Reflexive Event-B: Semantics and Correctness The EB4EB framework

Peter Riviere, Neeraj Kumar Singh, Yamine Aït-Ameur

Reflexive Event-B: Semantics and Correctness The EB4EB framework

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

***Abstract*—The Event-B method enables correct by construction modelling of systems. It relies on set theory and first-order logic, to describe a series of refined system models expressed as a set of events modifying state variables. Invariants and theorems are introduced to express system properties submitted to the proof system associated to Event-B. While Event-B has proven its efficiency for the proof of this type of properties, it does not offer powerful means allowing the explicit description of properties other than safety and specific forms of reachability. Checking other properties like deadlock-freeness, liveness or event scheduling, etc. requires ad hoc modelling techniques and external tools such as model checkers or other proof systems. This paper presents *EB4EB*, a new modelling framework offering the capability to introduce formally defined Event-B extensions, in particular new proof obligations corresponding to new properties. It is based on meta-modelling techniques. It includes a theory (a meta-theory) modelling Event-B and offers means for explicit manipulation of Event-B features and an extension mechanism to explicitly formalise and prove other properties. This reflexive framework relies on a trace-based semantics of Event-B and introduces a set of Event-B theories defining data types, operators, well-defined conditions, theorems and proof rules to define Event-B constructs and their semantics. Deep and shallow instantiation mechanisms are set up to instantiate the obtained meta-theory. The EB4EB framework and its instantiation mechanisms are developed in Event-B using the Rodin platform ensuring correctness and internal consistency of the defined theories. Lamport's clock example, instantiating EB4EB in both shallow and deep mechanisms, is used to evaluate the proposed approach.**

I. Introduction

Metamodelling is a standard approach in software engineering for describing abstractions of models and properties, as well as performing analysis to guarantee the quality of the developed models, rules, operations, and constraints. This approach is widely used in the field of model-driven engineering. Formal methods also offer frameworks to support meta-modelling facilities through the development of meta-theories, axiomatising metamodels, to represent higher-level reasoning concepts used in the specification, development, and verification of complex systems [10], [24], [37], [47].

Event-B [2] enables correct by construction modelling of systems. It relies on set theory and first-order logic (FOL), to describe a series of refined system models expressed as a set of events modifying state variables. Invariants and theorems are introduced to express system properties submitted to the proof system associated to Event-B. An integrated development environment (IDE), Rodin [4], enables model development as well as the automatic generation of proof obligations. The associated proof process ensures system consistency thanks to the proof system it supports. Rodin has been extended with several plugins including composition/decomposition [46], Theory plug-in [3], [17], code generation [26], [35] and so on. In particular, the theory plugin [3], [17] enables to extend the core concepts of Event-B by defining new data types, theories, and operators that can be used in Event-B models. In addition to the classical theories for lists, trees, graphs and reals, several other theories have been developed to support complex constructs like continuous features [21], [22] or domain knowledge ontologies [33], [34].

For checking system consistency and refinement, Event-B and its Rodin IDE rely on induction and provide automatically generated proof obligations for invariant preservation, variant progress, events feasibility, proof theorems, guard strengthening, refinement, and so on. To check additional properties such as deadlock freeness, liveness, reachability, event scheduling, and domain-specific properties, the designer must provide an adhoc Event-B model based on the core Event-B features or must rely on other modelling tools, such as model checkers and external interactive theorem provers. Indeed, there is no mechanism for encoding and reasoning on Event-B trace semantics. Moreover, Event-B does not offer the capability to manipulate Event-B concepts explicitly to formalise properties in a generic an reusable setting. *Therefore, performing advanced reasoning level by introducing new, reusable and automatically generated POs for any designed model is not yet possible.*

Our objective is to define a novel framework based on a reflexive formalisation, using Event-B, of a meta-theory allowing to manipulate Event-B concepts. This theory is enriched by new concepts allowing to formalise and generate new proof obligations formalising advanced and reusable reasoning mechanisms.

This paper extends our work presented in [41]. Our primary contribution is to present an EB4EB framework based on meta modelling concepts, including trace semantics of the core Event-B, for explicitly manipulating Event-B features and extending its reasoning mechanism to support other properties. In order to express the Event-B core modelling constructs, trace-based semantics, and new proof obligations for the EB4EB framework, a set of theories including data types, operators, well-defined conditions, theorems, and proof rules is developed. In addition, these theories enable manipulation of static and dynamic concepts of Event-B features as well as

defining new proof obligations to support a *reusable* (defined once and for all) advanced reasoning level. Two instantiation mechanisms, deep and shallow, associated to these generic theories, are introduced to exploit this correct by construction framework to support new Event-B model analyses. The formalised trace-based semantics is used to prove the soundness of the defined models analyses associated to the native and new generated Event-B POs allowed by EB4EB. Finally, we evaluate our approach on Lamport's clock example.

This paper is organised as follows. Section II presents Event-B modelling concepts, including refinement and proof obligations. Section III describes reflexive concepts, related work, and the EB4EB framework. The core concepts of Event-B are described in Sections IV and V of the EB4EB framework. Section VI describes the trace semantics and the correctness of proof obligations is provided in Section VII. Deep and shallow embeddings are described in Section VIII. Section IX describes Lamport's clock example, which is used to describe the application of EB4EB framework by applying the deep and shallow embeddings in Section X. Section XI presents the EB4EB reasoning mechanism, including a new set of proof obligations. The proof process related to the development of the clock model and deadlock reasoning extension is described in Section XII. Finally, Section XIII concludes the paper and discusses future work.

II. Event-B

Event-B [2] is a correct-by-construction method supporting the development of large and complex systems. Its formal modelling language is based on set theory and first-order logic (FOL) and relies on the definition of state variables characterising systems state and a set of events to model state changes. A system model is designed as a series of refined intermediate models starting from an abstract model. The main components of the Event-B modelling language are summarised below.

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}.

`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.

- *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.

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

(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

- *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 mathematical definitions has been proposed in [3], [18]. This extension, like Isabelle/HOL [38] or PVS [39], allows to define new *theories* by introducing new data types, operators, theorems and proof rules. They can be further used in the core development of Event-B models.
- *Theory description.* Table IV shows core modelling elements for developing new theories. The core modelling elements are classified in different clauses known as data types, 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. Data types (`DATATYPES` clause) can be defined with *constructors*, and each constructor can have some *destructors*. Note that a destructor can also have an inductive definition.

Theory
THEORY Th
IMPORT Th1, ...
TYPE PARAMETERS E, F, ...
DATATYPES
Type2(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

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 [18], [21].

- *Well-definedness (WD) in Theories.* This useful clause associates *well-definedness* (WD) conditions to each operator defined in a theory. This condition restricts the use of an operator to its licit parameters (partial definitions). In particular, when a function is denoted as operator, this condition defines the domain of this function as well-definedness additional constraints. When the defined operator is used, a WD proof obligation is generated and must be discharged to ensure the correctness of the modeled specification as well as defined 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 with proof like predicate provers or SMT solvers.

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

III. The EB4EB Framework

A. Motivation

As mentioned in the introduction, Event-B extensions are not possible as the modelling language does not offer the capability to manipulate Event-B concepts as first-order objects. Meta-modelling features are not available in the core Event-B modelling language. Offering meta-modelling capabilities is the main idea of the EB4EB framework.

Embedding modelling language constructs in another modelling language is well accepted by the model-driven engineering. When this embedding is realised in the same modelling language, it is qualified as reflexive. Two embedding techniques have been identified: deep and shallow embeddings. Deep embedding describes explicitly the semantics and syntax of the source language in the logic of the host language, whereas shallow embedding simply expresses by translation the semantics of the source language in the semantics host language [16] (i.e. here the translator carries the semantics). Both approaches have their 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 logic of the host modelling language, 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 directly formalised in the modelling language enconding the transformation. It leads to limited access to the source modelling language constructs for formal verification, in particular when tracing verification results (e.g. counter-examples). Munoz et al. [37] 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.

In order to design a formal setting for defining Event-B extensions, the proposed EB4EB framework defines a reflexive embedding on Event-B in an Event-B theory. Before entering into the details of the EB4EB framework, we review some approaches of the literature which addressed the problem of embedding formal modelling languages in other formal modelling languages.

B. Related work

Several modelling languages use a reflexive approach to handle higher-order modelling concepts and their manipulation for improving reasoning mechanisms and other advanced level modelling features. Riccobene et al. [40] proposed the ASM-Metamodel (AsmM) for manipulating Abstract State Machine (ASM) [15] concepts like abstract machines, signatures, terms, rules, and so on. The developed API offers to express analyses and ASM tool extensions, such as requirements validation [42], model checking [7], animation [14], [19], flattener for the ASMETA framework [9], and reviewing ASM model by meta property verification [8]. Bicarregui et al. [11] proposed reflexive concepts for VDM [30] in a mathematical reasoning environment MURAL [31] to provide modelling and reasoning capabilities for higher-order concepts. The Event-B

API available in Rodin tools enables the development of core plug-ins such as model checker and animation ProB [32], code generation [26], [35], extending modelling features [28], [46].

