m, e))$
Mch_FIS_Init <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition $Inv(m) \cap AP(m) \neq \emptyset$
Mch_FIS_One_Ev <*predicate*> $(m : Machine(STATE, EVENT), e : Event)$
  well−definedness $e \in Progress(m)$
  direct definition $Inv(m) \cap Grd(m)[\{e\}] \subseteq dom(BAP(m)[\{e\}])$
Mch_FIS <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $Mch\_FIS\_Init(m) \land (\forall e \cdot e \in Progress(m) \Rightarrow Mch\_FIS\_One\_Ev(m, e))$
Mch_VARIANT_One_Ev <*predicate*> ...
Mch_VARIANT <*predicate*> ...
Mch_NAT_One_Ev <*predicate*> ...
Mch_NAT <*predicate*> ...

Listing 3: Well defined Data-type operators (behavioural semantics)

The details of the invariant preservation (INV - 3 and 4 in Table 2) and feasibility (FIS - 5 in Table 2) POs are shown in Listing 3. Three operators are associated to the definition of these POs: `Mch_INV_Init`, stating that an invari-

ant holds at initialisation (i.e., states after the AP are included in the invariant states, $AP(m) \subseteq Inv(m)$); `Mch_INV_One_Ev`, stating that any event e characterised by its guard and BAP preserves the invariant (e.g. the image of invariant states through BAP is included in invariant states, $BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$); and `Mch_INV`, the conjunction of these two operators, where `Mch_INV_One_Ev` must hold for all progress events. Similarly, three operators `Mch_FIS_Init`, `Mch_FIS_One_Ev` and `Mch_FIS_Init` define the event feasibility PO (existence of a next state after AP or BAP of progress events). The other POs in Table 2 are defined in the same manner.

The POs of an Event-B machine are gathered in the conjunctive predicate `check_Machine_Consistency`, with `Machine_WellCons` as well-definedness (see Listing 4). It formalises machine's behavioural semantics and general correctness.

check_Machine_Consistency <*predicate*> $(m : Machine(STATE, EVENT))$
  well−definedness Machine_WellCons(m)
  direct definition $Mch\_THM(m) \wedge$
  $Mch\_INV(m) \wedge Mch\_FIS(m) \wedge$
  $Mch\_VARIANT(m) \wedge Mch\_NAT(m)$

Listing 4: Operator encoding Event-B machine consistency

When this operator is used in a `theorem` clause, two POs, corresponding to its definition and WD condition, are automatically generated. Proving the theorem ensures the consistency of the machine, defined as an instance of the meta-theory.

Instantiation of the meta-theory. Specific Event-B machines are defined by instantiating the meta-theory. The instantiation process presented in this paper is so-called *deep*, as it relies *solely* on set theory and FOL with a set of axioms and theorems. It consists in defining an Event-B context with witnesses (sets) for type parameters `STATE` and `EVENT` defined as sets using `Cons_machine`. Operators may be used in theorems, triggering the generation of POs ensuring machine consistency. Another instantiation process qualified as *shallow* has also been defined [29,30]. It relies on the definition of an Event-B machine and its refinement. It is not reviewed here as it is not used in this paper.

3.2 The Clock Example

This section presents a case study adapted from Lamport's clock case study [19]. It is used to demonstrate the application of the proposed framework, including meta-theory instantiation and definition of new POs. Note that this simple case study is chosen to demonstrate the usability of the new extended mechanism.

The functional requirements of the clock state that minutes and hours progress by 1 and hours are represented in a 24-hour format. The clock must converge to midnight, and never stop. Listing 5 gives a model of the clock as an Event-B machine. In this model, variables m and h represent minutes and hours, respectively. A safety property ($inv2$) ensures that minutes m (resp. hours h) are always less than 60 (resp. 24). The clock's behaviour is expressed through three events: `tick_min` (progressing minutes by 1), `tick_hours` (progressing hours by 1) and `tick_midnight` (resetting the clock to midnight).

```
MACHINE Clock
VARIABLES m, h
INVARIANTS
  inv1: m ∈ ℕ ∧ h ∈ ℕ
  inv2: m < 60 ∧ h < 24
EVENTS
  INITIALISATION
  THEN act1: m, h :| m' = 0 ∧ h' = 0
  END
  tick_min
  WHERE grd1: m < 59
  THEN act1: m :| m' = m + 1
  END
  tick_hour
  WHERE grd1: m = 59 ∧ h < 23
  THEN act1:
       m, h :| m' = 0 ∧ h' = h + 1
  END
  tick_midnight
  WHERE grd1: m = 59 ∧ h = 23
  THEN act1: m, h :| m' = 0 ∧ h' = 0
  END
END
```

