Non-Intrusive Annotation-Based Domain-Specific Analysis to Certify Event-B Models Behaviours

Ismail Mendil, Peter Riviere, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry, Philippe Palanque

Non-Intrusive Annotation-Based Domain-Specific Analysis to Certify Event-B Models Behaviours

I. Mendil[1], P. Rivière[1], Y. Aït-Ameur[1], N. K. Singh[1], D. Méry[2], and P. Palanque[3]

[1]INPT-ENSEEIHT/IRIT, University of Toulouse, France
[2]Telecom Nancy, LORIA, Université de Lorraine, France
[3]IRIT, Université de Toulouse, France
{ismail.mendil,peter.riviere,yamine,nsingh}@enseeiht.fr,
dominique.mery@loria.fr, palanque@irit.fr

Abstract. System engineering advocates a thorough understanding of the engineering domain or certification standards (aeronautics, railway, medical, etc.) associated to the system under design. In this context, engineering domain knowledge plays a predominant role in system design and/or certification. Furthermore, it is a prerequisite to achieve the effectiveness and performance of the designed system. This article proposes a formal method for describing and setting up domain-specific behavioural analyses. It defines a formal verification technique for dynamic properties entailed by engineering domain knowledge where Event-B formal models are annotated and analysed in a non-intrusive way, i.e. without destructive alteration. This method is based on the formalisation of behavioural properties analyses relying on domain knowledge as an ontology on the one hand and a meta-theory for Event-B on the other hand. The proposed method is illustrated using a critical interactive system.

Keywords: Domain knowledge · Formal Methods · Ontology · Event-B Theories · Refinement · Proof · Behavioural Analyses.

1 Introduction

Context. System behaviour analysis necessitates handling domain constraints, knowledge and standards together with the use of different logics and in particular temporal logic which allows for expressing requirements related to the system behaviour. Event-B [1], like other formal methods, offers built-in mechanisms like invariant preservation for verifying properties of complex systems models expressed using abstract machines. However, in order to formalise complex properties, in particular behavioural properties, the designer must accommodate the Event-B modelling language constructs especially since these constructs are based on first-order logic and set theory.

While safety properties are explicitly modelled in Event-B thanks to invariants and theorems, the temporal properties related to liveness require a complex operational formalisation. Moreover, handling constraints raised by standards or domain knowledge properties require ad hoc modelling by the designer.

In [11], the authors extend the class of liveness properties for Event-B by defining a list of proof obligations used to express proof rules. In [18,10], a behavioural semantics to Event-B and a collection of conditions allowing for verifying LTL properties across a refinement chain are proposed. Although the approach supports extended LTL operators, the paper does not address the issue of explicitly handling domain knowledge. Several modelling frameworks, including DOL, CASL [14] and RAISE [6,7,8], advocate for explicit domain knowledge in formal modelling where several fields, such as railways systems, shipping, and logistics, are described. Additionally, [4] highlights the benefits of expressing explicitly domain properties and [3] compiled a collection of applications that use explicit domain knowledge in modelling. In [13], we discussed a certification process ensuring that a system model meets the requirements of a standard formalised as an ontology. The approach is constructive, it relies on the annotation of state variables by references to ontology concepts and on a set of operators used to transfer to models domain knowledge formalised as properties.

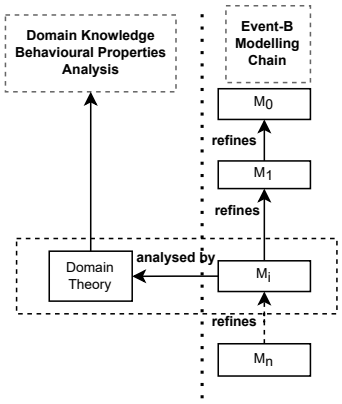

Fig. 1: Methodology overview

In this paper, we describe behavioural analyses mined from domain knowledge. Moreover, the analyses discussed here do not require an *a priori* alteration of the models (non-intrusive approach) but rather the certification of the behavioural model is established using annotations.

Objective of this paper. Our goal is to provide an integrated framework (see Fig. 1) for verifying domain-specific behavioural properties of formal design models. The framework meets an important requirement: non-intrusiveness, achieved by annotation i.e. the model's elements are associated with domain knowledge concepts. This high-level goal is divided into subgoals as follows:

1. Supply a language for expressing domain knowledge concepts and rules. Ontologies are good candidates for this purpose.
2. Lift formal models to a meta-level to manipulate explicitly their constituents.
3. Provide a mechanism for defining analyses combining both domain knowledge and formal modelling concepts
4. Bind system models concepts and domain knowledge concepts and set up the *domain-specific behavioural analysis*.

Organisation of this paper. The sequel of this paper is structured following our solutions to the identified sub-goals. Section 2 overviews the Event-B method which is the backbone of the proposed framework. Section 3 recalls the two formal languages used to address sub-goals (1) and (2) respectively, where the ontology modelling and the meta-Event-B languages are introduced. Section 4 presents the integrated framework, meeting sub-goals (3) and (4), to address

formal model analyses and annotation mechanisms. Section 5 describes the case study illustrating our approach and section 6 defines a specific domain knowledge based analysis according to the methodology devised in this article. Sections 7 draws an assessment and section 8 provides a conclusion and future perspectives.

2 Event-B method

Event-B [1] is a *correct-by-construction* method based on set theory and first-order logic. It supports state-based modelling where a set of events encodes state changes. Proof Obligations (PO) (see Table 2) are automatically generated.

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

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

2.1 Contexts and machines (see Table 1.(b) and 1.(c))

A `Context` describes the static properties of a model: Axioms and theorems describing required concepts using *carrier sets* s, *constants* c, *axioms* A and *theorems* T_{ctx}. A `Machine` describes the model behaviour as a transition system. A set of guarded events is used to modify the state using Before-After Predicates.

