EB4EB: A Framework for Reflexive Event-B

Peter Riviere, Neeraj Kumar Singh, Yamine Aït-Ameur

EB4EB: A Framework for Reflexive Event-B

Peter Rivière, Neeraj Kumar Singh, Yamine Aït-Ameur
INPT-ENSEEIHT/IRIT
University of Toulouse, Toulouse, France
{peter.riviere, neeraj.singh, yamine.aitameur}@toulouse-inp.fr

***Abstract*—Event-B is a correct-by-construction rigorous state-based method offering features for formal modelling and proof automation. An inductive proof schema allows to prove system properties, in particular invariants.In the current setup, verifying other properties such as deadlock-freeness, reachability, event scheduling, liveness, etc., requires adhoc modelling.These properties can be established partially using model checkers or by using third party interactive provers.Other crucial aspects, such as deadlock-freeness, are difficult to express. The availabilty of a meta-modelling mechanism for explicit manipulation of Event-B concepts would allow to deal with higher order modelling concepts and to define generic properties and associated proof obligations.**

In this paper, we propose EB4EB, an Event-B based modelling framework allowing to manipulate Event-B features explicitly based on meta modelling concepts. This framework relies on a set of Event-B theories defining data-types, operators, well-defined conditions, theorems and proof rules. It preserves the core logical foundation, including semantics, of original Event-B models. Based on the instantiation of the introduced features at meta level, deep and shallow modelling approaches are proposed to exploit this framework. In addition, a case study is developed to demonstrate the use of our framework applying the deep and shallow embedding approaches. The whole framework is supported by the Rodin platform handling Event-B models and proofs.

***Index Terms*—Reflection, refinement and proof, meta-models, model instantiation, new proof obligations, theories, Event-B.**

I. Introduction

Motivation. Metamodelling is an engineering activity offering the capability to describe the core abstraction and properties to which models must adhere together with model analysis techniques. It has been widely adopted in the field of software engineering, particularly in model-driven engineering. Nowadays, formal methods have adopted such metamodelling techniques for developing theories axiomatising metamodels to represent higher level reasoning concepts used in the specification, development and verification of complex systems [6], [15], [21], [26].

Event-B [1] is a state-based formal method supporting the development of complex systems following a correct-by-construction approach. It is based on set theory and first-order logic, and it uses the Rodin [4] integrated development environment. Currently, the core modelling features of the Event-B language enable abstract system modelling as state-transitions systems, refinement based development, and interactive and automatic proofs. There are also a number of other RODIN plugins available to help with other modelling requirements, such as composition/decomposition [25], Theory plug-in [2], [9], code generation [16], [19] etc.

Among these plugins, the *Theory plug-in* [2], [9] offers powerful means to extend Event-B enabling for the development of additional data types, theories, and operators to extend the core modelling concepts and features of Event-B. For example, Dupont et al. [12], [13] have developed a set of theories to integrate continuous features in the Event-B modelling language for modelling differential equations.

Currently, Event-B framework only offers standard proof obligations (POs) that are generated automatically: invariant preservation, theorems proofs, variant decreasing, event feasibility, guard strenghtening, etc. For additional verifications, such as deadlock freeness, liveness, reachability, and domain specific properties, the designer relies on other tools based on interactive proof systems and model checkers. They require ad hoc modelling from the designer for each formalised model. There is a lack of access and explicit manipulation of Event-B concepts, thus it is impossible to express generic properties at a higher order level associated with extra reusable POs in a theory that permits *automatic* generation of such POs for any designed model.

Our claim. We claim that it is possible to express additional POs schemas using meta-modeling techniques without changing Event-B or Rodin, to automatically generate new POs for each Event-B model.

Our contribution. This paper proposes an Event-B-based modelling framework, EB4EB, that allows for the explicit manipulation of Event-B features using meta modelling concepts. To cover Event-B modelling language semantics, this framework relies on a set of Event-B theories that define data-types, operators, well-defined conditions, theorems, and proof rules. It allows for the manipulation of static and dynamic aspects of Event-B modelling features, to encode new proof obligations related to other types of properties once and for all. Deep and shallow modelling approaches are proposed to exploit this framework based on the instantiation of the introduced features at the meta level. In addition, a case study demonstrating the use of our framework using the deep and shallow embedding approaches is developed. The Rodin platform, which handles Event-B models and proofs, underpins the entire framework.

As far as we know, this is the only reflection framework in the Event-B language that allows explicit meta-level manipulations of Event-B concepts, including the support of higher level reasoning mechanisms by defining datatypes, operators, well-definedness, and new POs in theories.

Organisation of the paper. This paper is organised as follows. Section II presents related work and the core concept of Event-B language is described in Section III. Sections IV describes EB4EB framework. Section V illustrates the development of meta-theories for Event-B. In Section VI, we present an application of the Event-B meta-theories by applying deep and shallow embedding on the clock model. We demonstrate a new reasoning mechanism related to deadlock freeness in Section VII. In Section VIII, we provide an assessment and Section IX concludes the paper with future work.

II. Related Work

When two languages have the same or different semantics, one language can be mechanized into another by embedding the source logic of the first modelling language into the host modelling language. Deep and shallow embeddings are two widely used methods. Deep embedding describes explicitly the semantics and syntax of the source language in the host logic, whereas shallow embedding simply translates the semantics of the source language in the host logic [8] (i.e. the translator carries the semantics). Both approaches have their own pros and cons. Deep embedding requires more modelling effort to address structural and semantic elements of the source language. As a result, while this approach may be difficult to grasp and tedious, it offers full access, in the host logic, to the elements of the source modelling language for formal verification. On the other hand, the shallow embedding approach is straightforward and easy to use once the semantics of the source modelling language is encoded in the modelling language transformation. It leads to a limited access to the source modelling language constructs for formal verification, in particular when tracing verification results (e.g. counter-examples). Munoz et al. [21] proposed a structural embedding approach in which only the language structure is deep/shallow embedded in the host logic and the source language expression is replaced by the host logic expression.

There are few formal modelling languages that allow for abstract reasoning about models characteristics while also working on concrete models. In our case, the source and host modelling languages are the same (i.e. Event-B); it is a reflexive relationship. Such work has already been carried out in various formal techniques, such as in Coq [6] with the syntactic representation of Coq in Coq with Template Coq [5] and the semantics in MetaCoq [26]. Similarly, a reflection API has been developed in Agda [27], HOL [15], Idris [11], and Lean [14] to automate and/or simplify the definition of tactics. Some of them are motivated by other factors such as code generation and meta-programming.

Here, we discuss some approaches for Event-B. In [18] [20], the authors provided a comprehensive description of the logic's syntax with a type discipline and the intended semantics, including soundness, for an untyped first-order fragment of logic. In [24], Event-B logic is defined in order to describe modelling components like guards and invariants, as well as to express and discharge proof obligations. In this work, the abstract syntax of Event-B, including the concepts of theory and proof, is provided as a three-valued semantics in terms of a shallow embedding in Isabelle/HOL.