The reflexive approach is not limited to modelling languages; other formal methods approaches related to type theory use it to manipulate their syntax and higher-order modelling concepts. For example, the reflexive approach is proposed for Agda [48], Lean [23], and Coq [5]. Moreover, this approach can be used in functional programming languages such as MetaML [49], and Template Haskell [45]. In [47], the authors proposed a framework in the MetaCoq project to define the semantic of Coq in order to support the certified meta-programming environment. This framework aided in the development of CertiCoq [6], a certified compiler of Coq. The reflection principle is implemented in Isabelle/HOL [24] to express HOL models as well as reasoning mechanisms in order to describe complex systems with self-replacement functionality. Similarly, Mitra et al. [36] proposed the reflection mechanism in PVS based on theories and templates to generalise proofs and make them highly reusable using *strategies* concepts for proving abstraction relation between automata.

Regarding the B method [1], Munoz et al. [37] proposed a formalisation in the higher-order theorem prover PVS [39]. In the same vein, Event-B is also formalised to ensure the correctness of modelling and reasoning concepts. Bodeveix et al. [12] proposed context formalisation in order to prove the theorems expressing properties on Event-B models. Schneider et al. [43] proposed the core semantics of Event-B, including refinement, in CSP [29], which is based on trace semantics. In [25], the authors proposed the Event-B formalisation to express the theory of institution, but it is not tool supported. Event-B modelling constructs are also formalised in Coq to express Event-B traces, and a set of theorems is proved in Coq to ensure the correctness of proof obligations.

Our approach provides a homogeneous framework for using the Event-B machine concept as a first-class object in models, similar to Coq and HOL, and the user does not require to use different semantic frameworks to manipulate it, as described in CSP [43] and Coq [20]. Thus, our work is free of semantic heterogeneity constraints, which could reduce embedding correctness. Our method can handle two types of semantics: *native* and *axiomatic*. The first *native* semantics deal with the core concept of state-based modelling and refinement, while the second *axiomatic* semantics deal with first-order logic. These semantical representations play a central role in analysing and ensuring any complex systems that have been built correctly in a non-intrusive manner.

In [20], [27], [32], [43], [44], trace-based semantic was used to validate the Event-B modelling and analysis concepts. Most of this work emphasises on Event-B embedding in other formalisms or APIs to ensure the Event-B semantics, whereas our work provides a set of operators, axioms, and theorems developing a theory (a meta-theory) to manipulate and extend the core concepts of Event-B while preserving the semantics in the same formal modelling language. Moreover, our framework allows expressing some important properties such as liveness, deadlock freeness, event scheduling and so on. In addition, this framework also provides a set of operators to represent the Event-B trace semantics in order to ensure the correctness of modelling features, functionalities, properties, and proof obligations. This framework enables non-intrusive analysis for checking the correctness of complex systems. As far as we know, this is the first *reflexive* framework for the Event-B method to analyse a system systematically.

C. The EB4EB framework

The EB4EB framework is based on first-order logic and set theory, which enable a simple and easy mechanism for exporting Event-B core concepts, including semantics, in other formalisms without redoing the entire work, and its use does not impose many well-typed proof obligations. This framework supports two types of proof processes: the first is operational with axiomatic semantics in the Event-B context, and the second is induction to handle machine mechanisms similar to Event-B native proof process.

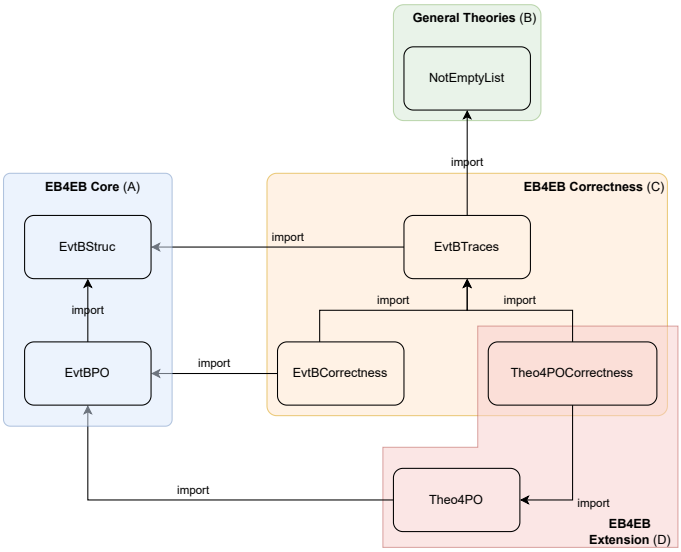


Fig. 1: Architecture of the theories

The EB4EB framework defines a set of generic and reusable Event-B theories formalising all the concepts available in an Event-B model. It uses an algebraic style with concept types, constructors, operators and a set of axioms and theorems providing their properties. This theory is instantiated to define specific Event-B models. Two instantiation mechanisms have been defined: *deep* and *shallow*. Fig. 1 depicts the architecture of this framework. The core theory (Fig 1.A) models the core Event-B method. The correctness of the defined proof obligations with respect to the provided-trace based semantics is supported by Fig. 1.B and Fig. 1.C. Last, the extensions of the framework and their correctness are presented in the theories of Fig. 1.D.

In the following, we provide a detailed presentation of this framework. The formalisation of the model the constructs (Section IV), Event-B proof obligations (Section V) and the semantics and the theorem guaranteeing the correctness of the approach (Sections VI and VII) are presented. The two instantiation mechanisms are presented in Section VIII.

IV. EB4EB STRUCTURE (SEE FIG. 1.(A))

This section introduces the EvtBStruc Event-B theory of EB4EB (see Fig. 1.(A)) dedicated to the definition of the structure of an Event-B model. It includes data types constructors and well-structured machine.

A. Data types and constructors

In order to model states and events (transitions), the two main components of a state-transition system, the Event-B meta-theory EvtBStruc introduces, in the TYPE PARAMETERS clause, two polymorphic type parameters, represented as carrier sets, $STATE$ and $EVENT$ (see Listing 1). The type parameter $STATE$ is used to represent a set of variables. An explicit description of each variable is not required at this abstract level. Indeed, the type parameter $STATE$ abstracts the state as a Cartesian product of all variables. At the 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 single constructor Cons_machine is defined in the CONSTRUCTOR clause associated with destructors to represent and access various constituents of Event-B components. The following destructors are defined.

- $Event$ - a set of machine events;
- $State$ - a set of machine states;
- $Init$ - an initialisation event;
- $Progress$ - a set of progress events;
- AP - the after-predicate defining the initialisation state;
- Grd - a set of event guards as a pair made of allowed state and an event;
- BAP - a set of before after-predicates as a triple made of an event and before and after states;
- Inv - machine invariants as a set of licit states;
- Thm - machine theorems as a set of licit states;
- $Variant$ - machine variants as a pair associating an integer to a state;
- $Ordinary$ - a set of ordinary events, i.e. events which do not constrain the variant;
- $Convergent$ - a set of events decreasing a variant.

```
THEORY EvtBStruc // Part 1
  TYPE PARAMETERS EVENT, STATE
  DATA TYPES
    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 Definition

B. Well Structured Machine

The DATATYPES clause defines a constructor and destructors to access the Event-B modelling components. The above-defined constructors and destructors contain typing information only. Therefore, they may lead to ill-defined datatype definitions. It is necessary to associate well-definedness (WD) conditions that restrict their use in consistent cases. For example, the BAP destructor is a relation between events and states, but the initialisation event is not concerned by this before-after relation and shall be excluded from the set of events involved in a BAP.

In order to avoid such ill-defined typing definitions, we introduce a set of new operators in Listing 2 each of which is equipped with WD conditions. In Listing 2, the first well-defined operator BAP_WellCons 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_WellCons 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 definedness condition of the Event operator, the Event_WellCons is declared. Its direct definition states that the union of the progress and initialisation events equals the machine events.

The direct definition of the next Variant_WellCons 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_WellCons operator shows that the union of convergent and ordinary events equals mutually exclusive machine events and the initialisation event is an ordinary event.

The last Machine_WellCons operator is important. It collects all the well-definedness conditions of all the defined operators. Its direct definition is the conjunction of all other well-defined operators. It represents the global well-defined condition associated with an Event-B machine m.