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

Table 2: Relevant Proof Obligations for Event-B contexts and machines

Refinements. Refinement decomposes a *machine* into a less abstract one with more design decisions moving from an abstract level to a less abstract one. Gluing invariants relating abstract and concrete variables ensure property preservation.

Core Well-definedness (WD). WD POs are associated with all Event-B operators. Once proved, these WD conditions are used as hypotheses to prove other POs.

2.2 Event-B extensions with Theories (see Table 1.(a))

To handle more complex and abstract concepts beyond set theory and first-order logic, an Event-B extension for externally defined mathematical objects has been proposed in [2,9]. It introduces user data types, operators, theorems and associated rewrite and inference rules, all bundled in so-called *theories* similar to other proof assistants like Coq [5], Isabelle/HOL [15] or PVS [16].

Theories Definition. Theories contain *datatypes* and *operators* that can be used in Event-B expressions as *predicates* or *expressions* producing values. Operators may be defined explicitly in the `OPERATORS` clause, or defined axiomatically in the `AXIOMATIC DEFINITIONS` clause. Last, a theory may provide theorems and proof rules . Many theories have been defined for lists, reals, differential equations, etc.

Well-definedness (WD) in Theories. An important feature provided by Event-B theories is the possibility to define *Well-Definedness* (WD) conditions (close to TCC conditions in PVS [16]). Each defined operator (thus partially defined) is associated with a user-defined condition ensuring its correct definition. When it is applied this WD condition generates a PO that needs to be discharged.

Event-B proof system and Rodin. Rodin[1] is an open source IDE for modelling in Event-B. It provides model editing, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. The Event-B theories extension is available as a plug-in. They tightly integrated in the proof process. Depending on their definition, operator definitions are expanded either using their direct definition or by enriching the hypotheses pool using their axiomatic definition. Theorems may be imported as hypotheses. Many provers for first-order logic as well as SMT solvers are plugged into Rodin.

3 Previous Work

3.1 Ontology Modelling Language as Event-B Theory

The ontology modelling language [12] is used for describing domain-specific behavioural properties. It features a trade-off between the expressive power of first-order logic and the practicality of high-level primitives. It comes as an Event-B theory providing one data type and a collection of operators and theorems.

Listing 1 shows important elements used for modelling an ontology. `OntologiesTheory` is an Event-B theory which is parameterised by `C`, `P` and `I` denoting Classes, Properties and Instances, respectively. This theory defines a constructor `consOntology` with 7 attributes: `classes`, `properties`, `instances`, `classProperties`, `classInstances`, `classAssociations`, `instanceAssociations`.

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

In addition, expression and predicate operators allowing to manipulate the ontology are defined. Operators returning an expression allows computing values based on ontology attributes. Predicate operators are used defining well-definedness conditions and to check logical properties. For example, `getClassInstances` is an operator that allows retrieve the relation class to instances in a *safe way* since the operator is correctly used only if its well-defined condition is discharged. The well-definedness condition of this operator is formalised in the `isWDClassInstances` stating that the classes and instances of the relation correspond respectively to classes and instances of the ontology. Another important operator is `isWDOntology` which checks that an ontology is well defined in the sense that all the attributes obey the individual well-definedness condition. Other operators are provided among them `isA` and `ontologyContainsClasses` where the former checks where a class subsumes another class and the latter verifies whether a collection of classes belongs to a given ontology.

Last, theorems may be derived for the ontology modelling language from the data type definition and the operator specifications. For example, the theorem `isATran` states that `isA` is transitive. It is noteworthy that the theorems are valid provided that the ontology is well defined, the same applies to all the operators which require a well-defined ontology as an argument.

THEORY $OntologiesTheory$
TYPE PARAMETERS C, P, I
DATA TYPES
$\quad Ontology(C, P, I)$
CONSTRUCTORS
$\quad consOntology(classes : \mathbb{P}(C), properties : \mathbb{P}(P), instances : \mathbb{P}(I),$
$\quad\quad classProperties : \mathbb{P}(C \times P), classInstances : \mathbb{P}(C \times I),$
$\quad\quad classAssociations : \mathbb{P}(C \times P \times C), instanceAssociations : \mathbb{P}(I \times P \times I))$
OPERATORS
isWDClassInstances <*predicate*> ...
getClassInstances <*expression*> ...
isWDOntology <*predicate*> $(o : Ontology(C, P, I))$
$\quad$ **direct definition**
$\quad\quad isWDClassProperites(o) \land isWDClassInstances(o) \land$
$\quad\quad isWDClassAssociations(o) \land isWDInstancesAssociations(o)$
ontologyContainsClasses <*predicate*> ...
isA <*predicate*> $(o : Ontology(C, P, I), c1 : C, c2 : C)$
$\quad$ **well−definedness** $isWDOntology(o), ontologyContainsClasses(o, c1, c2)$
$\quad$ **direct definition**
$\quad\quad getInstancesOfaClass(o, c1) \subseteq getInstancesOfaClass(o, c2)$
...
THEOREMS
isATrans:
$\quad \forall o, c1, c2, c3 \cdot o \in Ontology(C, P, I) \land isWDOntology(o) \land$
$\quad\quad c1 \in C \land c2 \in C \land c3 \in C \land ontologyContainsClasses(o, c1, c2, c3)$
$\quad\quad\quad \Rightarrow (isA(o, c1, c2) \land isA(o, c2, c3) \Rightarrow isA(o, c1, c3))$

Listing 1: Ontology modelling language Event-B theory

3.2 The Event-B Meta-theory

For enhancing the reasoning support of Event-B, a reflexive framework has been defined in [17]. To access Event-B components as first-class elements and keep the semantics, and propose a reasoning mechanism expressed with the meta-level.