Note that the above mentioned work mainly address the semantics of Event-B modelling components, whereas our work targets the verification of important properties such as liveness, deadlock freeness, and event scheduling, among others, by developing meta-modelling concepts in order to manipulate Event-B concepts using deep and shallow embedding. In addition, it is developed in the Even-B language itself, using Event-B theories expressed using the Theory plugin.

III. Event-B

Event-B [1] method is based on set theory and first order logic (FOL). It relies on a powerful state-based modelling language where a set of events models state changes.

A. Event-B Contexts and Machines

`Contexts` (Tables I(a)) describe all the static elements of the models through the definition of *carrier sets* s, *constants* c, *axioms* A and *theorems* T_{ctx}.

Context	Machine	Refinement
CONTEXT Ctx	**MACHINE** M^A	**MACHINE** M^C
SETS s	**SEES** Ctx	**REFINES** M^A
CONSTANTS c	**VARIABLES** x^A	**VARIABLES** x^C
AXIOMS A	**INVARIANTS** $I^A(x^A)$	**INVARIANTS**
THEOREMS T_{ctx}	**THEOREMS** $T_{mch}(x^A)$	$J(x^A, x^C) \wedge I^C(x^C)$
END	**VARIANT** $V(x^A)$	**EVENTS**
	EVENTS	**EVENT** evt^C
	EVENT evt^A	**REFINES** evt^A
	ANY α^A	**ANY** α^C
	WHERE $G^A(x^A, \alpha^A)$	**WHERE** $G^C(x^C, \alpha^C)$
	THEN	**WITH**
	$x^A :\mid BAP^A($	$x^{A\prime}, \alpha^A : W(x^{A\prime}, \alpha^A,$
	$\alpha^A, x^A, x^{A\prime})$	$x^A, \alpha^C, x^C, x^{C\prime})$
	END	**THEN**
	END	$x^C :\mid BAP^C($
		$\alpha^C, x^C, x^{C\prime})$
		END
		END
(a)	(b)	(c)

TABLE I: Global structure of Context, Machines and Refinements

`Machines` (Table I(b)) describe model behaviour. It consists of *Variables* x, *Invariants* $I(x)$, *Theorems* $T_{mch}(x)$ and *Variants* $V(x)$. It defines a transition system represented as a set of guarded events evt recording state changes using a Before-After Predicates (BAP). Events which decrease the variant are tagged as *convergent* otherwise they are *ordinary*. *Invariants* $I(x)$ and *Theorems* $T_{mch}(x)$ ensure safety properties, while *Variant* $V(x)$ ensures convergence properties for *convergent* events.

(1)	Theorems (THM)	$A \Rightarrow T_{ctx} A \wedge I^A(x^A) \Rightarrow T_{mch}(x^A)$
(2)	Initialisation (INIT)	$A \wedge G_A(\alpha^A) \wedge BAP^A(\alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(3)	Invariant preservation (INV)	$A \wedge I_A(x^A) \wedge G_A(x^A, \alpha^A)$ $\wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(4)	Event feasibility (FIS)	$A \wedge I_A(x^A) \wedge G^A(x^A, \alpha^A)$ $\Rightarrow \exists x^{A\prime} \cdot BAP^A(x^A, \alpha^A, x^{A\prime})$
(5)	Variant progress (VAR)	$A \wedge I^A(x^A) \wedge G^A(x^A, \alpha^A)$ $\wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow \ V(x^{A\prime}) < V(x^A)$

TABLE II: Machine Proof obligations

(6)	Event Simulation (SIM)	$A \wedge I^A(x^A) \wedge J(x^A, x^C) \wedge G^C(x^C, \alpha^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge BAP^C(x^C, \alpha^C, x^{C\prime})$ $\Rightarrow BAP^A(x^A, \alpha^A, x^{A\prime})$
(7)	Guard Strengthening (GRDS)	$A \wedge I^A(x^A) \wedge J(x^A, x^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge G^C(x^C, \alpha^C) \Rightarrow G_A(x^A, \alpha^A)$

TABLE III: Refinement Proof obligations

- *Refinements.* Refinement (see Table I(c)) enables incremental design by introducing characteristics such as functionality, safety, reachability at different abstraction levels. It decomposes a *machine*, a state-transition system, into a more concrete model, by refining events and variables (simulation relationship). Introduction of gluing invariants preserves already proven properties.

- *Proof Obligations (PO) and Property Verification.* Several POs are associated with the Event-B models shown in Table II and III. These POs are generated automatically, and all of them must be successfully discharged to guarantee the correctness of an Event-B model, including refinements. Two additional POs related to refinement, guard strengthening and simulation, are required in our shallow modeling approach.

- *Core Well-definedness (WD).* The WD POs are associated to all built-in operators of the Event-B modelling language. Once proved, these WD conditions are used as hypotheses to prove other POs related to invariants, theorems, feasibility, etc.

B. Event-B extensions with Theories

In order to handle more complex modelling concepts not supported by native Event-B, an extension of Event-B based on the mathematical definitions has been proposed in [3], [10]. This extension, like Isabelle/HOL [22] or PVS [23], allows to define new *theories* by introducing new datatypes, operators, theorems and proof rules. They can be further used in the core development of Event-B models.

Theory
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 $WD(p_1, \ldots)$
direct definition D_1
AXIOMATIC DEFINITIONS
TYPES A_1, ...
OPERATORS
AOp2 <nature> (p_1: T_1, ...): T_r
well–definedness $WD(p_1, \ldots)$
AXIOMS A_1, ...
THEOREMS T_1, ...
END

TABLE IV: Global structure of Event-B Theories

- *Theory description.* Table IV shows core modelling elements for developing new theories. The core modelling elements are classified in different clauses known as datatypes, operators, axiomatic definitions, axioms, theorems and proof rules. A theory can be parameterized by Type in the clause `TYPE PARAMETERS`. The description of the data-type, operator, theorems and proof rules use the type parameters. Datatypes (`DATATYPES` clause) can be defined with *constructors*, and each constructor can have some *destructors*. Note that the destructors can also have inductive definition.

A theory may contain several *operators* of different nature (<nature> tag), expression or predicate. These new defined operators extend the capabilities of the Event-B core language and can be used directly in core modelling components like expression and predicate. Operators may be defined in two ways. First, explicitly in the `direct definition` clause where the operator is equivalent to an expression, and second, axiomatically in the `AXIOMATIC DEFINITIONS` clause where the behaviour of the operator is expressed by a set of axioms. Last, a theory defines a set of theorems proven with the help of defined operators and axioms.

Many theories have been defined for sequences, lists, groups, reals, differential equations, and so on [10], [12].

- *Well-definedness (WD) in Theories.* For each defined operator, a useful clause is *well-definedness* (WD) conditions. This clause ensures that the definition is correct. A WD proof obligation is generated when the operators are used in an Event-B expression. A correct definition is related to mathematical correctness, but it is also related to any other condition necessary if the operators guarantee some properties.