```
//THEORY EvtBStruc  Part 2
  OPERATORS
    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))
    Variant_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ◁ Variant(m) ∈ Inv(m) → ℤ
    Tag_Event_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        partition(Event(m), Ordinary(m), Convergent(m))∧
            Init(m) ∈ Ordinary(m)
    Machine_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        BAP_WellCons(m)∧
        Grd_WellCons(m)∧
        Event_WellCons(m)∧
        Tag_Event_WellCons(m)∧
        Variant_WellCons(m)
```

Listing 2: Machine Well Constructed Operators

V. EB4EB Proof obligations (see Fig. 1.(A))

Once Event-B models are structurally well built, semantics can be addressed. This section presents a set of proof obligations formalised as operators of the `EvtBPO` theory (see Fig. 1.(A)) of the EB4EB framework. These operators express and help to discharge the generated proof obligations given in Section II, such as `INV`, `FIS`, `NAT` and `VAR`. Their definitions are inductive as they apply to the initialisation and then to all other events. The formalisation relies on an encoding of FOL expressions set comprehension. Below, we formalise all POs at the meta-theory level.

A. Feasibility Proof Obligation (FIS)

1) Principle: The objective of this proof obligation rule is to ensure that when the guard of an event holds, its BAP allows to reach the next state, i.e. the action defined by the BAP is feasible. It is defined as,

$$\frac{M \vdash \forall i, \alpha, x \cdot G_i(\alpha, x) \wedge I(x) \Rightarrow (\exists x' \cdot BAP_i(\alpha, x, x'))}{M \vdash FIS}$$

2) FIS Operators formalised in EB4EB: The feasibility rule is encoded in the Event-B meta-theory presented in Listing 3. We defined three operators `Mch_FIS_Init`, `Mch_FIS_One_Ev`, and `Mch_FIS`. The first two operators represent the base case and induction case for the feasibility PO, respectively. The first operator is declared with one argument machine m, and its direct definition for the base case ensures that the intersection of machine invariants ($Inv(m)$) and machine after-predicates ($AP(m)$) should not be empty. The second operator is declared with two arguments machine m and event e. 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. The last operator checks the machine feasibility by induction. Its direct definition shows that the machine is feasible at initialisation (base case) and for all progress events (inductive case).

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
    $Mch\_FIS\_Init(m) \wedge$
    $(\forall e \cdot e \in Progress(m) \Rightarrow Mch\_FIS\_One\_Ev(m, e))$

Listing 3: Feasibility proof obligation operators

B. Invariant Proof Obligation (INV)

1) Principle: Invariant proof obligation rule ensures that each event of a machine preserves the invariant. It uses two abbreviations to increase it readability. It is defined as,

$$initCase_{INV} \equiv \forall \alpha, x' \cdot AP(\alpha, x') \Rightarrow I(x')$$

$$inducCase_{INV}(i) \equiv \forall \alpha, x, x' \cdot G_i(\alpha, x) \wedge BAP_i(\alpha, x, x') \wedge I(x) \Rightarrow I(x')$$

$$\frac{M \vdash initCase_{INV} \qquad M \vdash \forall i \cdot i \in 1..n \Rightarrow inducCase_{INV}(i)}{M \vdash INV}$$

Here, for an Event-B machine containing n progress events, $G_i(x, \alpha)$ and $BAP_i(\alpha, x, x')$ represent the guard and the before-after predicate of the event $e_i(i \in 1..n)$

2) INV Operators in EB4EB: The invariant proof obligation rule is also formalised in Event-B meta-theory presented in Listing 4. Three predicate operators, `Mch_INV_Init`, `Mch_INV_One_Ev` and `Mch_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 the first operator states that the machine after predicate ($AP(m)$) is a subset of machine invariant. The next operator, `Mch_INV_One_Ev`, ensures that the BAP of progress event e preserves the invariants if guards and invariants are held before. The last operator is defined to check each event preserves the machine invariants by induction. The direct definition of this operator shows that the machine invariant is preserved at initialisation and for all progress events, i.e. invariant for all machine events.

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
    $Mch\_INV\_Init(m) \wedge$
    $(\forall e \cdot e \in Progress(m) \Rightarrow Mch\_INV\_One\_Ev(m, e))$

Listing 4: Invariant proof obligation operators

C. Natural Variant Proof Obligation (NAT)

1) Principle: The objective of this proof obligation rule is to ensure that a proposed numeric variant is a natural number under the guards of each convergent event. The variant proof obligation rule is defined as,

$$\frac{M \vdash \forall i, \alpha, x \cdot e_i \in convergent \wedge G_i(\alpha, x) \wedge I(x) \Rightarrow v(x) \in \mathbb{N}}{M \vdash NAT}$$

2) Natural Variant in EB4EB: Listing 5 shows two operators, `Mch_NAT_One_Ev` and `Mch_NAT`, to define a variant for an event e as a natural number and that all convergent events have a natural number as a variant, respectively. Their direct definitions are provided below.

Mch_NAT_One_Ev ***predicate*** $(m : Machine(STATE, EVENT),$
    $e : EVENT)$
  well-definedness $e \in Convergent(m)$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[\{e\}]] \subseteq \mathbb{N}$
Mch_NAT ***predicate*** $(m : Machine(STATE, EVENT))$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[Convergent(m)]] \subseteq \mathbb{N}$

Listing 5: Variant proof obligation operators

D. Variant decrease Proof Obligation (VAR)

1) Principle: This proof obligation rule ensures that each convergent event decreases the proposed numeric variant. This proof obligation rule is defined as,

$$\frac{M \vdash \forall i, \alpha, x, x' \cdot e_i \in Convergent \wedge G_i(\alpha, x) \quad \wedge I(x) \wedge BAP(x, x', \alpha) \Rightarrow v(x') < v(x)}{M \vdash VAR}$$

2) Variant decrease in EB4EB: Two new operators, `Mch_VARIANT_One_Ev` and `Mch_VARIANT`, are declared to represent convergent properties in Listing 6. The `Mch_VARIANT_One_Ev` 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 and the required WD conditions for the variants. Similarly, the operator `Mch_VARIANT` generalises the definition of convergence, it checks the required properties for all convergent events of machine m.

```
Mch_VARIANT_One_Ev predicate (
    m : Machine(STATE, EVENT), e : EVENT, s : STATE)
  well-definedness Variant_WellCons(m),
      Mch_INV_One_Ev(m, e), e ∈ Progress(m),
      e ∈ Convergent(m), s ∈ Inv(m), s ∈ Grd(m)[{e}]
  direct definition
    ∀sp · sp ∈ BAP(m)[{e}][{s}]
      ⇒ (Inv(m) ◁ Variant(m))(s) >
          (Inv(m) ◁ Variant(m))(sp)
Mch_VARIANT predicate (m : Machine(STATE, EVENT))
  well-definedness Variant_WellCons(m), Mch_INV(m),
      BAP_WellCons(m), Tag_Event_WellCons(m),
      Event_WellCons(m)
  direct definition
    ∀e, s · e ∈ Event(m) ∧ e ∈ Convergent(m)∧
      s ∈ State(m) ∧ s ∈ Inv(m) ∧ s ∈ Grd(m)[{e}]
      ⇒ Mch_VARIANT_One_Ev(m, e, s)
```

Listing 6: Variant decrease proof obligation operators

E. Theorem THM

1) Principle: This rule ensures that a context or machine theorem can be proven. Theorems are important for simplifying some proofs. The theorem proof obligation rule states that theorems are deduced from invariants, it is defined as,

$$\frac{M \vdash \forall x \cdot I(x) \Rightarrow Thm(x)}{M \vdash THM}$$

2) Theorem THM in EB4EB: The declared operator `Mch_THM` consists of one argument machine m, and its direct definition shows that the invariants are a subset of theorems in Listing 7.

```
Mch_THM predicate (m : Machine(STATE, EVENT))
  direct definition
    Inv(m) ⊆ Thm(m)
```

Listing 7: Theorem proof obligation operator

F. Proof Obligation Generation

Listing 8 shows the most important predicate operator `check_Machine_Consistency`. When this predicate is used as a theorem at the instance level, it allows generating automatically all possible POs. This predicate expresses the proof obligations as the conjunction of all the proof obligations related to Event-B constituents expressed using the predicate operators previously defined. By the WD condition associated to this operator, it only applies to well-built machines, as defined by the properties described in Section IV.