Listing 2 shows the data type representing the machine's elements, which are parameterised by two types: `Ev` and `St`. A constructor is defined `Cons_machine` where each argument corresponds to a machine component. The machine denotes a state transition system on the set of states (`State`) constrained by the invariant (`Inv`).

```
THEORY EvtBTheo
TYPE PARAMETERS St, Ev
DATATYPES Machine(St, Ev)
 CONSTRUCTORS
    Cons_machine(
       Event : ℙ(Ev),
       State : ℙ(St),
       Init : Ev, Progress : ℙ(Ev)
       Variant : ℙ(St × ℤ),
       AP : ℙ(St),
       BAP : ℙ(Ev × (St × St)),
       Grd : ℙ(Ev × St),
       Inv : ℙ(St),
       ...)
```

Listing 2: Machine Data type

Machine structure. Events are triggered by the initialisation event (`Init`) then by progress events (`Progress`). State changes are provoked by the After predicate (`AP`) for `Init`, and the Before After Predicate (`BAP`) for progress events if their corresponding guards (`Grd`) are true. A numeric variant (`Variant`) is defined, it is used for liveness properties.

```
Event_WellCons <predicate> (m : Machine(St, Ev))
  direct definition partition(Event(m), {Init(m)}, Progress(m))
  ...
Machine_WellCons <predicate> (m : Machine(St, Ev))
  direct definition Event_WellCons(m) ∧ ...
```

Listing 3: Operators to check well-defined data type (static semantics)

Well-Constructed machines. The data type requires to formalise the constraints on the constructor's arguments. For example `Event_WellCons` (see Listing 3) encodes the property stating that events are partitioned as initialisation event and progress events and `Machine_WellCons` defines well constructed machines (not detailed here because of space limitations reasons).

Machine POs (Semantics of Event-B machines). The proof obligations are formalised based on the semantics of guarded transitions systems. Each proof obligation is formalised using set theory. Predicates over state variables are modelled as sets of states satisfying the predicate and logical connectives are formalised by operations on sets.

```
Mch_THM <predicate> ...
Mch_INV_Init <predicate> (m : Machine(St, Ev))
  direct definition  AP(m) ⊆ Inv(m)
Mch_INV_One_Ev <predicate> (m : Machine(St, Ev), e : Ev)
  well−definedness e ∈ Progress(m)
  direct definition BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
Mch_INV <predicate> (m : Machine(St, Ev))
  direct definition
    Mch_INV_Init(m) ∧ (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
Mch_FIS_Init <predicate> ...
Mch_FIS_One_Ev <predicate> ...
Mch_FIS <predicate> ...
Mch_VARIANT_One_Ev <predicate> ...
Mch_VARIANT <predicate> ...
Mch_NAT_One_Ev <predicate> ...
Mch_NAT <predicate> ...
```

Listing 4: Well-defined data type operators (behavioural semantics)

For example, Listing 4 describes the induction principle for verifying the invariant PO where `Mch_INV_Init` predicate states that the initialisation event must establish the invariant ($AP(m) \subseteq Inv(m)$) and `Mch_INV_One_Ev` states that a given progress event e must preserve the invariant ($BAP(m)[\{e\}]\,[Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$). Last, the `Inv` PO (see. Table 2) is formalised by the `Mch_INV` operator as the conjunction of the two previous operators. Likewise, all POs are formalised using the same transformation principle.

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

Listing 5: Operator for Event-B machine consistency

Last, the operator `check_Machine_Consistency` of Listing 5 is the conjunction of all the predicates formalising the various POs. It formalises an Event-B machine correctness condition.

When this predicate is used as a `theorem` in an Event-B system development then the core POs (see Table 2) as well as the well-definedness POs are automatically generated by the Rodin platform. Discharging all the generated POs along with this theorem ensures the consistency of the machine.

Instantiation of the meta-theory. The defined meta-theory is instantiated to define specific Event-B machines. Instantiation consists in defining an Event-B context with instances for the type parameters `St` and `Ev` and providing instances for the attributes of `Cons_machine`.

4 Domain-Specific Behavioural Analysis

Our proposal consists in describing domain knowledge properties, particularly behavioural ones, as generic properties expressed on Event-B model concepts. Such properties usually occur in domain requirements or standards.

The proposed framework (see Fig. 2) is composed of two basic blocks: ontology modelling language (see Section 3.1) and meta-Event-B language (see Section 3.2). The first component provides primitives to write domain concepts and constraints as ontologies, while the second component allows for the abstraction of a system as an instance of the meta-Event-B language, allowing for a reasoning extension. Moreover, all the behavioural properties encoded in first-order logic can be written in our framework and validated on models using our methodology. In addition to purely temporal properties, the framework allows expressing enriched analyses that take the domain knowledge into account. A mechanism for referencing domain knowledge in design models is also defined. The framework is composed of three parts, and the resulting step-by-step methodology for analysing some Event-B models is divided into four major steps.

4.1 Components of the methodology

Three main components have been identified: Event-B development Fig. 2.(A), theories Fig. 2.(B) and instances as Event-B contexts Fig. 2.(C).