All the WD POs and theorems are proved using the Event-B proof system.

- *Event-B proof system and its IDE Rodin.* Rodin[1] is an open source Eclipse-based Integrated Development Environment 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 theories extension for Event-B 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 can be imported as hypotheses and used in proofs just like any other theorem. The proof system is partially automatic, the other parts are interactive. Many tools are available to help the proof like predicate provers or SMT solver.

IV. Our Framework

The primary objective of the developed framework is to offer the capability to *explicitly* manipulate Event-B features as first-order objects.

A. Methodology

For this purpose, we define a three steps methodology.

At the first step, we rely on the definition of Event-B theories describing states and events as data-types associated to a set of operators allowing to manipulate them. In addition, a couple of axioms defining these operators are introduced. These axiomatic definitions formalise the state-based semantics of Event-B models (contexts and machines). Finally a set of proven theorems are given. They formalise relevant properties on states, events and operators. The result of this step is a *Meta-theory* for Event-B defined once and for all.

In the second step, any specific Event-B model is obtained by instantiating the above mentioned theory to define states, events and related properties. Two possible instantiations are identified: *shallow* and *deep* embeddings.

[1]Rodin Integrated Development Environment http://www.event-b.org/index.html

Last, the third step entails proving the correctness of the instantiated models, i.e. proving that models are well-defined, invariants and theorems of the specific Event-B model hold, and other user expressed properties are true. In the meantime, the theorems of the theory are instantiated as well, they are useful for the proofs as they provide, for free, additional hypotheses.

B. Deep or Shallow modelling

As depicted in Figure 1, once the Event-B Meta-theory($\mathcal{D}.1$&$\mathcal{S}.1$) is described, Event-B models are described following two approaches.

1) Deep embedding (see Figure 1a): It consists in defining machine instances as an Event-B context ($\mathcal{D}.2$) where the generic type parameters of the meta-theory are instantiated by sets describing machines state variables and events. Similarly, all Event-B machine constructs such as event guards and before-after predicates, invariants, theorems and so on are formalised. They are defined using the operators introduced in the Meta-theory. This instantiation generates two kinds of POs. First, it generates the WD POs for each operator application to ensure that the Event-B machine is well-defined. Second, the POs related to the predicate operators that define machine consistency like invariant preservation, variant decreasing corresponding to WD POs are produced.

These obtained POs are proved using the Event-B Rodin theorem prover or other theorem provers.

2) Shallow embedding (see Figure 1b): In shallow embedding, an additional generic machine ($\mathcal{S}.2$) is defined at the meta-theory level. In the same line of TLA$^+$ [17] this machine formalises the behavioural semantics of Event-B. Using the meta-theory features, this machine defines *abstract* state variables and two abstract events `Init` and `Progress` recording respectively state variables initialisation and state transitions. Then, instantiation is defined using the *native Event-B refinement* where the first `Init` event is refined using the initialisation after-predicate and second for each event of the concrete machine (model), the `Progress` event is refined using the before-after predicates issued from the meta-theory. Note that the `Progress` is a collection of *ordinary* and *convergent* events used directly in the generic abstract model.

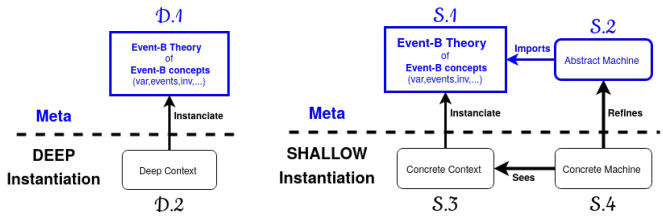
In this case, unlike from the deep modelling approach, proving the generated proof obligations relies on the Event-B inductive principle introduced by this generic machine.

Our framework provides meta-theory for deep modelling ($\mathcal{D}.1$) and shallow modelling ($\mathcal{S}.1$), and an abstract machine for shallow instantiation ($\mathcal{S}.2$). Deep instantiation ($\mathcal{D}.2$) and shallow instantiations ($\mathcal{S}.3$ and $\mathcal{S}.4$) need to be developed by users.

All of the developments presented in this paper can be found on `http://singh.perso.enseeiht.fr/Conference/ICECCS2022/EB4EB_Models.pdf`.

V. Meta-Theories for Event-B

In this section, we describe the development of meta-theories by defining data-types and operators ensuring well-defined conditions and proofs. This development allows to



(a) Deep Modelling (b) Shallow Modelling

Fig. 1: EB4EB framework

define different Event-B modelling components as first-order objects that can be accessed and manipulated using the defined data-types and operators. Note that the given definition for modelling components are derived from the Event-B book [1].

Listings 1, 2, 3 and 4 show meta-theory structure that can include several elements as follows:

A. Type Parameters and Datatypes

The Event-B meta-theory `EvtBTheo` introduces two polymorphic type parameters $STATE$ and $EVENT$ in the `TYPE PARAMETERS` clause (see Listing 1). These parameters are similar to carrier sets in contexts. The first attempt to represent the variables is with the type parameter $STATE$. An explicit description of each variable is not required at this abstract level. In fact, the type parameter $STATE$ abstracts the state as a cartesian product of all variables. At instantiation step, this abstract type is replaced with concrete variables of the considered Event-B model. The second type parameter $EVENT$ is used to abstract the label of events. These type parameters are used in the definition of a new datatype $Machine$ in the `DATATYPES` clause. A constructor `Cons_machine` is defined in the `CONSTRUCTOR` clause associated to destructors, i.e, $Event$, $State$, $Init$ and so on, to represent and to access different constituents of Event-B components. All these destructors define accesses for: $Event$ - machine events; $State$ - machine states; $Init$ - initialisation event; $Variant$ - machine variants; $Convergent$ - convergent events; $Ordinary$ - ordinary events; AP - after-predicates; BAP - before after-predicates; Grd - event guards; Inv - machine invariants; and Thm - machine theorems.

In our approach, we express the semantics of Event-B models and we use set comprehension to express different modelling components using $STATE$ and $EVENT$. For example, $Invariant$ is defined as a set of states that satisfy a set of properties. We also use quantification to express event parameters in the set comprehension's predicate.

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

$Grd : \mathbb{P}(EVENT \times STATE)$,
$Inv : \mathbb{P}(STATE)$
$Thm : \mathbb{P}(STATE)$)

Listing 1: Machine Data-type ($\mathcal{D}$.1&$\mathcal{S}$.1)

B. Theory operators

In the Event-B meta-theory, `EvtBTheo`, we introduce several operators to manipulate the modelling objects as well as checking the well-defined conditions.

1) Manipulation operators: Listing 2 shows a list of manipulation operators. We introduce `isInit`, `isProgress`, and `Progress` operators. `isInit` is declared as a prefix operator with two arguments represented by machine m and event e. In terms of existing expression language, the direct definition clause defines the predicate $e = Init(m)$. Similarly, another `isProgress` prefix operator is declared with two arguments, machine m and event e. Its direct definition clause, expressed as a predicate, states that the predicate holds if the event e is not the initialisation event and is a progress event of the machine m. The last operator `Progress` is declared as prefix operator with one argument machine m. Its direct definition clause is the expression $Event(m) \setminus Init(m)$ identifying a set of progress events. Note that our defined operators handle the *ordinary* and *convergence* events.