```
check_Machine_Consistency predicate (
    m : Machine(STATE, EVENT))
  well-definedness Machine_WellCons(m)
  direct definition
    Mch_INV(m)∧
    Mch_FIS(m)∧
    Mch_NAT(m)∧
    Mch_VARIANT(m)∧
    Mch_THM(m)
```

Listing 8: Machine Proof Obligation

Note that the POs generation mechanism described in this section can be seen as another approach to generating them. Instead of using the PO generator of the RODIN platform, one can use the WD and Theorem proof obligations obtained by the use of the EB4EB theories.

VI. Trace's Semantics of Event-B

In this section, we present a trace-based semantics for Event-Machines and then relate it to the proof obligations formalised in the previous section.

A. Event-B traces

For a given machine M, a sequence of states $tr = s_0 \mapsto s_1 \mapsto \ldots \mapsto s_n$ describes a trace of machine M iff,

1) tr contains at least one state. A trace includes at least the initialisation event;
2) s_0 is the initial state satisfying the After Predicate (AP) of the initialisation event;
3) for each successive states s_i, s_{i+1} of the sequence tr, there exists a progress event e such that
   - s_i satisfies the guard of e and
   - $s_i \mapsto s_{i+1}$ satisfies the Before-After Predicate of the event e

This notion of trace is formalised in an Event-B theory.

B. Trace's Semantics in EB4EB

1) Non-empty lists (see Fig. 1.(B)): To formalise the trace semantics of Event-B, we develop another theory `NotEmptyList` as shown in Listing 9. This theory relies on the list data type and declares a List type `T`. The data type `NotEmptyList` has two constructors, one for describing the base case and one for describing the inductive case. The base case constructor has only one element in the list, while the inductive case constructor has one element at the head of the list and a tail to represent a list of other elements. In this theory, we define the `first` operator, which takes a list as an argument and returns the first element of the list. The list theory is used to represent a trace as a list of states, where the list is the inverse of the state sequence. If n is the size of the list, then the state s_i is at the index $n - i$ of the list, and the state s_n is at the head of the list.

```
THEORY NotEmptyList // Part 1
  TYPE PARAMETERS T
  DATA TYPES
    NotEmptyList(T)
    CONSTRUCTORS
      cons(el : T,
           next : NotEmptyList(T))
      base_case(base : T)
  OPERATORS
    first expression (
        l : NotEmptyList(T))
      recursive definition
        case 1:
          cons(t, q) => t
          base_case(t) => t
```

Listing 9: An inductive list

2) A theory of Event-B traces (See Fig. 1.(C)): The formalisation of Event-B machine traces is proposed in the theory of Listing 10. This theory imports the two developed theories, `EvtBStruc` and `NotEmptyList` allowing to access to the Event-B features already formalised and to lists respectively. It defines two operators: `IsATrace` and `IsANextState`. The first operator is a predicate that checks if a trace tr of a machine m is a trace of this machine. It is defined inductively on the trace structure and refers to the second operator `IsANextState` to check that every state in the trace is a correct next state. The direct definition of this operator states that there exists a progress event e such that s belongs to the guard of event e and $s \mapsto sp$ satisfies the Before-After predicates of event e.

```
THEORY EvtBTraces IMPORT EvtBStruc , NotEmptyList
  TYPE PARAMETERS STATE, EVENT
  OPERATORS
    IsATrace predicate (tr : NotEmptyList(STATE),
          m : Machine(STATE, EVENT))
      recursive definition
        case tr :
          base_case(s) => s ∈ AP(m)
          cons(sp, q) =>
                IsANextState(sp, first(q), m) ∧ IsATrace(q, m)
    IsANextState predicate (sp : STATE ,s : STATE ,
          m : Machine(STATE, EVENT))
      direct definition
        ∃e · e ∈ Progress(m)∧
             s ∈ Grd(m)[{e}] ∧ s ↦ sp ∈ BAP(m)[{e}])
```

Listing 10: Inductive trace of Event-B

VII. EB4EB Correctness (see Fig. 1.(B,C))

The correctness of the expression of each proof obligation defined in the Event-B Theory presented in section V is established at the trace semantics level of Section VI. Correctness is stated according to the definition of the proof obligations available in the Event-B reference book [2]. The EB4EB is set up for this purpose. A theorem ensuring that the defined proof obligation entails the property on the traces is formalised and proved.

A. Principle (See Fig.1.(C))

To establish correctness, we introduce another Event-B theory $EvtBCorrectness$ that imports the $EvtBPO$ and $EvtBTraces$ two theories (See Fig.1.(C)). It includes a set of theorems stating that the expressed PO implies that the PO property holds on the trace.

We demonstrate our approach using invariant proof obligation. In this case, we ensure that invariant is a valid machine invariant if any state in a trace satisfies it.

B. Correctness of the Invariant PO formalised in EB4EB

In order to define the theorem ensuring the correctness of the definition of the Invariant PO defined in Section V-B2, the $NotEmptyList$ theory has been extended with the `AllAre` operator (see Listing 11) checking that the predicate $pred$ (expressed using set-theoretical "belongs to" relationship) holds for all the elements of a non-empty list l (used later to model a trace). It is recursively defined on the elements of a non-empty list

```
// THEORY NotEmptyList Part 2
OPERATORS
    AllAre predicate (l : NotEmptyList(T) ,pred : ℙ(T))
      recursive definition
        case l :
          base_case(el) => el ∈ pred
          cons(t, q) => t ∈ pred ∧ AllAre(q, pred)
END
```

Listing 11: New Operator on the list

Listing 12 presents the proved theorem for checking the Invariant's PO correctness. It states that if for all traces tr of any well structured Event-B machine m satisfying the invariant PO Mch_INV of Listing 4, then the invariant property $inv(m)$ of machine m holds in any state of the trace tr. The previous $AllAre(tr, inv(m))$ operator is used for this purpose.

```
THEORY EvtBCorrectness
  IMPORT EvtBTraces , EvtBPO
  TYPE PARAMETERS STATE, EVENT
  THEOREMS
    thm1 :
      ∀m, tr · m ∈ Machine(STATE, EVENT)∧
        Machine_WellCons(m) ∧ IsATrace(tr, m) ∧ Mch_INV(m)
            ⇒ AllAre(tr, Inv(m))
END
```

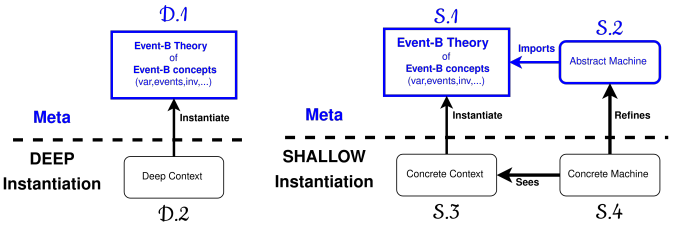
Listing 12: Theorem of correction of the proof obligation

The theorem of Listing 12 has been proved using the proof system of the Rodin platform. The main proof step of this proof is a case-based proof on the structure of the trace by unfolding the definition of the $AllAre$ operator. Following this approach, similar theorems have been defined and proved for the remaining proof obligations.

VIII. Modelling Event-B machines in EB4EB

In the previous section, we presented the theory axiomatising Event-models. The well-definedness conditions and the relevant theorems introduced allows to check the correctness of the specific models defined as instances of this generic theory. Below, we describe the defined instantiation mechanisms.

A. Instantiation Methodology



(a) Deep Modelling (b) Shallow Modelling

Fig. 2: EB4EB framework

Two instantiation mechanisms, depicted in Fig. 2, are envisioned: *deep* and *shallow* modelling.

Deep modelling consists in the definition of the various elements composing a machine conforming to the `EvtBStruc`. At instantiation, these definitions are provided in an Event-B context, where witnesses for type parameters STATE and EVENT are provided. Deep instantiation mechanism is recommended when manipulating and/or reasoning on Event-B

features as first order objects is required. In particular, this mechanisms allows for the extension of Event-B to formalise and prove other properties not available in core Event-B.

Shallow modelling is close to shallow embedding [16]. It relies on an abstract Event-B machine and model instances are defined as a refinement of this machine. This instantiation mechanism is recommended when the model can be expressed and proved. The benefit of this mechanism is the use of the refinement operation associated to the built-in induction principle available in core Event-B.

To compare both mechanisms, one can state that deep instantiation mechanism offers the capability to extend the Event-B method to handle other type of properties not available in core Event-B while shallow instantiation mechanism allows for the use of the refinement operation offered by core Event-B.

In both mechanisms, the defined operators, when invoked in a machine or a context, automatically generate well-definedness and theorem POs that must be proven in order to ensure machine consistency. These two instantiation mechanisms are detailed below.

B. Deep modelling based instantiation (see Fig. 2a)

This instantiation mechanism consists in defining an instance of the data type $Machine$ in 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. All the Event-B constructs described in the `EvtBStruc` Meta-theory such as invariants, theorems, event guards and before after predicate and so on are defined in the form of axioms. Consistency is ensured by the introduction of the theorem `check_Machine_Consistency` corresponding to the *predicate* consistency operator of the Meta-theory. It generates two kinds of proof obligation: first the well-definedness PO to ensure that the Event-B machine is well built, and second the PO related to the Event-B machine consistency such as invariant preservation, and variant decreasing corresponding to the theorem proof obligation. Both proof obligations must be proved (see Listing 13). These obtained POs are proved using the Event-B Rodin theorem prover as well as the other supporting theorem provers.

Listing 13 represents the skeleton of a context representing an instantiated machine in which each axiom characterises different components of the Event-B machine for reasoning and analysis using EB4EB framework. Our goal is to generate an instance machine automatically, thus the skeleton of the context model is fixed in a sequence of axioms. These sequences are: $axm1$ - to define partitions set Ev of machine events; $axm2$ - to define a machine m as an instance of theory's data type by instantiating $EVENT$ as an event set Ev, and $STATE$ as a cartesian product of variables type; $axm3$ - to set event accessor as partition event set Ev; $axm4$ - to set state accessor as a cartesian product variables type; $axm5$ - to set initial event of a machine m; $axm6$ - to define a set of progress events; $axm7$ - to define a set of machine after-predicates; $axm8$ - to define a set of machine guards; $axm9$ - to define a set of machine before-after predicates; $axm10$ - to define a set of machine invariants; $axm11$ - to define a set of machine theorems; $axm12$ - to define a set of a machine variants; $axm13$ - to define a set of ordinary events; $axm14$ - to define a set of convergent events. Finally, the theorem $check\_Machine\_Consistency$ is defined to ensure that the machine m is well-constructed and consistent by satisfying all associated proof obligations. A THM PO is generated for this theorem.