Listing 5: Clock as Event-B machine

```
CONTEXT ClockMachineInstance
SETS Ev, ℤ × ℤ
CONSTANTS clock, tick_min, tick_hour,
    tick_midnight, init
AXIOMS
  axm1: clock ∈ Machine(ℤ × ℤ, Ev)
  axm2: partition(Ev, {init}, {tick_midnight},
        {tick_hour}, {tick_min})
  axm3: State(clock) = ℤ × ℤ
  axm4: Event(clock) = Ev
  axm5: Init(clock) = init
  axm6: Inv(clock) = {m ↦ h | m ∈ ℕ ∧ h ∈ ℕ
        ∧ m < 60 ∧ h < 24}
  axm7: AP(clock) = {m ↦ h | m = 0 ∧ h = 0}
  axm8: Grd(clock) = {e ↦ (m ↦ h) |
      (e = tick_min ∧ m < 59)∨
      (e = tick_hour ∧ m = 59 ∧ h < 23)∨
      (e = tick_midnight ∧ m = 59 ∧ h = 23)}
  axm9: BAP(clock) =
      {e ↦ ((m ↦ h) ↦ (m' ↦ h')) |
      (e = tick_min ∧ m' = m + 1 ∧ h' = h)∨
      (e = tick_hour ∧ m' = 0 ∧ h' = h + 1)∨
      (e = tick_midnight ∧ m' = 0 ∧ h' = 0)}
  ...
THEOREMS
  thm1: check_Machine_Consistency(clock)
END
```

Listing 6: Clock as meta-theory instance

While the previous example does not show parameterised events, however, our approach handles such events. The same approach has been successfully applied to complex case studies in [21] for critical interactive systems.

3.3 The clock machine as an instance of *EvtBTheo* theory

Listing 6 shows the Event-B context `ClockMachineInstance` instantiating the meta-theory `EvtBTheo`. First, $axm1$ defines the *clock* machine with the sets Ev (set of events enumerated in $axm2$) and $\mathbb{Z}\times\mathbb{Z}$ (for m and h). $axm3-axm9$ define associated machine components. Note that invariant is defined ($axm6$) on the state as a set of pairs $m \mapsto h$, AP is defined on the initialisation event $axm7$ and guards and BAPs are associated with an event and a state and defined ($axm8$ and $axm9$) on a set of triples $e \mapsto m \mapsto h$. In the case of BAPs, it is necessary to record before $(m \mapsto h)$ and after $(m' \mapsto h')$ states ($axm9$).

Last, theorem $thm1$ uses `check_Machine_Consistency` (see Listing 4). It is associated with a well-definedness (WD) PO, $Machine\_WellCons(clock)$, and a theorem (THM) PO for machine correctness.

4 POs for new properties: Extending the Meta-Theory

The meta-theory *EvtBTheo* presented in Section 3.1 is highly extensible: every Event-B feature is explicitly formalised, and can be manipulated using operators, making it possible to define specific development operations or new reasoning

mechanisms as new operators. Doing so is *non-intrusive* (self-contained), in the sense that no modification is needed to the classical development of Event-B models, as machines are handled as instances of the meta-theory.

The main design principle for such Event-B machine analyses, including theories with required operators, definitions, and WD conditions, is given below.

4.1 Analysis principle: New POs

In the proposed extension to the EB4EB framework, a model analysis is defined as a PO and must meet two requirements: 1) it must be reusable, and 2) it must be generated automatically. The first requirement is met by formalising the PO at the meta-theory level, while the second one is met by leveraging automatically generated well-definedness (WD) and theorem (THM) POs.

Event-B machine analysis pattern. Listing 7 depicts a generic pattern for defining new POs for Event-B machine analysis. *Theo4PO* theory imports the meta-theory *EvtBTheo* and introduces a third, optional type parameter T_{Args} possibly needed by the analysis, depending on the nature of new POs (e.g. guards, BAP, etc.). The PO associated to the analysis is formalised as a predicate operator `[PO]_Definition`. Then, checking the PO is done using the `check_Machine[PO]` predicate, which is well-defined when machine m is consistent.

```
THEORY Theo4PO IMPORT EvtBTheo
TYPE PARAMETERS STATE, EVENT, T_Args
OPERATORS
  [PO]_Definition <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition ...
    direct definition ...
  check_Machine_[PO] <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition Machine_WellCons(m)
    direct definition [PO]_Definition(m, args)
END
```