OPERATORS
isInit <*predicate*>
$(m : Machine(STATE, EVENT), e : EVENT)$
direct definition
$e = Init(m)$
isProgress <*predicate*>
$(m : Machine(STATE, EVENT), e : EVENT)$
direct definition
$\neg isInit(m, e) \land e \in Event(m)$
Progress <*expression*> $(m : Machine(STATE, EVENT))$
direct definition
$Event(m) \setminus \{Init(m)\}$

Listing 2: Destructor operator of Data-type ($\mathcal{D}$.1&$\mathcal{S}$.1)

2) Well-defined operators: The `DATATYPES` clause defines a constructor and destructors to access the Event-B modelling components. These destructors contain typing information only and may lead to be an ill-defined datatype definition. For example, the BAP destructor is a relation between events and states, but the initialisation event is not one of the events of this before-after relation. In order to avoid such ill-defined typing definitions, we introduce a set of new operators in Listing 3 equipped with WD conditions. Due to space constraints, we only show an extract of the set of well-defined operators.

In Listing 3, the first well-defined operator `BAP_WellDefined` is declared with one argument machine m, and its direct definition shows that all events in the domain of the BAP relation are progress events, implying that the event set contains no initialisation event. The next well-defined operator `Grd_WellDefined` is also defined with single machine m argument. Its direct definition states that all events in the domain of the Grd relation are progress events. To check the well-defined condition of `Init` operator, the `Init_WellDefined` operator is declared. Its direct definition states that the initialisation event belongs to machine events and is an ordinary event.

BAP_WellDefined <*predicate*> $(m : Machine(STATE, EVENT))$
direct definition
$dom(BAP(m)) = Progress(m)$
Grd_WellDefined <*predicate*> $(m : Machine(STATE, EVENT))$
direct definition
$dom(Grd(m)) = Progress(m)$
Init_WellDefined <*predicate*> $(m : Machine(STATE, EVENT))$
direct definition
$Init(m) \in Event(m) \land Init(m) \in Ordinary(m)$
Variant_WellDefined <*predicate*>
$(m : Machine(STATE, EVENT))$
direct definition
$Inv(m) \lhd Variant(m) \in Inv(m) \to \mathbb{Z}$
Tag_Event_WellDefined <*predicate*>
$(m : Machine(EVENT, STATE))$
direct definition
$Convergent(m) \cup Ordinary(m) = Event(m) \land$
$Convergent(m) \cap Ordinary(m) = \emptyset$
Machine_WellDefined <*predicate*>
$(m : Machine(STATE, EVENT))$
direct definition
$BAP\_WellDefined(m) \land$
$Grd\_WellDefined(m) \land$
$Init\_WellDefined(m) \land$
$Tag\_Variant\_WellDefined(m) \land$
$Variant\_WellDefined(m)$

Listing 3: Operator of well defined Data-type ($\mathcal{D}$.1&$\mathcal{S}$.1)

The direct definition of the next `Variant_WellDefined` operator shows that all the states belonging to the variant states are convergent and identified from the set of invariant states, i.e. each variant state element is associated with an integer. Note that the variant is a total function in the invariant states. The direct definition of the `Tag_Event_WellDefined` operator shows that the union of convergent and ordinary events are equal to the progress events, and are exclusive sets of events. The last `Machine_WellDefined` operator is important; the direct definition shows the conjunction of all other well-defined operators. It represents the global well-defined condition associated with an Event-B machine m.

3) Proof obligation operators: Once the notion of machine is defined, the last set of operators deal with the explicit definition of Event-B POs (see Listing 4). They help to discharge the generated proof obligations as given in Section II, such as `INV`, `FIS`, `NAT` and `VAR`. Their definitions are inductive as they apply on the initialisation and then on all other events.

All the defined operators have a machine m as argument. The first declared operator `Mch_THM` shows that the invariants are a subset of theorems. The next two operators, `Mch_FIS_Init` and `Mch_FIS`, represent the base case and induction case for the feasibility POs, respectively. The direct definition for the induction case ensures that the invariants and guards of the progress event e are a subset of the domain of the BAP of e. Similarly, the next three *predicate* operators, `Mch_INV_Init`, `Mch_INV` and `All_Progress_INV`, define the initialisation, the induction case of the invariant PO for a single event e, and the induction case of invariant properties for all progress events, respectively. The direct definition of `Mch_INV` ensures that BAP of progress event e preserves the invariants if guards and invariants are true before.

The next two operators `Mch_VARIANT` and `All_Convergent_VAR` are declared to represent convergent properties. The `Mch_VARIANT` definition

guarantees that if invariants and guards hold, then the BAP decreases the variant associated with the convergent event e. The WD clause defines other well-defined operators to ensure the correctness as well as the required WD conditions for the variants. Similarly, the operator `All_Convergent_VAR` generalises the definition of convergence, it checks the required properties for all convergent events of machine m. Then, the next two operators, `Mch_NAT` and `All_Convergent_NAT`, define a variant for an event e as a natural number and all convergent events have a natural number as variant, respectively.

The predicate operator, `Machine_PO` is the most important. It allows to generate, thanks to the WD PO mechanism, all possible POs related to a machine. Its direct definition is **the conjunction of all the other PO operators related to Event-B constituents previously defined**. Its well-definedness clause uses the `Machine_WellDefined` operator. This operator is associated to the last defined operator `check_Machine_WellDefined` ensuring that all the destructors are well-defined and the machine satisfies POs. Observe that there is no direct definition for this operator as it shall always be true $\top$. The well-definedness clause contains the required condition that allows to check the machine well-definedness and machine POs together. When this operator is used, it generates a PO for this WD condition.