```
CONTEXT Deep
SETS Ev, ...
CONSTANTS mch, ...
AXIOMS
  axm1: partition(Ev, ...)
  axm2: mch ∈ Machine(..., Ev)
  axm3: Event(mch) = Ev
  axm4: State(mch) = ...
  axm5: Init(mch) = ...
  axm6: Progress(mch) = {...}
  axm7: AP(mch) = {...}
  axm8: Grd(mch) = {...}
  axm9: BAP(mch) = {...}
  axm10: Inv(mch) = {...}
  axm11: Thm(mch) = {...}
  axm12: Variant(mch) = {...}
  axm13: Ordinary(mch) = {...}
  axm14: Convergent(mch) = {...}
THEOREMS
  thm1: check_Machine_Consistency(mch)
END
```

Listing 13: A skeleton of a machine in the deep modelling

C. Shallow modelling based instantiation (see Fig. 2b)

As mentioned above, this mechanism introduces a context and an abstract machine to be refined by the instance Event-B model. Listings 14 and 15 show these context and machine of the abstract model for shallow instantiation. The context defines the sets Ev and St as instances of the type parameters for events and states. For this purpose, a constant mch is introduced as a member of $Machine(St, Ev)$.

```
CONTEXT ShallowGenCtx
SETS St, Ev
CONSTANTS mch
AXIOMS
  axm1: mch ∈ Machine(St, Ev)
END
```

Listing 14: A static element of abstract machine ($\mathcal{S}.2$)

In the abstract machine model, two variables s and $InitDone$ (for scheduling event triggering) are declared in the $inv1 - inv2$ invariant clauses. These variables are set in the INITIALISATION event. $inv3$ ensures that the invariant $Inv(mch)$ of the instance model is satisfied. In this model, we define three events: `Do_Init`, `Do_Ordinary`, and `Do_Convergent` whose actions modify the state using the AP (for initialisation) and BAP operators ($act1$). The first event is used to initialise state variables in action ($act1$). Its guards ensure that $InitDone$ is $FALSE$ ($grd1$), and the feasibility and invariants hold for the $Init$ event ($grd2$). The `Do_Ordinary` event updates the machine state s for an event e annotated as $Ordinary$. Its guards state that $InitDone$ is TRUE; the event e is a progress and ordinary event ($grd2$); the machine state s belongs to Grd of e ($grd3$); and feasibility and invariant properties of mch hold for the event e ($grd4$). Similar to the ordinary event, the last event `Do_Convergent` contains additional guards $grd2$ to tag the

event e as convergent and $grd6$ to ensure that the variant properties of mch 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 spirit of TLA$^+$, where the *progress* event can be refined by *ordinary* and *convergent* events later in further refinement. In this instantiation mechanism, the proof process relies on the induction principle offered by Event-B. Following the shallow modelling principle, the proof of machine consistency is delegated to Event-B itself. The defined properties are verified in the machine refining the generic machine *ShallowMchGen*.

```
MACHINE ShallowGenMch
SEES ShallowGenCtx
VARIABLES
  s,
  InitDone
INVARIANTS
  inv1: s ∈ St
  inv2: InitDone ∈ BOOL
  inv3: InitDone = TRUE ⇒ s ∈ Inv(mch)
EVENTS
  INITIALISATION
  THEN
    act1: s, InitDone :| s' ∈ St ∧ InitDone' := FALSE
  END

  Do_Init
  WHERE
    grd1: InitDone = FALSE
    grd2: Mch_INV_Init(mch) ∧ Mch_FIS_Init(mch)
  THEN
    act1: s, InitDone :| s' ∈ AP(mch) ∧ InitDone := TRUE
  END

  Do_Ordinary
  ANY e
  WHERE
    grd1: InitDone = TRUE
    grd2: e ∈ Progress(mch) ∧ e ∈ Ordinary(mch)
    grd3: s ∈ Grd(mch)[{e}]
    grd4: Mch_INV_One_Ev(mch, e) ∧ Mch_FIS_One_Ev(mch, e)
  THEN
    act1: s :∈ BAP(mch)[{e}][{s}]
  END

  Do_Convergent
  ANY e
  WHERE
    grd1: InitDone = TRUE
    grd2: e ∈ Progress(mch) ∧ e ∈ Convergent(mch)
    grd3: s ∈ Grd(mch)[{e}]
    grd4: Mch_INV_One_Ev(mch, e) ∧ Mch_FIS_One_Ev(mch, e)
    grd5: Variant_WellCons(mch)
    grd6: Mch_VARIANT_One_Ev(mch, e, s)∧
          Mch_NAT_One_Ev(mch, e)
  THEN
    act1: s :∈ BAP(mch)[{e}][{s}]
  END

END
```

Listing 15: A generic abstract machine ($\mathcal{S}.2$)

Listing 16 presents a skeleton of an Event-B machine representing an instance formalising an Event-B model. Similarly to the deep modelling instantiation approach, a static part is described in another context. This skeleton machine refines the abstract machine (see Listing 15). It represents the dynamic part of the machine instantiate, where:

- **inv1** provides the gluing invariant of the abstract state s and the concrete one.
- each event is refined by the concrete one, and each concrete event provides witnesses ($WITH$ clause) for the event parameters.
- **grd1** is the guard's state, $InitDone$ is true or false depending on which events are refined, and the refinement of abstract state s in the concrete guard.
- **grd2** and **grd3** introduces instances of the guard, and BAP/AP.
- **act1** defines the concrete assignment as well as updates event parameters with a concrete one.

```
MACHINE ShallowMch REFINES ShallowGenMch SEES ...
VARIABLES
  ...,
  InitDone
INVARIANTS
  inv1: s = ... // Gluing invariant for abstract state
                // variables s and concrete state variables
EVENTS // Static parts describe in the Event-B context
  INITIALISATION
  WITH s': s' = ...
  THEN
    act1: ..., InitDone :| ... ∧ InitDone' = FALSE
  END
  Do_Init REFINES Do_Init
  WHERE
    grd1: InitDone = FALSE
    grd2: ... = AP(mch)
  WITH s': s' = ...
  THEN
    act1: ..., InitDone :| ... ∈ AP(mch) ∧ InitDone' = TRUE
  END
  ... REFINES Do_Ordinary // Refines Do_Convergent if the event
                          // has the tag convergent
  WHERE // Guard strengthening encode the PO describe
        // as a guard in the abstract m.
    grd1: InitDone = TRUE ∧ ... ∈ Grd(mch)[{...}]
    grd2: ... = Grd(mch)[{...}]
    grd3: ... = BAP(mch)[{...}]
  WITH e: e = ..., s': s' = ...
  THEN
    act1: ... :| ... ∈ BAP(mch)[{...}][{...}]
  END
  ...
END
```

Listing 16: A shallow modelling machine skeleton

IX. Case Study

To illustrate our approach, we have chosen the case study of a 24h hour clock originally defined by L. Lamport. 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 at midnight.
- **REQ3** The clock never stops.

In Listing 17, we describe the clock model that is formalised in the 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 than 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, in addition to the initialisation ordinary event $INITIALISATION$, we introduce three events: *tick_min* - to model the minute progress by 1; *tick_hour* - to model the hour progress by 1 (two

convergent events); and *tick_midnight* - to reset the clock at midnight (an ordinary event). The required guards are added in the defined events to update the minute m and hour h.