Listing 7: Analyses Theory Pattern

Checking PO context pattern. Listing 8 shows an Event-B context pattern for checking the newly defined PO. A consistent instance machine context *MachineInstance*, that defines the Event-B machine m by instantiation of the meta-theory *EvtBTheo*, is extended by context *MachinePO* instantiating the extended theory *Theo4PO*. Theorem *thmPO* performs the check of the defined PO for machine m. The associated WD and THM POs are automatically generated.

```
CONTEXT MachinePO
EXTENDS MachineInstance
THEOREMS
  thmPO: check_Machine_[PO](m, args)
END
```

Listing 8: Analyses Machine

Following this idea, this section introduces new reasoning mechanisms, not natively present in Event-B, based on the EB4EB framework and the *EvtBTheo* meta-theory, in the form of analyses that handle Event-B components. Three analyses are detailed: *deadlock-freeness*, *invariant weakness analysis* (tracking model holes) and *reachability*. The key points of using this framework are that: 1) WD conditions ensure elements are used correctly, 2) meta-properties on these analyses are established once and for all, and 3) these analyses can be performed without altering the machine's behaviour, in a non-intrusive way.

Note that only the definition of the *[PO]_Definition* operator is given, as *check_Machine_[PO]* is derived by replacing `[PO]` with the proposed PO name.

4.2 Deadlock-freeness

Requirements. Deadlock-freeness states that a machine m can always progress; i.e., there is always at least one enabled event in machine m, or more formally when the invariant holds then the disjunction of the guards holds.

PO Definition. The PO states that, for a machine m, there exists a progress event e such that any correct state $s \in Inv(m)$ verifies the guard of e ($s \in Grd(m)[\{e\}]$). When expressed using the meta-theory operators, it is formalised as $Inv(m) \subseteq Grd(m)[Progress(m)]$. This operator does not require any additional argument for *args*.

```
THEORY Theo4Deadlock IMPORT EvtBTheo
TYPE PARAMETERS STATE, EVENT
OPERATORS
  DeadlockFreeness_Definition <predicate> (m : Machine(STATE, EVENT))
    direct definition Inv(m) ⊆ Grd(m)[Progress(m)]
  ...
END
```

Listing 9: DeadlockFree Theory

Following the defined pattern, Listing 9 introduces a new theory `Theo4Deadlock` with two new operators together with the required WD condition.

```
CONTEXT ClockDeadlockFree
EXTENDS ClockMachineInstance
THEOREMS
  thmDeadlock: check_Machine_DeadLock(clock)
END
```

Listing 10: Clock DeadlockFreeness

Deadlock-freeness PO for Clock model. Listing 10 shows the context with *thmDeadlock* theorem generating WD and THM POs of the *clock* machine.

4.3 Invariant Weakness as a Non-intrusive Analysis

Requirements. A deployed system may present a number of vulnerabilities, that can be exploited by opponents (or make it weak to the environment) to modify its behaviour. These vulnerabilities usually come from *under-specification*, i.e., "holes" in the system's requirements or in its formal specification. To address this issue, a non-intrusive analysis of the model's specification is implemented, that does not alter its behaviour. It consists in investigating the robustness of the model's invariants with regard to *bad-events*, that model potential attacks (under-specification) against the system (model holes). *If the system's invariant is preserved by the bad-event*, it implies that *the invariant is* ***not strong enough*** to prevent the attack. For instance, the bad event of Listing 11 can be added to the *clock* machine without falsifying its original invariant. Similarly, other bad-events may be introduced: the event `tick_H5` guarded by $h < 19$ with action $m, h :| m' = 0 \wedge h' = h + 5$ and the event `tick_HM1` guarded

```
tick_M5
WHERE grd1 : m < 55
THEN act1 : m :| m' = m + 5
END
```

Listing 11: An Bad-event: progress by 5 min.

by $h < 23 \wedge m < 59$ with action $m, h :| m' = m + 1 \wedge h' = h + 1$. Note that a class of bad events could be added using two parameters $hn \neq 1$ and $mn \neq 1$ and a corresponding action of the form $m, h :| m' = m + mn \wedge h' = h + hn$.

Bad-events PO definition. This PO is formalised with the `AllowedMachineHoleSub_Definition` operator (Listing 12), with the bad-events as parameters.

```
THEORY EvtBTheorySubs  IMPORT THEORY EvtBTheory
TYPE PARAMETERS STATE, EVENT
OPERATORS
  AllowedMachineHoleSub_Definition <predicate> (m : Machine(STATE, EVENT),
      nGrd : ℙ(STATE), nBAP : ℙ(STATE × STATE))
    direct definition nBAP[Inv(m)∩nGrd]⊆Inv(m)
  ...
END
```

Listing 12: Weak specification analysis theory

Each bad-event is characterised by its guard $nGrd$ and its BAP $nBAP$. This operator defined as $nBAP[Inv(m) \cap nGrd] \subseteq Inv(m)$ states that the bad-event preserves the invariant. So, if the given PO is proved, the bad-event represents a successful attack, and the defined invariant is not strong enough.