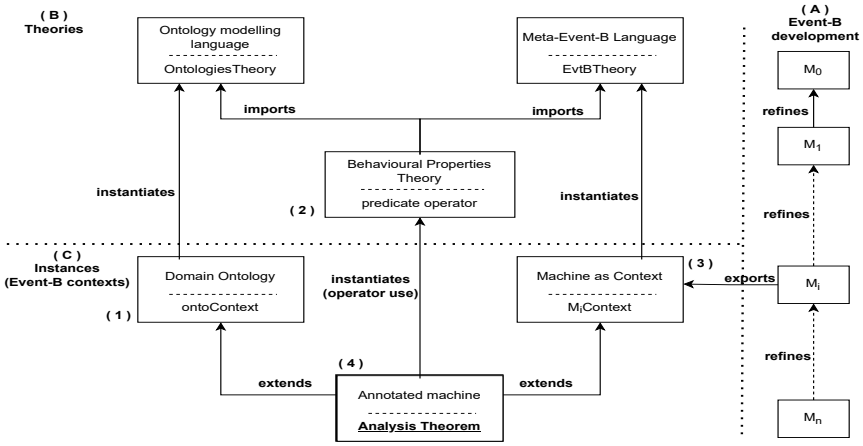

Fig. 2: Event-B-based framework for domain behavioural properties analysis

(A) Event-B development represents the Event-B model development, including the refinement chain for the system design with an abstract machine $\texttt{M}_i$to be analysed;

(B) Theories enabling a designer to manipulate and analyse Event-B models. Three theories have been introduced as follows.

- `EvtBTheory` Event-B theory, introduced in section 3.2, allowing to formalise an Event-B model as a context and to define proof obligations;
- An ontology modelling language (`OntologiesTheory` in Listing 1 of Section 3.1) for describing domain knowledge concepts and constraints.
- A theory importing both theories in order to 1) annotate Event-B models with ontological concepts and to 2) define new proof obligations as predicate operators expressing specific behavioural domain properties and used as theorems on specific Event-B models expressed as instances.

(C) Instances (Event-B contexts). The third part describes the contexts instantiating the theories of Part (B). The context (4) describes the annotated model, which extends the ontology context, the Event-B model context, and defines theorems corresponding to the behavioural properties to be checked. The proof of these theorems guarantees that the properties hold on the analysed machine $\texttt{M}_i$ of part (A).

4.2 A Methodology for defining Event-B models domain knowledge based analyses

The methodology we propose follows four steps (see Fig. 2).

Step 1: *Ontology definition.* This step consists in describing the domain ontology with the domain concepts and properties as an instance, represented by an Event-B context, of the theory presented in Section 3.1.

Step 2: *Express a domain-specific behavioural analysis.* It consists in defining a predicate based on an ontology and applying it to an Event-B model

Step 3: *Export the Event-B machine.* Export the Event-B machine as an instance of the theory *EvtBTheory* to be analysed. The produced instance allows the machine elements to be explicitly manipulated using the operators of Section 3.2 theory.

Step 4: *Annotate the Event-B machine.* Annotate the Event-B machine with the ontology concepts from Step 1. Machine concepts (variables and events) are linked to ontology concepts (tags). The annotated machine is an instance of the theory defining the behavioural properties with a theorem, formalising this property requirement by a theorem to be proved.

The remainder of the article demonstrates the approach using Event-B method.

5 Case Study

5.1 Informal Description

A critical interactive system is modelled to demonstrate our approach: the user interface of an automatic teller machine (ATM), where the primary requirement is that an authenticated client withdraws banknotes safely. Below, we present important requirements associated with HMI only.

REQ-1 A user can exclusively use a keyboard or a screen.

REQ-2 To withdraw banknotes, a user must be authenticated.

REQ-3 A user can adjust the brightness a finite number of times.

REQ-4 Any entered passcode must be followed by a confirmation.

First, a user inserts a credit card and chooses an input device to enter a passcode. The user must confirm the entered passcode. Before performing this operation, the user may adjust the brightness of the screen. When the user confirms the input, validation starts. It may result in the acceptance or refusal of the passcode. If the passcode is correct, the ATM delivers banknotes and ejects the card. Otherwise, the user may try again to enter the correct passcode. A user makes new attempts a limited and fixed number of times only.

5.2 Formal Description in Event-B

This section presents the formal development of the ATM user interface corresponding to Fig. 2.(A). It consists of a context and a machine defining static dynamic properties, respectively. Listing 6 shows relevant types and constants for modelling the ATM interface. In `axm1` and `axm2`, two enumerated types, `IPT_MOD` and `IRT_STS`, are defined to select possible input devices (keyboard,

CONTEXT $ATMEnvironment$
SETS IPT_MOD, IRT_STS, STR
CONSTANTS $MAX\_ATP, CRT\_KYW, KBD,$
$SCR, IN, OUT, BRT\_LVS, BRT\_MIN, B$
$RT\_MAX, EPT\_STR, MAX\_BRT\_UPD$
AXIOMS
axm1: $partition(IPT\_MOD, \{KBD\}, \{SCR\})$
axm2: $partition(IRT\_STS, \{IN\}, \{OUT\})$
axm3: $MAX\_ATP \in \mathbb{N}1$
axm4: $CRT\_KYW \in STR$
axm5: $EPT\_STR \in STR$
axm6: $CRT\_KYW \neq EPT\_STR$
axm7–8: $BRT\_MIN \in \mathbb{N} \land BRT\_MAX \in \mathbb{N}$
axm9: $BRT\_MAX > BRT\_MIN$
axm10: $BRT\_LVS = BRT\_MIN..BRT\_MAX$
axm11: $MAX\_BRT\_UPD \in \mathbb{N}$
END

Listing 6: Context of the ATM

screen) and credit card modes (in, out), respectively. `MAX_ATP` represents the maximum number of attempts. Contextual information for managing the brightness limit and brightness levels are defined using axioms (`axm7-axm11`). Moreover `axm4` to `axm6` defines the string type with two important constants: `EPT_STR` for the empty string and `CRT_STR` representing the correct password.

Listing 7 shows an extract of the machine where 14 variables and safety properties are introduced to formalize ATM interactions. Several Events are introduced; `chnBrt` is defined to cover **REQ-3**, for adjusting the brightness of the screen. The guard of `chnBrt` ensures that the maximum number of updates is not exceeded.