```
MACHINE Clock
VARIABLES m , h
INVARIANTS
  inv1 : m ∈ ℕ
  inv2 : h ∈ ℕ
  inv3 : m < 60
  inv4 : h < 24
THEOREMS
  thm1 : m < 59∨
         (m = 59 ∧ h < 23)∨
         (m = 59 ∧ h = 23)
VARIANTS
    24 ∗ 60 − 1 − (m + h ∗ 60)
EVENTS
  INITIALISATION
  THEN act1 : m, h :|
         m′ = 0 ∧ 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 17: A machine of clock

X. EB4EB deep and Shallow modelling of the clock case study

Below we present the two instantiation mechanisms corresponding to the clock model of Listing 17.

A. Deep modelling instantiation for the clock model

We describe the development of the clock case study using the deep modelling instantiation technique of Section VIII using the meta-theory introduced in Section IV and V. All constituents of the Clock model are explicitly expressed in terms of the `EvtBStruc` and `EvtBPO` Meta-theory constructs. The Clock Event-B model is represented as an Event-B context using the skeleton shown in the Listing 13 of the Section VIII, 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 17 is presented in Listing 18. 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.Note that $\mathbb{Z} \times \mathbb{Z}$ and Ev correspond respectively to the instantiation of the type parameters $STATE$ and $EVENT$ of the $EvtBPO$ theory. Furthermore, three axioms ($axm3 - axm5$) are used to instantiate $Event$ with the enumerated set Ev, $State$ with $\mathbb{Z} \times \mathbb{Z}$, and $Init$ with the event label $init$.

The next axiom ($axm6$) is instantiated with progress events. Axiom $axm7$ instantiates the after-predicate AP derived from the action of the initialisation event ($act1$) in the Clock machine. Similarly, axioms $axm8$ and $axm9$ are used to instantiate the guard Grd and the before-after predicate BAP with a set of guards and actions of all events derived from the Clock machine using comprehensive sets. The next axiom ($axm10$) is defined to instantiate invariant Inv using comprehensive sets derived from $inv1 - inv4$ of Listing 17, and $axm11$ is used to instantiate theorem Thm derived from $thm1$ of the clock model (see Listing 17). Axiom $axm12$ is used to instantiate the $Variant$ with the defined variant of the Clock model. The next axioms $axm13 - axm14$ instantiate the $Ordinary$ and $Convergent$ with a list of ordinary and convergent events, respectively. In this model, we have only two ordinary events $init$ and $tick\_midnight$ and two convergent events $tick\_min$ and $tick\_hour$.

Machine correctness. It is important to note the introduction, in Listing 18, of a *theorem* $thm1$ referring to the $check\_Machine\_Consistency$ operator. The automatically generated well-definedness PO associated to the $check\_Machine\_Consistency$ operator and the one for the theorem shall be proved. They entail machine correctness.

```
CONTEXT ClockDeep
SETS Ev
CONSTANTS clock , tick_min , tick_hour , tick_midnight , init
AXIOMS
  axm1 : partition(Ev,
           {init}, {tick_midnight}, {tick_hour}, {tick_min})
  axm2 : clock ∈ Machine(ℤ × ℤ, Ev)
  axm3 : Event(clock) = Ev
  axm4 : State(clock) = ℤ × ℤ
  axm5 : Init(clock) = init
  axm6 : Progress(clock) = {tick_midnight, tick_hour, tick_min}
  axm7 : AP(clock) = {m ↦ h | m = 0 ∧ h = 0}
  axm8 : Grd(clock) = {t ↦ (m ↦ h) | (
           (t = tick_min ∧ m < 59)∨
           (t = tick_hour ∧ m = 59 ∧ h < 23)∨
           (t = tick_midnight ∧ m = 59 ∧ h = 23))}
  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 : Inv(clock) = {m ↦ h | m ∈ ℕ ∧ h ∈ ℕ ∧ m < 60 ∧ h < 24}
  axm11 : Thm(clock) = {m ↦ h |
           m < 59 ∨ (m = 59 ∧ h < 23) ∨ (m = 59 ∧ h = 23)}
  axm12 : Variant(clock) = {m ↦ h ↦ v |
           v = 24 ∗ 60 − 1 − (m + h ∗ 60)}
  axm13 : Ordinary(clock) = {init, tick_midnight}
  axm14 : Convergent(clock) = {tick_min, tick_hour}
THEOREMS
  thm1 : check_Machine_Consistency(clock)
END
```

Listing 18: A deep instance of the clock machine ($\mathcal{D}.2$)

B. Shallow modelling instantiation for the clock model

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

All the constituents of the Clock model are explicitly expressed using the `EvtBStruc` and `EvtBPO` Meta-Theory constructs. The corresponding Clock Event-B model is represented as a context (Listing 19) and a machine (Listing 20) corresponding to the skeleton machine of Listing 16. The POs guaranteeing the correctness of the instantiation are obtained by the theorems and by guard strengthening POs generated by the refinement of the abstract machine of Listing 15.

In the same vein as shallow embedding, we use Event-B to preserve semantics. 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.

The concrete model refines the abstract generic model introduced above. The static elements of the clock model are described by the context of Listing 19 and dynamic elements are described in machine of Listing 20.

Static constituents (Event-B context). In the context of Listing 19, we define a constant pr in $axm1$ as a bijection relation between $(\mathbb{Z} \times \mathbb{Z})$ and St 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, $State$ with St, and $Init$ with the event $init$. Axiom $axm6$ is defined to instantiate progress events of the clock machine. Axiom $axm7$ defined invariant Inv using comprehensive sets derived from $inv1 - inv4$ of Listing 17. Axiom $axm8$ instantiates theorem Thm derived from $thm1$ of the clock model. Variant of the clock machine is introduced in $axm9$. Then two axioms ($axm10 - axm11$) are used to instantiate $Ordinary$ and $Convergent$ with a set of ordinary and convergent events.

Context correctness. We define four theorems ($thm1$-$thm4$) to check the proof obligation associated with the well-constructed event, well-constructed variant, well-constructed events tags (ordinary or convergent), and machine theorem. Once proved, these theorems guarantee that the context is well-defined and the required properties hold.

CONTEXT ClockShallowCtx
EXTENDS ShallowGenCtx
CONSTANTS tick_min , tick_hour , tick_midnight , init , pr
AXIOMS
axm1: $pr \in (\mathbb{Z} \times \mathbb{Z}) \rightarrowtail St$
axm2: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm3: $Event(mch) = Ev$
axm4: $State(mch) = St$
axm5: $Init(mch) = init$
axm6: $Progress(mch) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm7: $Inv(mch) = pr[\{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}]$
axm8: $Thm(mch) = pr[\{m \mapsto h \mid m < 59 \lor (m = 59 \land h < 23) \lor (m = 59 \land h = 23)\}]$
axm9: $Variant(mch) = \{s, v, m, h \cdot s = pr(m \mapsto h) \land v = (24 * 60 - 1 - (m + h * 60)) \mid s \mapsto v\}$
axm10: $Ordinary(mch) = \{init, tick\_midnight\}$
axm11: $Convergent(mch) = \{tick\_min, tick\_hour\}$
THEOREMS
thm1: $Event\_WellCons(mch)$
thm2: $Variant\_WellCons(mch)$
thm3: $Tag\_Event\_WellCons(mch)$
thm4: $Mch\_THM(mch)$
END

Listing 19: Instances for static elements: clock machine ($\mathcal{S}.3$)

Dynamic constituents (event-B machine). The abstract machine is refined in Listing 20 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 the `Tick_min` event, $grd1$ is a data refinement, it introduces the two concrete variables m and h. The two other guards $grd2$ and $grd3$ respectively refer to the concrete guard and before after predicate of the $tick\_min$ event defined in the context of Listing 19. 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. Similarly, the two other events `Tick_hour` and `Tick_midnight` are also obtained by refining the abstract events $Do\_Convergent$ and $Do\_Ordinary$ by providing witnesses and appropriate instantiations of guards and before-after predicates.

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. Here, the classical proof process associated to Event-B with its inductive reasoning is used.