Mch_THM <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $Inv(m) \subseteq Thm(m)$
Mch_FIS_Init <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $Inv(m) \cap AP(m) \neq \emptyset$
Mch_FIS <*predicate*>
    $(m : Machine(STATE, EVENT), e : Event)$
  well-definedness
    $e \in Progress(m)$
  direct definition
    $Inv(m) \cap Grd(m)[\{e\}] \subseteq dom(BAP(m)[\{e\}])$
All_Progress_Mch_FIS <*predicate*>
    $(m : Machine(STATE, EVENT))$
  direct definition
    $\forall e \cdot e \in Progress(m) \Rightarrow Mch\_FIS(m, e)$
Mch_INV_Init <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $AP(m) \subseteq Inv(m)$
Mch_INV <*predicate*>
    $(m : Machine(STATE, EVENT), e : EVENT)$
  well-definedness $e \in Progress(m)$
  direct definition
    $BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$
All_Progress_INV <*predicate*> $(m : Machine(STATE, EVENT))$
  direct definition
    $\forall e \cdot e \in Progress(m) \Rightarrow Mch\_INV(m, e)$
Mch_VARIANT <*predicate*>
    $(m : Machine(STATE, EVENT), e : EVENT, s : STATE)$
  well-definedness Variant_WellDefined(m),
      All_Progress_INV(m), BAP_WellDefined(m),
      Tag_Variant_WellDefined(m), $e \in Convergent(m) \wedge$
      $s \in State(m) \wedge s \in Inv(m) \wedge s \in Grd(m)[\{e\}]$
  direct definition
    $\forall sp \cdot sp \in BAP(m)[\{e\}][\{s\}]$
    $\Rightarrow (Inv(m) \lhd Variant(m))(s) > (Inv(m) \lhd Variant(m))(sp)$
All_Convergent_VAR <*predicate*>
    $(m : Machine(STATE, EVENT))$
  well-definedness Variant_WellDefined(m),
      All_Progress_INV(m), BAP_WellDefined(m),
      Tag_Variant_WellDefined(m)
  direct definition
    $\forall e, s \cdot e \in Event(m) \wedge e \in Convergent(m) \wedge$
    $s \in State(m) \wedge s \in Inv(m) \wedge s \in Grd(m)[\{e\}]$
      $\Rightarrow Mch\_VARIANT(m, e, s)$

Mch_NAT <*predicate*>
    $(m : Machine(STATE, EVENT), e : EVENT)$
  well-definedness $e \in Convergent(m)$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[\{e\}]] \subseteq \mathbb{N}$
All_Convergent_NAT <*predicate*>
    $(m : Machine(STATE, EVENT))$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[Convergent(m)]] \subseteq \mathbb{N}$
Machine_PO <*predicate*> $(m : Machine(STATE, EVENT))$
  well-definedness Machine_WellDefined(m)
  direct definition
    $Mch\_THM(m) \wedge Mch\_FIS\_Init(m) \wedge Mch\_INV\_Init(m) \wedge$
    $All\_Progress\_Mch\_FIS(m) \wedge All\_Progress\_Mch\_INV(m) \wedge$
    $All\_Convergent\_Mch\_VARIANT(m) \wedge$
    $All\_Convergent\_Mch\_NAT(m)$
check_Machine_WellDefined <*predicate*>
    $(m : Machine(EVENT, STATE))$
  well-definedness
    $Machine\_WellDefined(m) \wedge Machine\_PO(m)$
  direct definition
    $\top$

Listing 4: Operator of well defined Data-type ($\mathcal{D}.1\&\mathcal{S}.1$)

VI. Application of Event-B Meta-Theories

In this section, we describe a simple case study, the clock, to demonstrate how our developed meta models can be applied.

A. Case Study: The Clock

The main functionalities (FUN) and requirements (REQ) of the clock case study are given as follows:

- **FUN1** A minute can progress
- **FUN2** An hour can progress
- **REQ1** The hours are represented in a 24-hour format.
- **REQ2** The clock must converge on midnight.
- **REQ3** The clock never stops.

In Listing 5, we describe the clock model that is formalised in native Event-B language. In this model, two variables are defined, minute m and hour h, in $inv1 - inv2$. Two safety properties are introduced in $inv3 - inv4$. The first safety property (REQ1) states that the minute m is always less than 60 and hour h is less then 24. The next safety property (REQ3) is defined as a theorem that is a disjunction of all guards to state that the clock never stops means always the guard of at least one event is true. The last safety property (REQ2) is related to convergence (variant) expressed by the number $24 * 60 - 1 - (m + h * 60)$. In this model, we introduce three events: *tick_min* - to model the minute progress by 1; *tick_hour* - to model the hour progress by 1; and *tick_midnight* - to reset the clock at midnight. The required guards are added in the defined events to update the minute m and hour h.

MACHINE Clock
VARIABLES m, h
INVARIANTS
  inv1-2 : $m \in \mathbb{N} \wedge h \in \mathbb{N}$
  inv3-4 : $m < 60 \wedge h < 24$
THEOREMS
  thm1 : $m < 59 \vee (m = 59 \wedge h < 23) \vee (m = 59 \wedge h = 23)$
VARIANT $24 * 60 - 1 - (m + h * 60)$
EVENTS
  INITIALISATION
  THEN
    act1 : $m, h :| m' = 0 \wedge h' = 0$
  END
  tick_min <*convergent*>
  WHERE
    grd1 : $m < 59$

```
  THEN
    act1 : m :| m' = m + 1
  END
  tick_hour <convergent>
  WHERE
    grd1 : m = 59 ∧ h < 23
  THEN
    act1 : m, h :| m' = 0 ∧ h' = h + 1
  END
  tick_midnight <ordinary>
  WHERE
    grd1 : m = 59 ∧ h = 23
  THEN
    act1 : m, h :| m' = 0 ∧ h' = 0
  END
END
```

Listing 5: A machine of clock

Once the theory for Event-B concepts is designed, two main approaches to instantiate it are envisioned, namely deep modelling and shallow modelling as described below.

B. Deep embedding of the clock model

We describe the development of the clock case study using the deep modelling instantiation technique of Section IV-B using the meta-theory introduced in Section V. All constituents of the Clock model are explicitly expressed in terms of the `EvtBTheo` Meta-theory constructs. The Clock Event-B model is represented as an Event-B context, and POs are described either as theorems or as well-definedness POs.

The deep modelling resulting context of the Event-B clock model given in Listing 5 is presented in Listing 6. In this context, a set Ev lists all the clock events in $axm1$. The clock machine $clock$ is defined by axiom $axm2$ as a member of $Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$, where the first argument defines machine state as $\mathbb{Z} \times \mathbb{Z}$ and the second one machine events Ev. Furthermore, three axioms ($axm3 - axm5$) are used to instantiate $Event$ with the enumerated set Ev, $Init$ with the event label $init$, and $State$ with $\mathbb{Z} \times \mathbb{Z}$.

```
CONTEXT ClockDeepInstance
SETS Ev
CONSTANTS clock , tick_min , tick_hour , tick_midnight , init
AXIOMS
  axm1 : partition(Ev,
         {init}, {tick_midnight}, {tick_hour}, {tick_min})
  axm2–3 : clock ∈ Machine(ℤ × ℤ, Ev) ∧ Event(clock) = Ev
  axm4–5 : Init(clock) = init ∧ State(clock) = ℤ × ℤ
  axm6 : Thm(clock) = {m ↦ h |
         m < 59 ∨ (m = 59 ∧ h < 23) ∨ (m = 59 ∧ h = 23)}
  axm7 : Inv(clock) = {m ↦ h | m ∈ ℕ ∧ h ∈ ℕ ∧ m < 60 ∧ h < 24}
  axm8 : AP(clock) = {m ↦ h | m = 0 ∧ h = 0}
  axm9 : BAP(clock) = {t ↦ ((m ↦ h) ↦ (mp ↦ hp)) |
                       (t = tick_min ∧ mp = m + 1 ∧ hp = h)∨
                       (t = tick_hour ∧ mp = 0 ∧ hp = h + 1)∨
                       (t = tick_midnight ∧ mp = 0 ∧ hp = 0)}
  axm10 : Grd(clock) = {t ↦ (m ↦ h) |
                        (t = tick_min ∧ m < 59)∨
                        (t = tick_hour ∧ m = 59 ∧ h < 23)∨
                        (t = tick_midnight ∧ m = 59 ∧ h = 23)}
  axm11 : Convergent(clock) = {tick_min, tick_hour}
  axm12 : Ordinary(clock) = {tick_midnight, init}
  axm13 : Variant(clock) = {m ↦ h ↦ v |
                            v = 24 ∗ 60 − 1 − (m + h ∗ 60)}
THEOREMS
  thm1 : check_Machine_WellDefined(clock)
END
```