Other requirements are checked by invariant proving like `inv16`: the entered password is never displayed. **REQ-4** embeds a different kind of properties since (1) it references domain knowledge concepts (2) it is a behavioural property. Due to space limitation, we show only important variables and safety properties.

MACHINE ATMUserInterface
SEES ATMEnvironment
VARIABLES str, $scrReg$, $kbdReg$, atp, $cnfSts$, $valSts$, $dlvSts$, $isStrVis$, $iptMod$, $crdSts$, brt, $brtUpd$, $cnfKBDstr$, $cnfSCRstr$
INVARIANTS
inv1−14: ...
inv15: $str = scrReg \lor str = kbdReg$
inv16: $isStrVis = \bot$
EVENTS
INITIALISATION...
etrKBDStr
WHERE
grd1: $0 \leq atp \land atp < MAX\_ATP$
grd2−3: $iptMod = KBD \land crdSts = IN$
grd4: $cnfKBDstr = \bot$
THEN
$act1 : str, kbdReg :| kbdReg' \in STR \land str' = kbdReg'$
$act2 : brtUpd, cnfKBDstr := 0, \top$
END

chnBrt
WHERE
grd1: $crdSts = IN$
grd2: $brtUpd < MAX\_BRT\_UPD$
THEN
act1: $brt :\in BRT\_LVS$
act2: $brtUpd := brtUpd + 1$
END
cnfKBDStr
WHERE
grd1: $0 \leq atp \land atp < MAX\_ATP$
grd2−3: $iptMod = KBD \land crdSts = IN$
THEN
act1: $atp := atp + 1$
act2: $cnfSts := \top$
act3: $cnfKBDstr := \bot$
END
...

Listing 7: ATM machine

6 Our methodology at work

The methodology's first step (see Section 4) is to define an ontology of events that will serve as the basis for modelling domain analyses. Our main objective is to check if the ATM model follows **REQ-4** stating that when an event annotated as input is triggered, a confirmation event must be triggerable in the future.

6.1 Step 1 - Event Ontology Instantiation (Fig. 2.(1))

The ontology modelling language (see. Section 3.1) is used to describe Event tags. `input`, `confirmation`, and `finite` are particularly relevant in the ATM case study. The first two tags are used to denote interaction events that provide user input information and formalise a user response. Finally, `finite` designates events that does not occur indefinitely. Listing 8 (corresponding to `ontoContext`

of Fig. 2.(1)) contains the instantiation of the ontology modelling theory. It provides 3 type parameters: `tags` for ontology classes, `Ps` for tag properties, and instances for model events. The other ontology contents are not provided here as they are not relevant for our development.

```
CONTEXT  EventTagOntology
SETS  Tags, Ps, Ev
CONSTANTS  eventOntology, tag, input, confirmation, ..., finite
AXIOMS
  axm1: partition(Tags, {tag}, {input}, {confirmation}, ..., {finite})
  axm2: eventOntology ∈ Ontology(Tags, Ps, Ev)
  axm3–4: classes(eventOntology) = Tags ∧ instances(eventOntology) = Ev
  ...
END
```

Listing 8: Context for event ontology instantiation

6.2 Step 2 - Behaviour Analysis definition (Fig. 2.(2))

The definition of the the analysis is composed of 2 phases. First, the terms defining the analysis are specified and then the predicate formed by the conjunction of these conditions is parameterised by the domain ontology. This subsection is divided accordingly and starts with the latter phase.

Domain-Specific Analysis Operator Definition. The theory for expressing behavioural properties is presented in Listing 9. It corresponds to the `predicate operator` in Fig. 2.(2). For example, **REQ-4** is formalised using two operators, `isNecFollowedByWD` defining the WD condition of the second one defining the property analysis `isNecFollowedBy` . Indeed, `isNecFollowedBy` verifies whether events annotated by tags in `srcTg` are always followed by events annotated by tags in `trgTg` passing through the intermediate events annotated with `internalTg`. It asserts that each `EvtInst` event annotated as `srcTg` is reachable in the sense of `is_Reachable` predicate operator.

```
THEORY BehaviouralPropertiesTheory
IMPORT THEORY Theo4Reachability, OntologiesTheory
TYPE PARAMETERS St, Ev, Tg, Prop
OPERATORS
isNecFollowedByWD <predicate> (m : Mach(St, Ev), eo : Ontology(Tg, Prop, Ev),
                    srcTg : ℙ(Tg), internalTg : ℙ(Tg), trgTg : ℙ(Tg), v : ℙ(E × ℙ(St × ℤ)))
  direct definition
    isWDOntology(eo) ∧ srcTg ∪ internalTg ∪ trgTg ⊆ cls(eo)∧
    srcTg ≠ ∅ ∧ trgTg ≠ ∅ ∧ internalTg ≠ ∅∧
    srcTg ∩ internalTg = ∅ ∧ trgTg ∩ internalTg = ∅∧
    (∀ti · ti ∈ srcTg ∪ internalTg ∪ trgTg ⇒ insOfC(eo, ti) ≠ ∅)∧
    v ∈ insOfC(eo, srcTg) → ℙ(St × ℤ)∧
    (∀i, t · i ∈ insOfC(eo, srcTg) ∧ t ∈ insOfC(eo, trgTg) ⇒
        WD_reach(m, i, insOfC(eo, trgTg), insOfC(eo, internalTg), v(i)))
isNecFollowedBy <predicate> (m : Mach(St, E), eo : Ontology(Tg, Prop, Ev),
                srcTg : ℙ(Tg), internalTg : ℙ(Tg), trgTg : ℙ(Tg), v : ℙ(Ev × ℙ(St × ℤ)))
  well–definedness isNecFollowedByWD(m, eo, srcTg, internalTg, trgTg, v)
  direct definition
    ∀EvtInst · EvtInst ∈ insOfC(eo, srcTg) ⇒
                Is_Reachable(m, EvtInst, insOfC(eo, trgTg),
                    insOfC(eo, internalTg), v(EvtInst))
END
```