MACHINE ClockShallow **REFINES** ShallowGenMch
SEES ClockShallowCtx
VARIABLES
m ,
h ,
$InitDone$
INVARIANTS
inv1: $s = pr(m \mapsto h)$ *// Gluing Invariant*
EVENTS
INITIALISATION
WITH
s': $s' = pr(m' \mapsto h')$
THEN
act1: $m, h, InitDone :\mid m' \in \mathbb{Z} \land h' :\in \mathbb{Z} \land InitDone' = FALSE$
END

Do_Init REFINES Do_Init
WHERE
grd1: $InitDone = FALSE$
grd2: $pr[\{0 \mapsto 0\}] = AP(mch)$
WITH
s': $s' = pr(m' \mapsto h')$
THEN
act1: $m, h, InitDone :\mid pr(m' \mapsto h') \in AP(mch) \land InitDone = TRUE$
END

Tick_min REFINES Do_Convergent
WHERE
grd1: $InitDone = TRUE \land pr(m \mapsto h) \in Grd(mch)[\{tick\_min\}]$
grd2: $pr[\{ms, hs \cdot ms < 59 \land hs \in \mathbb{Z} \mid ms \mapsto hs\}] = Grd(mch)[\{tick\_min\}]$
grd3: $\{ss, ssp, ms, hs, msp, hsp \cdot ss = pr(ms \mapsto hs) \land ssp = pr(msp \mapsto hsp) \land msp = ms + 1 \land hs = hsp \mid ss \mapsto ssp\} = BAP(mch)[\{tick\_min\}]$
WITH
e: $e = tick\_min$, s': $s' = pr(m' \mapsto h')$
THEN
act1: $m, h :\mid pr(m' \mapsto h') \in BAP(mch)[\{tick\_min\}][\{pr(m \mapsto h)\}]$
END

Tick_hour REFINES Do_Convergent
WHERE
grd1: $InitDone = TRUE \land pr(m \mapsto h) \in Grd(mch)[\{tick\_hour\}]$
grd2: $pr[\{ms, hs \cdot hs < 23 \land ms = 59 \mid ms \mapsto hs\}] = Grd(mch)[\{tick\_hour\}]$
grd3: $\{ss, ssp, ms, hs, msp, hsp \cdot ss = pr(ms \mapsto hs) \land ssp = pr(msp \mapsto hsp) \land msp = ms \land hsp = hs + 1 \mid ss \mapsto ssp\} = BAP(mch)[\{tick\_hour\}]$
WITH
e: $e = tick\_hour$, s': $s' = pr(m' \mapsto h')$
THEN
act1: $m, h :\mid pr(m' \mapsto h') \in BAP(mch)[\{tick\_hour\}][\{pr(m \mapsto h)\}]$
END

Tick_midnight REFINES Do_Ordinary
WHERE
grd1: $InitDone = TRUE \land pr(m \mapsto h) \in Grd(mch)[\{tick\_midnight\}]$
grd2: $pr[\{ms, hs \cdot ms = 59 \land hs = 23 \mid ms \mapsto hs\}] = Grd(mch)[\{tick\_midnight\}]$
grd3: $\{ss, ssp, ms, hs, msp, hsp \cdot$

```
        ss = pr(ms ↦ hs) ∧ ssp = pr(msp ↦ hsp)∧
        msp = 0 ∧ hsp = 0 | ss ↦ ssp}
            = BAP(mch)[{tick_midnight}]
  WITH
    e: e = tick_midnight , s': s' = pr(m' ↦ h')
  THEN
    act1: m, h :|
        pr(m' ↦ h') ∈ BAP(mch)[{tick_midnight}][{pr(m ↦ h)}]
  END
END
```

Listing 20: A shallow instance of the clock machine ($\mathcal{S}.4$)

XI. Extending the EB4EB Reasoning Mechanism (see Fig. 1.(D))

Extensibility is one of the benefits of the meta-theory *EvtBStruc* and *EvtBPO* of Sections IV and V: 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, in a *non-intrusive* way. By non-intrusive, we mean that these development operations do not affect the classical Event-B development as machines are manipulated as instances of the meta-theory *EvtBStruc* and *EvtbPO*. The principle of designing such Event-B machine analyses is described below.

A. Analysis principle: New POs

In our framework, model analysis is defined by introducing a new PO which must fulfil two requirements 1) first this PO shall be reusable and 2) and second, it shall be generated automatically. The first requirement is met by formalising the PO at the meta-theory level as a predicate operator and the second one relies on the exploitation of well-definedness (WD) and Theorems (THM) POs automatically generated.

1) Event-B machine analysis pattern: The definition of a new PO is depicted by the theory pattern of Listing 21. `Theo4PO` theory (see Fig. 1.(D)) imports the meta-theory $EvtBPO$ and introduces a third, optional type parameter T_{Args} possibly needed by the analysis. The PO associated to the analysis is defined by a predicate operator `[PO]_Definition` formalising the PO as a predicate. Then, checking the defined PO, is realised by the `check_Machine[PO]` predicate operator which is well-defined when machine m is well structured and consistent.

```
THEORY Theo4PO IMPORT EvtBPO
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) ∧ check_Machine_Consistency(m)
    direct definition [PO]_Definition(m, args)
END
```

Listing 21: Analyses Theory

Once a PO is defined, its correctness is established following the same approach as the one we set up in section VII to prove the correctness of native proof obligation. Another theory `Theo4POCorrectness` (see Fig 1.(C, D)) is introduced. It relies on trace semantics proposed in Section VI. Listing 22 shows the skeleton of the `Theo4POCorrectness` theory with a correctness theorem to be proved. It states that `[PO]_Definition` implies the expression $PO\_Sepc\_On\_Traces(\ldots)$, on the traces, of the specification of the concerned PO.

```
THEORY Theo4POCorrectness IMPORT EvtBTraces , Theo4PO
TYPE PARAMETERS STATE, EVENT, T_Args
THEOREMS
    thmCorrectnessPO :
        ∀m, tr · m ∈ Machine(STATE, EVENT)∧
        Machine_WellCons(m) ∧ IsATrace(tr, m)∧
        ... ∧ [PO]_Definition(m, args)
            ⇒ PO_Sepc_On_Traces(...)
```

Listing 22: Analyses Correctness

2) Checking PO context pattern: Listing 23 shows the Event-B context pattern defined to check the newly defined PO. A consistent instance machine context $Deep$, associated to an Event-B machine resulting from the instantiation of the meta-theory $EvtBStruc$ and $EvtBPO$, is extended by the context $MachinePO$ instantiating the extended theory $Theo4PO$. Theorem $thmPO$ checks that the PO holds for the machine mch. The WD and THM associated POs are automatically generated.

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

Listing 23: Analyses Machine

The main key points of using this framework is that 1) well-definedness conditions ensure elements are used correctly, 2) meta-properties on these analyses are done once and for all, and 3) these analyses can be performed without altering the machine's behaviour, in a non-intrusive way.

This approach is demonstrated below on the definition of the deadlock freeness proof obligation.

B. Introduction of deadlock-freeness as a new proof obligation

1) Requirements: Deadlock-freeness states that a machine m cannot be in a state where no progress is possible, i.e. at least one event in a machine m is always enabled. Informally, it can be formulated as: when the invariant holds then the disjunction of all the events guards holds.

2) PO Definition: The PO shall state that, for a machine m, there exists at least one event e such that the current state satisfies its guard i.e. $s \in Grd(m)(e)$. When expressed using the operators of the meta-theory, we write $Inv(m) \subseteq Grd(m)[Progress(m)]$. This operator does not require any additional argument for $args$.

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

Listing 24: DeadlockFree Theory

According to the defined pattern, we introduce, in Listing 24, a new theory `Theo4Deadlock`, with two new operators and required well-definedness condition.

3) Deadlock freeness PO for Clock model: The extended context with the $thmDLK$ theorem generating WD and THM POs of the $clock$ machine is shown in Listing 25.

```
CONTEXT ClockDeadlockFree
EXTENDS ClockDeep
THEOREMS
  thmDLK: check_Machine_DeadLock(clock)
END
```

Listing 25: Clock DeadlockFreeness

4) Correctness: We specify the deadlock freeness properties using the Event-B trace semantics described in SectionVII in order to check the correctness of the defined PO, and thus ensure that a developed model is deadlock-free. The deadlock freeness PO is defined as `DeadlockFreeness_Definition` in Listing 24.

```
THEOREMS
  ThmCorrectnessDeadlockFree :
    ∀m, tr · m ∈ Machine(STATE, EVENT)∧
    Machine_WellCons(m) ∧ IsATrace(tr, m)∧
    Mch_INV(m) ∧ DeadlockFreeness_Definition(m)
      ⇒ AllAre(tr, Grd(m)[Progress(m)])
```

Listing 26: Theorem of Deadlock freeness' correctness

Listing 26 presents the theorem `Theo4Deadlock-Correctness` to ensure the correctness of deadlock freeness PO. It states that for all well-constructed machines ($Machine\_WellCons(m)$) and for all traces of any machine such that traces of machines ($IsATrace(tr, m)$) are valid ones, machine invariants fold ($Mch\_INV(m)$), and the proof obligation of deadlock freeness ($DeadlockFreeness\_Definition(m)$) holds, then any state satisfies at least one guard of a progress event at any state of the trace ($AllAre$).