Bad events PO for clock model. The analysis to Check the clock specification forbids minutes from progressing by 5 rather than 1, is handled by theorem *thmInspectInvEVTM5* of Listing 13, using the `AllowedMachineHoleSub_Definition` operator, where the bad-event is enabled when minutes are below 55 and thus progresses by 5. This corresponds to adding event `tick_M5` of Listing 11. Similar theorems are written for the `tick_H5` and `tick_HM1` bad-events.

```
CONTEXT ClockInspectInv EXTENDS ClockMachineInstance
THEOREMS
  thmInspectInvEVTM5: check_Machine_AllowedMachineHoleSub(clock,
                {m ↦ h | h ∈ ℤ ∧ m < 55},
                {(m ↦ h) ↦ (m' ↦ h') | m' = m + 5 ∧ h' = h ∧ h ∈ ℤ})
  thmInspectInvEVTH5: check_Machine_AllowedMachineHoleSub ···
  thmInspectInvEVTHM1: check_Machine_AllowedMachineHoleSub ···
END
```

Listing 13: Performing analysis on clock model

Note that the *thmInspectInvEVTM5*, *thmInspectInvEVTH5* and *thmInspectInvEVTHM1* theorems are proven for the *clock* model of the *ClockMachineInstance* corresponding to the Event-B machine of Listing 5. As a conclusion, the original model is insufficiently strong and does not provide sufficient constraints on the safe evolution of variables.

A strengthened machine. The designer strengthens the original machine, through instantiation, resulting in the new model shown in Listing 14. New state variables mb and hb are introduced to explicitly record the value of minutes and hours before a tick event occurs. In addition, the events are required to explicitly link these variables as $m = mb + 1$ and $h = hb + 1$.

```
CONTEXT ClockInvStrong
SETS Ev, ℤ × ℤ × ℤ × ℤ
CONSTANTS clock, tick_min, tick_hour, tick_midnight, init
AXIOMS
  axm1: clock ∈ Machine(ℤ × ℤ × ℤ × ℤ, Ev) ...
  axm2: ...
  axm3: State(clock) = ℤ × ℤ × ℤ × ℤ
```

```
   axm4−5: ...
   axm6: Inv(clock) = {m ↦ h ↦ mb ↦ hb | m ∈ ℕ ∧ h ∈ ℕ ∧ m < 60 ∧ h < 24∧
                                (m = mb + 1 ∧ hb = h) ∨ (m = 0 ∧ (h = hb + 1 ∨ h = 0))}
   axm7: AP(clock) = {m ↦ h ↦ mb ↦ hb | m = 0 ∧ h = 0 ∧ mb ∈ ℤ ∧ hb ∈ ℤ}
   axm8: BAP(clock) = {t ↦ ((m ↦ h ↦ mb ↦ hb) ↦ (m′ ↦ h′ ↦ mb′ ↦ hb′)) |
           (t = tick_min ∧ m′ = m + 1 ∧ h′ = h ∧ hb′ = h ∧ mb′ = m ∧ mb ∈ ℤ ∧ hb ∈ ℤ)∨
           (t = tick_hour ∧ m′ = 0 ∧ h′ = h + 1 ∧ hb′ = h ∧ mb′ = m ∧ mb ∈ ℤ ∧ hb ∈ ℤ)∨
           (t = tick_midnight ∧ m′ = 0 ∧ h′ = 0 ∧ hb′ = h ∧ mb′ = m ∧ mb ∈ ℤ ∧ hb ∈ ℤ)}
   ...
THEOREMS
   thm1: check_Machine_Consistency(clock)
   thmInspectInvEVTM5: ¬check_Machine_AllowedMachineHoleSub(clock,
         {m ↦ h ↦ mb ↦ hb | mb ∈ ℤ ∧ hb ∈ ℤ ∧ h ∈ ℤ ∧ m < 55},
         {(m ↦ h ↦ mb ↦ hb) ↦ (m′ ↦ h′ ↦ mb′ ↦ hb′) |
            m′ = m + 5 ∧ h′ = h ∧ hb′ = h ∧ mb′ = m ∧ mb ∈ ℤ ∧ hb ∈ ℤ})
   thmInspectInvEVTH5: ¬check_Machine_AllowedMachineHoleSub ...
   thmInspectInvEVTMH1: ¬check_Machine_AllowedMachineHoleSub ...
END
```

Listing 14: Clock resulting after the strengthening of the invariant

To guarantee that the identified bad-events are no longer triggerable, the predicates are negated in `thmInspectInvEVTM5`, `thmInspectInvEVTH5` and `thmInspectInvEVTHM1`. These theorems are proven to hold, demonstrating that the provided specification prohibits the presented inconsistent behaviour.