Listing 6: A deep instance of the clock machine ($\mathcal{D}.2$)

The next two axioms ($axm6$ and $axm7$) are defined to instantiate theorem Thm and invariant Inv using comprehensive sets derived from $thm1$ and $inv1 - inv4$ of Listing 5. Axiom $axm8$ instantiates the after-predicate AP derived from the action of the initialisation event ($act1$) in the Clock machine. Similarly, axioms $axm9$ and $axm10$ are used to instantiate the before-after predicate BAP and the guard Grd with a set of actions and guards of all events derived from the Clock machine using comprehensive sets. The next axioms $axm11 - axm12$ instantiate the $Convergent$ and $Ordinary$ with a list of convergent and ordinary events, respectively. In this model, we have only two convergent events $tick\_min$ and $tick\_hour$ and two ordinary events $init$ and $tick\_midnight$. Axiom $axm13$ is used to instantiate the $Variant$ with the defined variant of the Clock model.

Machine correctness. It is important to note the introduction of a *theorem* $thm1$ to invoke the $check\_Machine\_WellDefined$ operator, which is used to check the well-definedness and POs generation of the Clock machine. This theorem shall be proved, it entails machine correctness.

C. Shallow embedding of the clock model

We describe the development of the clock case study using the shallow modelling instantiation technique of Section IV-B using the meta-theory introduced in Section V.

All constituents of the Clock model are explicitly expressed in terms of the `EvtBTheo` Meta-Theory constructs. The Clock Event-B model is represented as an Event-B context and machine, and POs are described either as theorems or as guard strengthening POs. It is inspired from shallow embeddings [8] used in other interactive provers like Isabelle/HOL and PVS. In the same vein as the shallow embedding, we use the Event-B to preserve semantics and the supporting syntax. Thus, we describe an abstract Event-B model formalising the required properties for Event-B models correctness: a `context` for the static part and properties and a generic `machine` for the dynamic parts i.e. transitions represented by events.

1) Abstract generic model: Listings 7 and 8 show the context and machine of the Clock generic model. The context contains sets Ev and S for events and states. A constant $machine$ is introduced as a member of $Machine(S, Ev)$.

```
CONTEXT ClockShallowGen
SETS Ev , S
CONSTANTS machine
AXIOMS
  axm1 : machine ∈ Machine(S, Ev)
END
```

Listing 7: A static element of abstract machine ($\mathcal{S}.2$)

In the generic machine model, we declare two variables s and $InitDone$ in invariants $inv1 - inv2$. These variables are set in the INITIALISATION event. To ensure that the invariant is satisfied, we introduce a new invariant $inv3$. In this model, we define three events `Do_Init`, `Do_Ordinary`, and `Do_Convergent` whose actions modify the state using the AP and BAP operators ($act1$). The first event is used to initialise state variables in actions ($act1 - act2$). Its guards ensure that $InitDone$ is $FALSE$ ($grd1$), and the feasibility and invariants hold for the $Init$ event. The `Do_Ordinary` event updates the machine state s for an event e annotated

as $Ordinary$. Its guards state that $InitDone$ is TRUE; the machine state s belongs to Grd of e ($grd2$); the event e is a progress and ordinary event ($grd3 - grd4$); and feasibility and invariant properties of $machine$ hold for the event e ($grd5 - grd6$). Similar to the ordinary event, the last event `Do_Convergent` contains additional guards $grd3$ to tag the event e as convergent and $grd7 - grd8$ to ensure that the variant properties of $machine$ for the event e hold.

Note that our generic abstract model contains *initialisation*, *ordinary* and *convergent* events, whereas we may only have *initialisation* and *progress* events, in the same of spirit of TLA$^+$, where the *progress* event can be refined by *ordinary* and *convergent* events later in further refinement.

MACHINE ClockShallowGenMac
SEES ClockShallowGen
VARIABLES s, $InitDone$
INVARIANTS
inv1–2: $s \in S \land InitDone \in BOOL$
inv3: $InitDone = TRUE \Rightarrow s \in Inv(machine)$
EVENTS
INITIALISATION
THEN
act1: $s :| s' \in S$
act2: $InitDone := FALSE$
END
Do_Init
WHERE
grd1: $InitDone = FALSE$
grd2–3: $Mch\_FIS\_Init(machine) \land Mch\_INV\_Init(machine)$
THEN
act1: $s :| s' \in AP(machine)$
act2: $InitDone := TRUE$
END
Do_Ordinary
ANY e
WHERE
grd1–2: $InitDone = TRUE \land s \in Grd(machine)[\{e\}]$
grd3–4: $e \in Progress(machine) \land e \in Ordinary(machine)$
grd5–6: $Mch\_FIS(machine, e) \land Mch\_INV(machine, e)$
THEN
act1: $s :| s' \in BAP(machine)[\{e\}][\{s\}]$
END
Do_Convergent
ANY e
WHERE
grd1–2: $InitDone = TRUE \land s \in Grd(machine)[\{e\}]$
grd3–4: $e \in Progress(machine) \land e \in Convergent(machine)$
grd5–6: $Mch\_FIS(machine, e) \land Mch\_INV(machine, e)$
grd7: $Mch\_VARIANT(machine, e, s)$
grd8: $Mch\_NAT(machine, e)$
THEN
act1: $s :| s' \in BAP(machine)[\{e\}][\{s\}]$
END
END

Listing 8: A generic abstract machine ($\mathcal{S}.2$)

2) Concrete model: The concrete model refines the abstract generic model introduced above. The static elements of the clock model are described by the context of Listing 9 and dynamic elements are described in machine of Listing 10.

Static constituents. In the context of Listing 9, we define a constant pr in $axm1$ as a bijection relation between $(\mathbb{Z} \times \mathbb{Z})$ and S to maintain an exact correspondence between abstract and concrete states. We enumerate the set Ev with clock events in $axm2$. Axioms ($axm3$-$axm5$) are used to instantiate $Event$ with enumerated set Ev, $Init$ with the event $init$, and $State$ with S. Axiom $axm6$ is defined to instantiate invariant Inv using comprehensive sets derived from $Inv1 - inv4$ of Listing 5. Variant of the clock machine is introduced in $axm7$. Then two axioms ($axm8 - axm9$) are used to instantiate $Convergent$ and $Ordinary$ with a set of convergent and ordinary events. The last axiom $axm10$ instantiates theorem concrete machine Thm derived from Listing 5.