XII. Proof Process

The Rodin platform is well-equipped with different types of provers and SMT solvers to support proof automation for the core Event-B. However, the current proof process for meta models developed in the EB4EB framework lacks automation. Thus, user interaction is needed to discharge the generated proof obligations. One of our primary goals is to experiment with shallow modelling mechanisms to use the native induction proof process in Event-B to reduce overall proof efforts and improve proof automation. On the other hand, we propose some proof rules in the developed theories to define some rewriting rules to simplify the direct definitions and well definedness of the EB4EB framework's defined operators. One of these rewrite rules is based on the deep modelling template given in the Listing 13, where the destructor of the machine instantiated in the proof obligation is replaced by the instance's set comprehension. Then, the definitions of these proof rules are integrated with existing or new proof tactics of Rodin. These rules are automatically applied by the Rodin prover when tactics are invoked.

Model	PO	Max Depth	Nodes	Interactive Nodes	Number of Tactic application
Clock deep	thm1/WD	47	137	1	2
	thm1/THM	108	352	0	1
DeadlockFree	thmDLK/THM	169	221	1	2

TABLE V: Proof statistics

Table V shows the number of automatic nodes for each theorem of the deep modelling. Without any tactic, these nodes are discharged manually for each operator by instantiating correctly. The introduction of new proof rules in form of tactics enables to discharge most of the nodes automatically. For example, most of the proof obligations of the clock model are discharged automatically and only one node requires manual interaction. Similarly, the deadlock freeness has 169 nodes without tactic and only one interactive node with tactic.

XIII. Conclusion

We presented the EB4EB framework, which allows for the explicit manipulation of Event-B features using meta-modelling concepts. The developed framework is a collection of theories that includes data types, operators, WD conditions, theorems, and proof rules. These theories are specially designed for encoding the core modelling constructs and proof obligation rules of Event-B. In addition, trace semantics is provided to ensure the correctness of the introduced Event-B language constructs. We developed two instantiation mechanisms, deep and shallow, to use the defined theories and associated operators, definitions, WD and proof rules. These mechanisms allow manipulation of static and dynamic concepts of Event-B features as well as defining new proof obligations to support advanced level reasoning once and for all. Note that these theories must be instantiated in new development, and the generated POs must be discharged to ensure correct instantiation. The expressiveness, effectiveness, portability, and scalability of our developed EB4EB framework and its trace semantics were evaluated using Lamport's clock case study. Finally, we showed, on the case of the deadlock freeness, that correct extensions can be introduced.

In the future, we intend to use EB4EB framework to extend the reasoning mechanism by supporting externally defined POs to analyse domain-specific properties, such as continuous behaviour, human-machine interactions etc. In addition, we plan to certify Rodin plugins like composition/decomposition, code generation and model transformations and so on, using EB4EB framework. Another important goal is to use Dedukti [13] to import and export the Event-B theory and models into proof assistants such as Coq, PVS and Isabelle/HOL.

Acknowledgements This study was undertaken as part of the EBRP (*Enhancing Event-B and RODIN: EventB-RODIN-Plus*) project funded by ANR, the French National Research Agency.

References

[1] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

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

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, 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] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[7] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Asmetasmv: A way to link high-level ASM models to low-level nusmv specifications. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *2nd International Conference, ABZ*, volume 5977 of *LNCS*, pages 61–74. Springer, 2010.

[8] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic review of abstract state machines by meta property verification. In César A. Muñoz, editor, *2nd NASA Formal Methods Symposium - NFM*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 4–13, 2010.

[9] Paolo Arcaini, Riccardo Melioli, and Elvinia Riccobene. Asmetaf: A flattener for the ASMETA framework. In Paolo Masci, Rosemary Monahan, and Virgile Prevosto, editors, *4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC*, volume 284 of *EPTCS*, pages 26–36, 2018.

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

[11] J. C. Bicarregui and B. Ritchie. Reasoning about vdm developments using the vdm support tool in mural. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods*, pages 371–388. Springer Berlin Heidelberg, 1991.

[12] Jean-Paul Bodeveix and Mamoun Filali. Event-b formalization of event-b contexts. In Alexander Raschke and Dominique Méry, editors, *Rigorous State-Based Methods*, pages 66–80, Cham, 2021. Springer International Publishing.

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

[14] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. AsmetaA: Animator for abstract state machines. In Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *6th International Conference, ABZ*, volume 10817 of *LNCS*, pages 369–373, 2018.

[15] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[16] 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 *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.

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

[18] 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.

[19] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for asms. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *First International Conference, ABZ*, volume 5238 of *LNCS*, pages 71–84. Springer, 2008.

[20] Pierre Castéran. *An Explicit Semantics for Event-B Refinements*, pages 155–173. Springer Singapore, 2021.

[21] 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.

[22] 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.

[23] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. 1(ICFP), 2017.

[24] 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*, volume 9236 of *LNCS*, pages 170–186. Springer, 2015.

[25] Marie Farrell, Rosemary Monahan, and James F. Power. Combining event-b and CSP: an institution theoretic approach to interoperability. In Zhenhua Duan and Luke Ong, editors, *19th International Conference ICFEM*, volume 10610 of *LNCS*, pages 140–156. Springer, 2017.

[26] 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. Springer, 2014.

[27] Thai Son Hoang and Jean-Raymond Abrial. Reasoning about liveness properties in event-b. In Shengchao Qin and Zongyan Qiu, editors, *13th International Conference on Formal Engineering Methods, ICFEM 2011*, volume 6991 of *LNCS*, pages 456–471. Springer, 2011.

[28] Thai Son Hoang, Dana Dghaym, Colin F. Snook, and Michael J. Butler. A composition mechanism for refinement-based methods. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS*, pages 100–109. IEEE Computer Society, 2017.

[29] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[30] Clifford B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[31] Clifford B. Jones, K. D. Jones, Peter Alexander Lindsay, and Richard C. Moore. *Mural - a formal development support system*. Springer, 1991.

[32] Michael Leuschel and Michael Butler. *ProB: A Model Checker for B*, pages 855–874. LNCS. Springer, 2003.

[33] Ismaïl Mendil, Yamine Aït Ameur, Neeraj Kumar Singh, Dominique Méry, and Philippe A. Palanque. Leveraging event-b theories for handling domain knowledge in design models. In Shengchao Qin, Jim Woodcock, and Wenhui Zhang, editors, *7th International Symposium, SETTA*, volume 13071 of *LNCS*, pages 40–58. Springer, 2021.

[34] Ismaïl Mendil, Yamine Aït Ameur, Neeraj Kumar Singh, Dominique Méry, and Philippe A. Palanque. Standard conformance-by-construction with event-b. In Alberto Lluch-Lafuente and Anastasia Mavridou, editors, *26th International Conference, FMICS*, volume 12863 of *LNCS*, pages 126–146. Springer, 2021.

[35] 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.

[36] Sayan Mitra and Myla Archer. 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).

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

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

[39] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction - CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[40] Elvinia Riccobene and Patrizia Scandurra. Towards an interchange language for asms. In Wolf Zimmermann and Bernhard Thalheim, editors, *11th International Workshop, Advances in Theory and Practice ASM*, volume 3052 of *LNCS*, pages 111–126. Springer, 2004.

[41] Peter Rivière, Neeraj Kumar Singh, and Yamine Aït Ameur. EB4EB: A Framework for Reflexive Event-B. In *26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 71–80, 2022.

[42] Patrizia Scandurra, Andrea Arnoldi, Tao Yue, and Marco Dolci. Functional requirements validation by transforming use case models into abstract state machines. In Sascha Ossowski and Paola Lecca, editors, *ACM Symposium SAC*, pages 1063–1068. ACM, 2012.

[43] Steve A. Schneider, Helen Treharne, and Heike Wehrheim. A CSP account of event-b refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *15th International Refinement Workshop, Refine@FM*, volume 55 of *EPTCS*, pages 139–154, 2011.

[44] Steve A. Schneider, Helen Treharne, and Heike Wehrheim. The behavioural semantics of event-b refinement. *Formal Aspects Comput.*, 26(2):251–280, 2014.

[45] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, dec 2002.

[46] 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.

[47] 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.

[48] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016.

[49] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, dec 1997.