4.4 Reachability

Requirements. The reachability property is not natively available in Event-B. Such a property can be expressed using the EB4EB framework. Reachability property asserts that particular states can be attained under given constraints. The definition used below asserts that there exists a trace where a given state is reachable. This definition differs from the eventually operator of LTL. Note that a formalisation of the eventually operator of LTL is available in [21, 31].

```
THEORY Theo4Reachability IMPORT THEORY EvtBTheory
TYPE PARAMETERS STATE, EVENT
OPERATORS
// At least one "trgSet" event is triggerable after "src" event
   At_Least_One_Triggerable_Evt <predicate> (m : Machine(STATE, EVENT),
         src : ℙ(EVENT), trgSet : ℙ(EVENT)) ...
// All "SubSetEvt" events decrease the "variant"
   VariantDecrease <predicate> (m : Machine(STATE, EVENT), variant : ℙ(STATE × ℤ),
         SubSetEvt : ℙ(EVENT))...
// For all "SubSetEvt" events, the "variant" is a Natural number
   NaturalVariant <predicate> (m : Machine(STATE, EVENT), variant : ℙ(STATE × ℤ),
         SubSetEvt : ℙ(EVENT))...
// When "variant" is not null, there exists a "SubSetEvt" triggerable
      event
   One_Next_Evt_Is_Triggerable <predicate> (m : Machine(STATE, EVENT),
         variant : ℙ(STATE × ℤ), SubSetEvt : ℙ(EVENT))...
// "trg" event is reachable from "src" event through at least one "
      SubSetEvt" event
   Evt_Is_Reachable_From_Definition <predicate> (m : Machine(STATE, EVENT),
         src : EVENT, trg : EVENT, SubSetEvt : ℙ(EVENT), variant : ℙ(STATE × ℤ))
      well−definedness Machine_WellCons(m), trg ∈ Progress(m), src ∈ Event(m),
            Inv(m) ◁ variant ∈ Inv(m) → ℤ, Mch_INV(m), SubSetEvt ⊆ Progress(m)
      direct definition
         NaturalVariant(m, variant, SubSetEvt)∧ // Preserve the "variant" natural
         VariantDecrease(m, variant, SubSetEvt)∧ // "SubSetEvt" decrease the "
               variant"
```

```
        Next_Conv_Evt_Is_Triggerable(m, variant, SubSetEvt)∧ // the "variant" are
            always possible to decrease
        At_Least_One_Triggerable_Evt(m, src, SubSetEvt)∧ // "src" can trigger a "
            SubSetEvt"
        variant^−1[ℤ \ ℕ] ∩ Inv(m) ⊆ Grd(m)[{trg}] // "variant"=0 can trigger "trg"
    ...
END
```

Listing 15: Thoery of reachable property in Event-B

A *trace* σ of a machine m is a sequence of states $s_0, s_1, \ldots$ where s_0 is in the AP of the initialisation event and, for two consecutive state s_i, s_{i+1} in the trace, s_i must satisfy the guards of at least one event and (s_i, s_{i+1}) must satisfy the before-after predicate of this event. For $k \geq 0$, $\sigma(k)$ denotes the k-th state s_k of the trace. Then, s_j is reachable from s_i (denoted $s_i \mathcal{R} s_j$) if and only if $\exists \sigma, k, n \cdot n \geq 0 \wedge \sigma(n) = s_i \wedge k > 0 \wedge \sigma(n+k) = s_j$.

Reachability PO definition. The reachability property $s_i \mathcal{R} s_j$ is encoded using the Event-B meta-theory (Listing 15). The `Theo4Reachability` theory begins by defining the `At_Least_One_Triggerable_Evt` predicate, which states that, for any state reached after the *source* event, the guard of at least one *target* event is enabled. Then, the predicates `VariantDecrease` and `NaturalVariant` are defined. The former is satisfied only if, for machine m, each event of the $SubSetEvt$ set decreases the given *variant*; the latter ensures that the guards of the $SubSetEvt$ events imply that the variant is a natural number. The `One_Next_Evt_Is_Triggerable` predicate evaluates to true in machine m if the given *variant* is positive and at least one event in $SubSetEvt$ is activated.