Context correctness. Four theorems $thm1 - thm4$ are introduced to check the POs associated with theorems, and well-definedness for variant and machine events (initial and tagged events). Once proved, these theorems guarantee that the context is well-defined and the required properties hold.

CONTEXT ClockShallow **EXTENDS** ClockShallowGen
CONSTANTS tick_min, tick_hour, tick_midnight, init, pr
AXIOMS
axm1–2: $pr \in (\mathbb{Z} \times \mathbb{Z}) \rightarrowtail S \land State(machine) = S$
axm3–4: $Event(machine) = Ev \land Init(machine) = init$
axm5: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm6: $Inv(machine) = pr[\{m \mapsto h \mid \boldsymbol{m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24}\}]$
axm7: $Variant(machine) = \{s, v, m, h \cdot s = pr(m \mapsto h) \land \boldsymbol{v = (24 * 60 - 1 - (m + h * 60))} \mid s \mapsto v\}$
axm8: $Convergent(machine) = \{tick\_min, tick\_hour\}$
axm9: $Ordinary(machine) = \{tick\_midnight, init\}$
axm10: $Thm(machine) = pr[\{m \mapsto h \mid \boldsymbol{m < 59 \lor (m = 59 \land h < 23) \lor (m = 59 \land h = 23)}\}]$
THEOREMS
thm1–2: $PO\_THM(machine) \land Init\_WellDefined(machine)$
thm3: $Variant\_WellDefined(machine)$
thm4: $Tag\_Event\_WellDefined(machine)$
END

Listing 9: Instances for static elements: clock machine ($\mathcal{S}.3$)

Dynamic constituents. The abstract machine is refined in Listing 10 to introduce the events of the $Clock$ Event-B machine. In this model, we declare two new variables m and h and a gluing invariant $inv1$ to link (glue) concrete and abstract variables. No new event is added but each abstract event has been refined by concrete ones by providing concrete guards and actions. The newly introduced variables are set in the refined INITIALISATION event, and a witness is provided to map the abstract and concrete variables. In the `Do_Init` event, we introduce a new guard ($grd2$) to instantiate AP operator and a witness is provided for the state s'. The action of this event modifies the concrete clock variables m and h.

The event `Do_Convergent` is refined by two concrete clock events `Tick_min` and `Tick_hour`, and the event `Do_Ordinary` is refined by the concrete event `Tick_midnight`. In `Tick_min` event, $grd1$ is similar to the abstract event, $grd2$ is updated according to the concrete variables m and h, and two new guards ($grd3 - grd4$) are added to instantiate Grd and BAP operators for $tick\_min$, respectively. In this event, two witnesses are provided for the abstract parameter event e and the state s'. The action of this event uses BAP operator to update concrete variables m and h. Note that the other refined events are similarly modeled by providing witnesses and appropriate instantiations.

Machine correctness. Gluing invariant ($inv1$), witnesses and guard strengthening are introduced to check the POs associated with machine events (initial and tagged events). New generated POs are proved to guarantee that the machine is correct and the required properties hold. The complete development of meta theory, instantiations of deep and shallow clock models, and the Event-B clock model are available at `http://singh.perso.enseeiht.fr/Conference/ICECCS2022/EB4EB_Models.pdf`.

```
MACHINE ClockShallowMac
REFINES ClockShallowGenMac
SEES ClockShallow
VARIABLES m, h, InitDone
INVARIANTS
  inv1: s = pr(m ↦ h)
EVENTS
  INITIALISATION
  WITH
    s': s' = pr(m' ↦ h')
  THEN
    act1: m, h :| m' ∈ ℤ ∧ h' ∈ ℤ
    act2: InitDone := FALSE
  END
  Do_Init REFINES Do_Init
  WHERE
    grd1: InitDone = FALSE
    grd2: pr[{ms ↦ hs | ms = 0 ∧ hs = 0}] = AP(machine)
  WITH
    s': s' = pr(m' ↦ h')
  THEN
    act1: m, h :| pr(m' ↦ h') ∈ AP(machine)
    act2: InitDone := TRUE
  END
  Tick_min REFINES Do_Convergent
  WHERE
    grd1: InitDone = TRUE
    grd2: m ↦ h ∈ pr ~ [Grd(machine)[{tick_min}]]
    grd3: pr[{ms, hs · ms < 59 ∧ hs ∈ ℤ | ms ↦ hs}] =
          Grd(machine)[{tick_min}]
    grd4: {ss, ssp, ms, hs, msp, hsp·
           ss = pr(ms ↦ hs) ∧ ssp = pr(msp ↦ hsp)∧
           msp = ms + 1 ∧ hsp = hs | ss ↦ ssp} =
          BAP(machine)[{tick_min}]
  WITH
    e: e = tick_min
    s': s' = pr(m' ↦ h')
  THEN
    act1: m, h :| pr(m' ↦ h') ∈
          BAP(machine)[{tick_min}][{pr(m ↦ h)}]
  END
  Tick_hour REFINES Do_Convergent
  ...
  THEN
    act1: m, h :| pr(m' ↦ h') ∈
          BAP(machine)[{tick_hour}][{pr(m ↦ h)}]
  END
  Tick_midnight REFINES Do_Ordinary
  ...
  THEN
    act1: m, h :| pr(m' ↦ h') ∈
          BAP(machine)[{tick_midnight}][{pr(m ↦ h)}]
  END
END
```

Listing 10: A shallow instance of the clock machine ($\mathcal{S}.4$)

VII. New Reasoning: Deadlock Freeness

Currently, modelling deadlock freeness (**FUN1**) is achieved manually by the model designer using an additional theorem stating that the disjunction of all events guards holds. Proceeding this way is error prone if the user does not write it correctly. *Generating this theorem as a new PO would avoid the designer* of the burden of writing this theorem.

Expressing such a PO (and others) at the Meta-theory level is possible provided that a specific operator with its WD condition is introduced. Listing 11 shows the formalisation of the `PO_DeadlockFreeness` operator in a theory extension `EvtBTheoDeadlock`. Then, checking the machine operator `check_Machine_DeadLock` is redefined so as to handle deadlock freeness in addition to the other WD conditions provided by `check_Machine_WellDefined` issued from the extended `EvtBTheo`. To generate the deadlock freeness PO for a concrete model, it is enough to state that predicate operator `check_Machine DeadLock` is a theorem in the context of the concrete model.

```
THEORY EvtBTheoDeadlock IMPORT EvtBTheo
  TYPE PARAMETERS STATE, EVENT
  OPERATORS
    // A predicate for machine Deadlock freeness
    PO_DeadlockFreeness <predicate>
          (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ⊆ Grd(m)[Progress(m)]
    // Well-defined machines with deadlock freeness.
    check_Machine_DeadLock <predicate>
          (m : Machine(STATE, EVENT))
      well-definedness check_Machine_WellDefined(m)∧
            PO_DeadlockFreeness(m)
      direct definition ⊤
END
```

Listing 11: A theory about Deadlock Freeness

VIII. Assessment

We evaluate our EB4EB framework in terms of proof effort and benefits of such extension. Rodin supports all the theories and case study. All the resulting POs were successfully discharged. The complete developments can be found on `http://singh.perso.enseeiht.fr/Conference/ICECCS2022/EB4EB_Models.pdf`.