Listing 9: Domain Behavioural Properties Theory

This operator has 6 arguments: `m` - machine to be analysed, `eo` - ontology to represent the domain concepts and constraints, `srcTg` - source tags, `internalTg` - transit tags, `trgTg` - target tags, and `v` - a list of variants. Note that `insOfC` returns events annotated by tags. To ensure the correct application of this operator, we provide a WD condition: `isNecFollowedByWD` predicate. This operator has six arguments similar to the previous operator. The direct definition of this operator ensures the well-defined ontology ($isWDOntology$), reachability conditions (WD_reach) and satisfies the given variant for each departing event.

Analysis Terms Definition Listing 10, based on `EvtBTheory` in Fig. 2.(2), shows reachability theory where several operators are defined for analysing the reachability properties. `WD_reach` operator is defined and its direct definition ensures that the machine m is well constructed (`Machine_WellCons`), the machine invariants are preserved (`Mch_INV`), the target event is not the initialisation event (`Init(m)`), the variant is defined for all reachable states.

Given a source event `src`, a set of intermediary events `es`, and a target event `trg`, `Is_Reachable(m, src, trg, es, v)` holds if, when `src` is activated, then some intermediate events in `es` may be observed finitely many times before `trg` is activated.

```
THEORY Theo4Reachability      IMPORT THEORY EvtBTheory
TYPE PARAMETERS St, Ev
OPERATORS
next_states <expression> (m : M(St, Ev)) direct definition ...
WD_reach <predicate> (m : M(St, Ev), src : Ev, trg : ℙ, es : ℙ(Ev), v : ℙ(St × ℤ))
  direct definition
    Machine_WellCons(m) ∧ trg ⊆ Progress(m) ∧ src ∈ Event(m)∧
    Inv(m) ◁ v ∈ Inv(m) → ℤ ∧ Mch_INV(m) ∧ es ⊆ Progress(m)
Init_Local_Inv <predicate> (m : M(St, Ev), src : Ev, lInv : ℙ(St)) ...
Local_Inv_Preserved <predicate> (m : M(St, Ev), initE : Ev, evs : ℙ(Ev), lInv : ℙ(St))
    ...
No_Exit <predicate> (m : M(St, Ev), yesE : ℙ(Ev), noE : ℙ(Ev), trg : ℙ(Ev), v : ℙ(St × ℤ))
    ...
TGMch_VARIANT <predicate> (m : M(St, Ev), v : ℙ(St × ℤ), es : ℙ(Ev)) ...
TGMch_NAT <predicate> (m : M(St, Ev), v : ℙ(St × ℤ), es : ℙ(Ev)) ...
Is_Reachable <predicate> (m : M(St, Ev), src : Ev, trg : Ev, es : ℙ(Ev), v : ℙ(St × ℤ))
  well−definedness WD_reach(m, src, trg, es, v)
  direct definition
    Init_Local_Inv(m, src, Grd(m)[trg])∧
    Local_Inv_Preserved(m, src, es, Grd(m)[trg])∧
    No_Exit(m, es, Progress(m) \ (es ∪ src ∪ trg), trg, v)
    TGMch_NAT(m, v, es) ∧ TGMch_VARIANT(m, v, es)∧
END
```

Listing 10: Reachability Theory

The definition of `Is_Reachable` operator states that first, the `src` event must imply either the guards of the events in `es` or the guard of the events in `trg` (`Init_Local_Inv`). Second, the local invariant `Local_Inv_Preserved` must be preserved by intermediary events. Note that the set `Grd(m)[{trg}]` defined by the local invariant guarantees that the guard of the `trg` always holds. Then, the definition excludes the events not in `es` (`No_Exit` operator). Last, the intermediary events `es` must not be indefinitely active (variants operator `TGMch_NAT` and `TGMch_VARIANT`).

The analysis is strong since it requires that the events imply the guards of the target events. Without a loss of generality, we can apply this analysis to systems not satisfying this condition, indeed by refinement we can address systems that reach the wanted states after many steps; the refinement of the intermediary events does not break the analysis.

6.3 Step 3 - Exporting Event-B models as instances of the Meta-Event-B theory (Fig. 2.(3))

Each Event-B machine can be formalised as an instance of the Event-B meta-theory (Machine $\mathtt{M}_i$ on Fig. 2.(A)) . To define an Event-B machine as an instance, it is enough to instantiate (give values) the `Machine(St, Ev)` attributes at instantiation (see Listing 2). The `St` type parameter is substituted by a Cartesian product of the set types of `ATMUserInterface` machine state variables (14 in total) and `Ev` by the set of the events of this machine.

Listing 11, corresponding to $\mathtt{M}_i$`Context` in Fig. 2.(3), shows an extract of the `ATMUserInterface` machine exported as an instance of the meta-Event-B theory. Guards and actions of the events are formalised, as instances, in the `Grd(ATM)` and `BAP(ATM)` sets (axioms `axm4` and `axm5`).