These four operators formalise the induction-based definition of reachability. They are used to define the main predicate, `Evt_Is_Reachable_From_Definiton`, stating that, in machine m, target event *trg* can be triggered after a (finite) sequence of $SubSetEvt$ event triggers for the given *variant*, beginning with *src* event. Formally, triggering *src* activates at least one event in $SubSetEvt$ and each event of $SubSetEvt$ decreases the variant and enables at least one other event of $SubSetEvt$, and then *trg* is enabled when the variant reaches 0.

```
CONTEXT ClockReachability EXTENDS ClockMachineInstance
THEOREMS
    thmReach: check_Machine_Evt_Is_Reachable_From(clock, init, tick_midnight,
                {tick_min, tick_hour}, {m ↦ h ↦ v ∣ v = 24 ∗ 60 − 2 − (m + h ∗ 24)})
END
```

Listing 16: Clock machine with a reachable property checked

Clock machine reachability PO for clock model. In the clock model of Listing 6, it is worth checking that midnight is reachable from the initial event. This analysis is performed with theorem `thmReach` (see Listing 16), that checks whether the event *tick_midnight* is reachable from the event *init*, via events *tick_min* and *tick_hours*. The proposed variant is then $v = 24*60-2-(m+h*24)$. Proving the generated POs for this theorem establishes reachability.

4.5 Proof assessment

The defined operators of the proposed framework have been designed in the spirit of Event-B, i.e., 1) complex analyses are decomposed into simple ones

(case of reachability in Section 4.4) and 2) expressed in a single semantic setting: the one of Event-B (reflexive modelling) with set theory. This formalisation is influenced by two characteristics of the proof process, that 1) the Rodin prover is efficient when handling set expressions, and 2) theories may define customised *proved rewrite rules*, that may be summoned manually or automatically in the proof. Automatic rewriting rules that substitute operators by definitions are automatically generated. These rules are written to extract relevant information from machine objects, add them to the hypotheses, and produce multiple simpler goals. They are defined to be applied automatically and chained together, greatly improving proof automation. Indeed, these rewrite rules are included in Rodin's user-defined proof tactics, once and for all, increasing automation when proving the theorems formalising the newly defined POs.

Model	PO	Max Depth	Nodes	Interactive Nodes	Number of Tactic application
DeadlockFree clock	thmDeadlock (THM)	169	221	1	2
Reachability clock	thmReach (WD)	112	577	0	1
	thmReach (THM)	191	731	4	5
Inspect Inv clock	thmInspectInvEVTM5 (THM)	111	167	0	1
	thmInspectInvEVTH5 (THM)	112	169	0	1
	thmInspectInvEVTMH1 (THM)	113	171	0	1
Strong Inv clock	thmInspectInvEVTM5 (THM)	105	158	0	1
	thmInspectInvEVTH5 (THM)	118	171	0	1
	thmInspectInvEVTHM1 (THM)	128	181	0	1

Table 3: Proof statistic for the Clock model and its analyses

Table 3 presents the proof statistics for each analysis. The important number of nodes (representing atomic steps) in the proof trees is due to the extensive use of theory operators which the prover cannot handle directly, and thus their definitions must be unfolded. The introduction of the rewrite rules in a proof tactic perform automatically these unfold and reductions, making almost all steps fully automatic despite the introduction of the meta level (An entry of 0 in the interactive nodes column of Table 3). The rightmost column provides the number of tactic applications (iterations) during the proof. Indeed, a single tactic application may not be sufficient to fully discharge the proof goals.

5 Positioning this approach

5.1 Related work

Formalising model analyses has been addressed by several authors: Riccobene et al. [28] presented the ASM-Metamodel (AsmM) for Abstract State Machine (ASM) models considering core modelling constructs and semantics, expressed as an API manipulating ASM-related concepts like abstract machines, signatures, terms, rules, and so on. It is used to embed ASM in another formal method. This work resulted in a number of analyses, tools, and extensions for a variety of purposes [17]. A similar approach exists for VDM with MURAL, an interactive mathematical reasoning environment extended to support VDM [6] specifications based on meta-modelling concepts, and designed to offer a theorem prover for VDM models. Similarly, the Rodin tool offers an API for handling Event-B

models, intended to be used to develop plug-ins. This API is used by ProB [20] as well as by plug-ins handling model development [18] and code generation [16,22].