1) Proof effort: Proof statistics for all the developed models (i.e clock, deep, shallow generic, shallow context, and shallow machine) are presented in Table V. Due to its simplicity and

Models	Total POs	Automatic POs	Interactive POs
Clock Event-B model (Listing 5)	25	25 (100%)	0 (0%)
Clock Deep model context (Listing 6)	2	0 (0%)	2 (100%)
Shallow generic machine (Listing 8)	14	8 (57%)	6 (43%)
Clock Shallow context (Listing 9)	5	1 (20%)	4 (80%)
Clock Shallow machine (Listing 10)	54	26 (48%)	28 (52%)

TABLE V: Proof Statistics

as expected, we observe that the clock model is automatically proved using Rodin provers. The proofs of correctness of the same case study using both deep and shallow modelling resulted in a greater number of proof obligations, some of them being proved interactively. Although the number of POs and interactive proofs are increasing, we believe that our framework offers two other complementary proving techniques (using deep and shallow instantiation) to the classical Event-B provers.

2) Deep and shallow modelling: In deep modelling, all Event-B concepts related to variables, events, guards, invariants, substitutions and so on are defined as instances of the developed meta-theory. For shallow modelling, a context and an abstract generic model instantiating the meta-theory with Event-B concepts are defined. They exploit the built-in inductive proof process offered by Event-B/Rodin. Both approaches allow us to double-check, based on different proof mechanisms, design decisions during system development.

Last, such approaches offer a shared format for Event-B models. Indeed, deep modelling style relies on first-order logic and set theory in contexts that can be exported to any other proof assistant, like Coq, PVS and Isabelle/HOL. For shallow modelling, export to other close modelling techniques like TLA is straightforward. Such export makes it possible to exploit other proof assistants than Event-B/Rodin.

3) Extensible framework: The EB4EB framework allows for the extension of the core reasoning mechanism of Event-B by including new POs defined as predicates in theories. Our theories for reflexive Event-B, for example, allowed us to reason about deadlock freedom, reachability, and so on. This capability makes Event-B/Rodin extensible as it can be enriched with additional POs not available in native Event-B.

IX. Conclusion

The EB4EB framework we presented allows users to manipulate Event-B features *explicitly* using reflection and meta-modelling concepts. It relies on Event-B theories that define data-types, operators, WD, theorems, and proof rules to formalise the semantics of Event-B. The developed theories enable manipulation of the static and dynamic properties of Event-B, including new POs associated to deadlock freeness, liveness, reachability, composition/decomposition, and so on. Deep and shallow embedding are used to instantiate the defined theories of the EB4EB framework. Furthermore, the Rodin tools have been used to formalise the development of EB4EB-related activities. We have demonstrated the approach on the clock model developed in both core modelling language and deep and shallow modelling. Both deep and shallow instantiated clock models preserve the required safety properties and functional behaviour encoded in theories.

Note that the EB4EB modelling concepts are formalised once and for all, meaning that they are reusable and don't need to be proven again. However, these theories must be instantiated in new developments, and the generated WD POs must be discharged to check instantiation is correct.

The developed theories have been applied on several case studies to assess the expressiveness, effectiveness, portability, and scalability of our approach. Our future goal is to analyse and identify Event-B refinement operations and its semantics to be included in the EB4EB framework. We plan to deploy the EB4EB framework to enhance and extend the reasoning mechanism by supporting other externally defined POs. In particular, we target the definition of domain specific POs issued from domain specific analyses of Event-B models, like continuous behaviours, human machine interaction, and so on. Analyzing and certifying existing plug-ins, such as code generation and composition/decomposition, is on the agenda. Our long-term goal is to use Dedukti [7] to import/export the Event-B theory and models into proof assistants like Coq, PVS and Isabelle/HOL using the deep modelling approach.

References

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

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Michael Leuschel, Matthias Schmalz, and Laurent Voisin. Proposals for mathematical extensions for Event-B. *Tech. Rep.*, 2009.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Michael Leuschel, Matthias Schmalz, and Laurent Voisin. Proposals for mathematical extensions for Event-B. Technical report, 2009.

[4] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[5] A. Anand, S. Boulier, C. Cohen, M. Sozeau, and N. Tabareau. Towards certified meta-programming with typed template-coq. In Jeremy Avigad and Assia Mahboubi, editors, *9th International Conference, ITP. Part of FloC 2018*, volume 10895 of *LNCS*, pages 20–39. Springer, 2018.

[6] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[7] Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. Dedukti: A Universal Proof Checker. In *Journées communes LTP - LAC*, Orléans, France, 2012.

[8] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, page 129–156. North-Holland Publishing Co., 1992.

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

[10] Michael J. Butler and Issam Maamria. 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*, pages 67–81, 2013.

[11] David Christiansen and Edwin Brady. Elaborator reflection: Extending idris in idris. *SIGPLAN Not.*, 51(9):284–297, September 2016.

[12] Guillaume Dupont, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. Event-b hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021.

[13] Guillaume Dupont, Yamine Aït Ameur, Marc Pantel, and Neeraj Kumar Singh. Formally verified architecture patterns of hybrid systems using proof and refinement with event-b. In *Rigorous State-Based Methods - 7th International Conference, ABZ*, volume 12071 of *LNCS*, pages 169–185. Springer, 2020.

[14] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[15] Benja Fallenstein and Ramana Kumar. Proof-producing reflection for HOL - with an application to model polymorphism. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, Proceedings*, volume 9236 of *LNCS*, pages 170–186. Springer, 2015.

[16] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code Generation for Event-B. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, pages 323–338, Cham, 2014. Springer.

[17] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[18] Farhad Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH Zurich, 01 2008.

[19] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from Event-B models. In *Proceedings of the Symposium on Information and Communication Technology, SoICT*, pages 179–188, 2011.

[20] C. Metayer and L. Voisin. The Event-B mathematical language. Technical report, 2009.

[21] César Muñoz and John Rushby. Structural embeddings: Mechanization with method. In *International Symposium on Formal Methods*, pages 452–471. Springer, 1999.

[22] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[23] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[24] Matthias Schmalz. The logic of event-b. *Technical Report/ETH Zurich, Department of Computer Science*, 698, 2011.

[25] Renato Silva and Michael Butler. Shared Event Composition/Decomposition in Event-B. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, pages 122–141. Springer, 2012.

[26] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The MetaCoq Project. *J. Autom. Reason.*, 64(5):947–999, 2020.

[27] Paul van der Walt. Reflection in Agda. Master's thesis, 2012.