CONTEXT $ATMmEBModel$
EXTENDS $ATMEnvironment, EventTagOntology$
CONSTANTS $ATM, init, istCrd, KBDStr, SCRStr, chnBrt, cnfKBDStr, cnfSCRStr, chkStrCrt, chkStrWrg, dlvBnkNts$
AXIOMS
axm1: $partition(Ev, \{KBDStr\}, \{chnBrt\}, \{cnfKBDStr\}, \ldots, \{chkStrWrg\}, \{dlvBnkNts\})$
axm2: $ATM \in Machine(STR \times STR \times STR \times \mathbb{Z} \times BOOL \times BOOL \times BOOL \times BOOL \times IPT\_MOD \times IRT\_STS \times \mathbb{Z} \times \mathbb{Z} \times BOOL \times AMT, Ev)$
axm3: $Event(ATM) = Ev$
axm4: $Grd(ATM) = \{e \mapsto (str \mapsto scrReg \mapsto kbdReg \mapsto atp \mapsto cnfSts \mapsto valSts \mapsto dlvSts \mapsto isStrVis \mapsto iptMod \mapsto crdSts \mapsto brt \mapsto brtUpd \mapsto nStr \mapsto sum) \mid$
$(e = istCrd \land crdSts = OUT) \lor$
$(e = KBDStr \land 0 \leq atp \land atp < MAX\_ATP \land iptMod = KBD \land crdSts = IN \land nStr = \bot) \lor$
$(e = chnBrt \land brtUpd \leq MAX\_BRT\_UPD \land crdSts = IN) \lor$
$(e = cnfKBDStr \land 0 \leq atp \land atp < MAX\_ATP \land iptMod = KBD \land crdSts = IN \land cnfSts = \bot \land valSts = \bot \land nStr = \bot) \lor \ldots\}$
axm5: $BAP(ATM) = \{e \mapsto ((str \mapsto \ldots. \mapsto sum) \mapsto (strp \mapsto \ldots \mapsto sump)) \mid$
$(e = KBDStr \land kbdRegp \in STR \land strp = kbdRegp \land brtUpdp = 0 \land nStrp = \top \land scrReg \mapsto atp \mapsto \ldots \mapsto sum = scrRegp \mapsto atpp \mapsto \ldots \mapsto sump) \lor$
$(e = cnfKBDStr \land atpp = atp + 1 \land cnfStsp = \top \land str \mapsto \ldots \mapsto sum = strp \mapsto \ldots \mapsto sump) \lor$
$(e = chnBrt \land brtp \in BRT\_LVS \land brtUpdp = brtUpd + 1 \land str \mapsto \ldots sum = strp \mapsto \ldots sump) \lor \ldots\}$
THEOREMS
thm: $check\_Machine\_Consistency(ATM)$
END

Listing 11: Annotation and analysis context

6.4 Step 4 - Annotation & analysis (Fig. 2.(4))

The final step before checking the property is *annotation*. It links the domain knowledge concepts and constraints to the design model. The events are assigned to tags satisfying the subsumption relation. For example, the `cnfKBDStr` is assigned to `textualConfirmation`, `confirmation` and `Tag`.

The model events are annotated using the `annotationDef` relation. They are related to the `eventOntology` via `classInstances`. Indeed, the events are instances of the ontology classes of tags. In Listing 12, which materialises `Analysis Theorem` in Fig. 2.(4), there are 3 main theorems `isWDOntologyThm`, `vThm` and `anaTh`. The first proves that the `eventOntology` is well-defined as described in Section 3.1. The second is important to establish the WD condition of the analysis operator; it ensures that variants are supplied for all events annotated with `input`. The last theorem is the most important, it ensures the correctness of the analysis performed by discharging the generated proof obligations associated to theorems. The proof of application of the predicate operator `isNecFollowedBy` states that the requirement **REQ-4** is satisfied for the `ATMUserInterface`.

CONTEXT $AnnotatedMachine$
EXTENDS $EventTagOntology, ATMmEBModel$
CONSTANTS $annotationDef, variantsDef$
AXIOMS
$\mathbf{axm1}: annotationDef = (\{bounded\} \times \{chnBrt\}) \cup (\{inputByKeyboard\} \times \{KBDStr\}) \cup \ldots \cup$
$(\{input\} \times \{KBDStr, SCRStr\}) \cup (\{textualConfirmation\} \times \{cnfKBDStr\}) \cup$
$(\{confirmation\} \times \{cnfSCRStr, cnfKBDStr\}) \cup$
$(\{interaction\} \times \{KBDStr, SCRStr, cnfKBDStr, cnfSCRStr\}) \cup (\{tag\} \times Ev$
$\mathbf{axm2}: classInstances(eventOntology) = annotationDef$
$\mathbf{axm3}: variantsDef = \{KBDStr \mapsto \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v$
$\mid p \mapsto bright \mapsto ck \mapsto cs \in State(ATM) \land v = MAX\_BRT\_UPD - bright\}\} \cup$
$\{SCRStr \mapsto \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v$
$\mid p \mapsto bright \mapsto ck \mapsto cs \in State(ATM) \land v = MAX\_BRT\_UPD - bright\}\}$
THEOREMS
$\mathbf{isWDOntologyThm}: isWDOntology(eventOntology)$
$\mathbf{vThm}: variantsDef \in annotationDef[\{input\}] \rightarrow \mathbb{P}(State(ATM) \times \mathbb{Z})$
$\mathbf{anaThm}: \mathbf{isNecFollowedBy(ATM, eventOntology,}$
$\mathbf{\{input\}, \{bounded\}, \{confirmation, abortion\}, variantsDef)}$
END

Listing 12: Annotation and analysis context

7 Assessment

This section discusses the evaluation of the framework described in section 4 corroborated by the observations of section 6. All the models are available at `https://www.irit.fr/~Ismail.Mendil/recherches`.

Principled Methodology vs. Ad hoc Analysis. The prime objective of the methodology is to define a set of principles that can be used for describing domain-specific behavioural analyses. A direct approach would be to follow a shallow paradigm analysis, which consists of incorporating the analysis elements into the model. The methodology provides two modules: (1) the ontology modelling language which allows defining domain knowledge as an ontology, and (2) the Meta-Event-B modelling language which provides a handle to reason on Event-B concepts. Furthermore, refinement is used when possible to overcome the definition of complex analyses (see. section 6.2).