Ebner et al. [14] described the meta-programming framework used in Lean, which is an interactive theorem prover based on dependent type theory. This framework provides a means for reflecting object-oriented expressions into a meta-language by extending Lean's object language, based on Lean's modelling constructs. In [27], the authors present reflection in Agda in the style of Lisp, MetaML, and Template Haskell, as well as several typed programming applications. The MetaCoq [32] project proposed a certified meta-programming environment in Coq based on meta-modelling Coq concepts, including typing and operational semantics. This certified meta-modelling environment was also used in the development of the CertiCoq [4] certified compiler project. Similarly, this reflection principle [15] is implemented in Isabelle/HOL to build a HOL model within HOL to analyse and reason about various modelling concepts such as infinite hierarchy of large cardinals, polymorphism, verifying systems with self-replacement functionality, etc. In PVS, Miltra et al. [23] proposed *strategies* for proving abstraction relations between automata, based on theories and templates. This mechanism generalises proofs, making them highly reusable. With regard to Event-B, the formalisation of *contexts* (and only contexts) in the Event-B language has been proposed [7]. In related approaches, the B method has been embedded in PVS [24], to benefit from the modelling power of B, while accessing the proving power of the PVS theorem prover. However, this embedding is not formalised, and leads to the use of two separate methods.

Abstract interpretation showed its power to check system properties (absence of runtime errors, dead code, ...). Frameworks like [9, 10, 12] apply to programs through the definition of parameterised abstract domains corresponding to model analyses. The correctness of these analyses is expressed outside the framework.

The proposed approach is based on reflecting Event-B in itself i.e. its elements can be used as first-class objects in models. This is similar in Coq and HOL based approaches using dependent types, except that 1) it relies on set theory and FOL, easing transfer to other formalisms and 2) it is defined in the same setting as the state-transitions model of the system to be designed.

5.2 Advantages of the approach

This paper highlights several advantages of the EB4EB framework.

- Formal modelling and verification integrated in EB4EB. This framework enables the simultaneous use of two approaches for both modelling (operational with machines or axiomatic with contexts) and proving (meta-theory-based and model/induction-based) allowing users to use one or the other non-intrusively on pre-existing models. The proposed theories of the EB4EB framework can be easily extended following the methodology introduced in this paper, to handle new reusable models analyses by introducing, in Event-B, new *automatically generated POs* that preserve the semantics of Event-B.

- Easing proof process. The EB4EB reflexive framework enables the explicit manipulation of Event-B components by introducing meta-elements such as re-

quired datatypes, operators and theorems, extremely useful for expressing complex problems as well as proposing new reasoning mechanisms. However, due to the lack of advanced level proof engines such as SMTs, this resulted in enormous manual proof efforts. The introduced proved proof rules reduce interactive proof efforts while increasing proof automation.

- On-the-fly analysis. The EB4EB framework, which includes reasoning extensions, enables on-the-fly model analysis as well as advanced reasoning level for each Event-B model in the refinement chain. Note that the majority of Event-B models consist of several refinement layers, where each model of a given abstraction level can be analysed; i.e, the model is *lifted* as an instance of the EB4EB meta-level and is submitted for performing model analyses, at an advanced reasoning level, ensured by new POs generation.

- Correctness of the defined analyses. The EB4EB framework associates a *trace* to any Event-B machine (trace-based semantics). Such semantics is used to prove the correctness of the defined analyses. Indeed, a theorem stating that the property specifying a given model analysis holds on the traces of a machine is defined for this purpose. Such a correctness theorem has been proved for each of the analyses introduced in Section 4.

6 Conclusion

This paper presented a technique allowing a designer to define new POs for Event-B corresponding to model analyses that are not available in core Event-B. It is based on the extension of the reflexive EB4EB framework and its meta-theory $EvtBTheo$. The defined extended reasoning mechanisms and POs are not available in core Event-B. They have been defined as Event-B meta-modelling concepts allowing to express deadlock-freeness, bad-events and invariant strengthening, and reachability. It is demonstrated that non-intrusive analysis for Event-B models formalised in Event-B can be performed, at any abstraction level in the refinement chain, and without resorting to another formal method, which would require additional proofs to ensure the correct embedding of Event-B in that method. Moreover, the proof process has been enriched with relevant and proved rewrite rules, included in tactics, leading to a high level of proof automation. All the developments shown in this paper are completely formalised and all the proofs are realised[1].

Two future directions extending this work have been identified. The first one consists in defining domain-specific engineering theories in order to define specific domain-oriented properties as POs to be satisfied by system models. Such an approach opens towards standard conformance and certification. The second future direction exploits the fact that EB4EB defines an Event-B machine as an instance of a meta-theory as a set of axioms and theorems instances in FOL and set theory. This format can be exported into the higher order framework Dedukti [8, 13], and thus makes way for the design of correct import in, and export from Event-B of formal models through Dedukti.