Domain-Specific Analyses and Reusability & Sharability. The ability to parameterise the behavioural analyses with domain-specific constraints and concepts is a significant aspect of the analyses discussed in this article. This enables precise analyses based on concepts and rules drawn from a domain

knowledge description. This was the case for the analysis presented in this article, which is based on an event ontology. The framework architecture (see Fig. 2) improves the reusability and shareability of the analysis.

The Methodology Is Non-Intrusive. Integrating domain knowledge concepts and constraints directly into the system model could be one approach to investigating the behaviour properties. Such an approach suffers from a lack of generalisation and is intrusive when modelling a system is mixed with analysis details and it may not be scalable. The approach presented in this article allows to avoid these difficulties; indeed, the methodology uses the model as an argument, and the annotation is not intrusive.

Proof-Based Verification. Another approach would be to boil down the domain knowledge behaviour properties into temporal properties and then use model checking to verify the resulting set of temporal properties. The model is passed as an argument to the analysis procedure alongside the domain knowledge model, which has the advantage of being non-intrusive. Yet, this method meets quickly its limitation when the systems are large and complex due to the classical problem of state explosion. Furthermore, manual translation of domain knowledge constraints is fastidious. The methodology proposed in this article goes beyond this limitation thanks to the proof-based verification. Indeed, the analysis is established by proving a predicate operator where variant and invariant proof obligations are discharged.

Proof & Modelling Effort Reduction. The methodology proposed in this article reduces proof effort by factorising out common parts such as the proof of the well-definedness analysis and the definition of a collection of lemmas useful at the model side. In addition, the framework components are described once and for all, and they are only proved before being deployed and used for multiple system models. All the proofs performed on the theory side (see Fig. 2.(A)) are achived once and for all. They are reused at the machine instance level.

8 Conclusion

In this article, we addressed the issue of analysing domain-specific behavioural properties over formal models of systems. The proposal begins by identifying several intermediate subgoals for achieving the main goal and consists of an integrated framework and methodology centred around the Event-B method for investigating non-intrusively behavioural properties mined from domain knowledge. Several challenges are identified: (1) formalising domain knowledge, (2) accessing and manipulating Event-B concepts (3) defining domain-specific behavioural analyses, and (4) annotating and analysing Event-B models. Solutions for all subgoals are proposed in the Event-B setting, indeed (1) domain knowledge is formalised using an ontology modelling language; (2) meta-Event-B is used as a handle for expressing properties on Event-B machines; (3) a methodology for formalising analyses based on the two former theories is presented; and (4) annotating Event-B models for analysis purposes. The methodology is illustrated through a concrete analysis applied to a real-world case study.

In future work, we plan to apply the framework to other case studies and generalise the ontology of events to include associations between events. In addition, we believe that the approach can be exploited for certification purposes. Indeed, a non-intrusive analysis may be carried out for such purposes if certification standards are formalised as theories formalising certification properties.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B. Tech. rep. (2009)
3. Aït Ameur, Y., Nakajima, S., Méry, D.: Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems. Springer (2021)
4. Aït Ameur, Y., Méry, D.: Making explicit domain knowledge in formal system development. Sci. Comput. Program. **121**, 100–127 (2016)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions (2010)
6. Bjørner, D.: Software Engineering 3 - Domains, Requirements, and Software Design. Texts in Theoretical Computer Science. An EATCS Series, Springer (2006)
7. Bjørner, D.: Manifest domains: analysis and description. Formal Aspects Comput. **29**(2), 175–225 (2017)
8. Bjørner, D.: Domain analysis and description principles, techniques, and modelling languages. ACM Trans. Softw. Eng. Methodol. **28**(2), 8:1–8:67 (2019)
9. Butler, M., Maamria, I.: Practical Theory Extension in Event-B, pp. 67–81. Springer (2013)
10. Hoang, S., Schneider, S.A., Treharne, H., Williams, D.M.: Foundations for using linear temporal logic in event-b refinement. Formal Aspects of Computing **28**, 909–935 (2016)
11. Hoang, T.S., Abrial, J.: Reasoning about liveness properties in event-b. In: Qin, S., Qiu, Z. (eds.) 13th International Conference ICFEM. LNCS, vol. 6991, pp. 456–471. Springer (2011)
12. Mendil, I., Aït Ameur, Y., Singh, N.K., Méry, D., Palanque, P.A.: Leveraging event-b theories for handling domain knowledge in design models. In: Qin, S., Woodcock, J., Zhang, W. (eds.) 7th International Symposium, SETTA. LNCS, vol. 13071, pp. 40–58. Springer (2021)
13. Mendil, I., Aït Ameur, Y., Singh, N.K., Méry, D., Palanque, P.A.: Standard conformance-by-construction with event-b. In: Lluch-Lafuente, A., Mavridou, A. (eds.) 26th International Conference, FMICS. LNCS, vol. 12863, pp. 126–146. Springer (2021)
14. Mossakowski, T.: The Distributed Ontology, Model and Specification Language - DOL, LNCS, vol. 10644, pp. 5–10. Springer (2016)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
16. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference CADE. LNAI, vol. 607, pp. 748–752. Springer-Verlag (1992)
17. Rivière, P., Singh, N.K., Aït Ameur, Y.: EB4EB: A Framework for Reflexive Event-B. In: 26th International conference ICECCS). pp. 71–80 (2022)
18. Schneider, S.A., Treharne, H., Wehrheim, H.: The behavioural semantics of event-b refinement. Formal Aspects of Computing **26**, 251–280 (2012)