[1] `https://www.irit.fr/~Peter.Riviere/models/`

References

1. Abrial, J.R.: Modeling in Event-B: System and software engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B. Tech. rep. (2009), `http://deploy-eprints.ecs.soton.ac.uk/216/`
3. Aït Ameur, Y., Dupont, G., Mendil, I., Méry, D., Pantel, M., Riviere, P., Singh, N.K.: Empowering the Event-B Method Using External Theories. In: IFM. Lecture Notes in Computer Science, vol. 13274, pp. 18–35. Springer (2022)
4. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL Workshop (2017)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer Publishing (2010)
6. Bicarregui, J.C., Ritchie, B.: Reasoning about VDM developments using the VDM support tool in Mural. In: Prehn, S., Toetenel, W.J. (eds.) VDM'91 Formal Software Development Methods. pp. 371–388. Springer Berlin Heidelberg (1991)
7. Bodeveix, J.P., Filali, M.: Event-B formalization of Event-B contexts. In: Rigorous State-Based Methods. pp. 66–80. Springer International Publishing (2021)
8. Boespflug, M., Carbonneaux, Q., Hermant, O., Saillard, R.: Dedukti: A Universal Proof Checker. In: Journées communes LTP - LAC. Orléans, France (2012)
9. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: Software Engineering and Formal Methods - 12th International Conference, SEFM 2014. LNCS, vol. 8702, pp. 271–277. Springer (2014)
10. Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. Ph.D. thesis, University of Rennes 1, France (2017)
11. Butler, M.J., Maamria, I.: Practical theory extension in Event-B. In: Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday. pp. 67–81 (2013)
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer (2005)
13. Dowek, G.: Deduction modulo theory. CoRR **abs/1501.06523** (2015), `http://arxiv.org/abs/1501.06523`
14. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP) (Aug 2017)
15. Fallenstein, B., Kumar, R.: Proof-producing reflection for HOL - With an application to model polymorphism. In: Interactive Theorem Proving - 6th International Conference, ITP 2015. LNCS, vol. 9236, pp. 170–186. Springer (2015)
16. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code Generation for Event-B. In: Integrated Formal Methods. pp. 323–338. Springer (2014)
17. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univers. Comput. Sci. **14**(12), 1949–1983 (2008)
18. Hoang, T.S., Dghaym, D., Snook, C.F., Butler, M.J.: A composition mechanism for refinement-based methods. In: 22nd International Conference on Engineering of Complex Computer Systems, ICECCS. pp. 100–109. IEEE (2017)

19. Lamport, L.: Specifying a simple clock. In: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers, chap. 2, pp. 15–22. Addison-Wesley (2002)
20. Leuschel, M., Butler, M.: ProB: A Model Checker for B, pp. 855–874. LNCS, Springer (2003)
21. Mendil, I., Riviere, P., Aït Ameur, Y., Singh, N.K., Méry, D., Palanque, P.A.: Non-intrusive annotation-based domain-specific analysis to certify event-b models behaviours. In: 29th Asia-Pacific Software Engineering Conference, APSEC. pp. 129–138. IEEE (2022)
22. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Symposium on Information and Communication Technology. pp. 179–188 (2011)
23. Mitra, S., Archer, M.: PVS strategies for proving abstraction properties of automata. Electronic Notes in Theoretical Computer Science **125**(2), 45–65 (2005), proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004)
24. Muñoz, C., Rushby, J.: Structural embeddings: Mechanization with method. In: International Symposium on Formal Methods. pp. 452–471. Springer (1999)
25. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag (2002)
26. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Automated Deduction - CADE-11, 11th International Conference on Automated Deduction. LNCS, vol. 607, pp. 748–752. Springer (1992)
27. Paul van der Walt: Reflection in Agda. Master's thesis (2012)
28. Riccobene, E., Scandurra, P.: Towards an interchange language for ASMs. In: Abstract State Machines (ASM). Advances in Theory and Practice, 11th International Workshop. LNCS, vol. 3052, pp. 111–126. Springer (2004)
29. Riviere, P., Singh, N.K., Aït Ameur, Y.: EB4EB: A Framework for Reflexive Event-B. In: International Conference on Engineering of Complex Computer Systems, ICECCS 2022. pp. 71–80. IEEE (2022)
30. Riviere, P., Singh, N.K., Aït Ameur, Y.: Reflexive Event-B: Semantics and Correctness the EB4EB Framework. IEEE Transactions on Reliability pp. 1–16 (2022)
31. Riviere, P., Singh, N.K., Aït Ameur, Y., Dupont, G.: Formalising liveness properties in Event-B. In: NASA Formal Methods 2023. LNCS (2023)
32. Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The MetaCoq Project. J. Autom. Reason. **64**(5), 947–999 (2020)