Deliverable T2.3-D2.2 (T2.3)
Annotating event-based models

June 1, 2017:3:14 P.M.

Project IMPEX

ANR-13-INSE-0001

IRIT, ENSEEIHT, Université de Toulouse
LABRI, Université de Bordeaux & CNRS
LORIA, Telecom Nancy, Université de Lorraine
LRI, CentraleSupelec
SAMOVAR, Telecom Sud Paris
Systerel, Aix-en-Provence
http://impex.gforge.inria.fr
16/12/2013-17/12/2017

Contents

List of Figures

1 Introduction

Critical systems are running in heterogeneous domains. This heterogeneity is rarely considered explicitly, when describing and validating processes. Handling explicitly such domain knowledge increases design models robustness due to the expression and validation of new properties mined from the domain models. The document summarizes the observations from case studies with respect to the question of *improving state-based models by annotation*s. Annotations are a way to *help* the proof tools when proving and to complete either static informations in context or dynamic informations in invariants for instance. We are reporting observations and proposals based on case studies leading to *annotations* as a key concept for improving the proof-based design of state-based models especially in Event-B.

The annotation process emerges while developing case studies, as for instance the Nose Gear Velocity case study reported in [48] where we have made a first important step towards the seamless integration of explicit semantic features into the formal development in Event-B. To explicitly define domain knowledge, we abstractly specified all the required entities (data types) and relationships between them in model's contexts. However, as per the main properties of domain descriptions, these descriptions should be *generic, extendable,* and *reusable*. Therefore, the next important step is to develop an independent Event-B component – *ontology/domain description library* – that will contain formal definitions of data types and functions for *all* (not only fundamental) units of physical quantities related to the *Nose Gear (NG) velocity* development (i.e., velocity, distance and time) in two most widely used systems of measurement (i.e., metric and imperial). Unlike the definitions presented in contexts of the *Nose Gear (NG) velocity* development, these new definitions will be fully axiomatised using standard physical metrics, which will allow us to guarantee that each definition in the library is sound and complete. Also, to make our definitions more flexible/usable we aim at redefining some of them as instances of the real numbers using the theory plugin. We do not include the works presented in [48] and refer the reader to the paper itself. The archive of the case study are available on request.

Two papers [9, 33] are reporting the stepwise methodology which is leading to enrich design models describing complex information systems with domain knowledge. They use ontologies to model such domain knowledge. Design models are annotated by references to domain ontologies. The resulting annotated models are checked. It becomes possible to verify domain-related properties and obtain strengthened models. The approach is deployed for two design model development approaches: a Model Driven Engineering (MDE) approach and a correct by construction formal modeling one based on refinement and proof using Event-B method.

The current document report works that are done also in the task 1 and there are some common experiments reported in [37, 38]. The IMPEX methodology requires the formalisation of properties in an ontological environment as presented in [37, 38]. We have also evaluated the OntoEventB plug-in [39] but there are still experiments required to provide better feedbacks and it will be a challenge for the task 3 and task 4 too.

Next section is providing a short introduction to the Event-B modelling language; in a way Section 2 can be omitted but completes the document. Section 3 is reporting simple case studies which are explaining simple examples of developed Event-B models related to simple case studies. Section 4 is synthesizing the emergent IMPEX methodology at the end of the task 2. Section 5 is concluding the document and adding perspectives for the task 3 which is replaying case studies using the IMPEX methodology.

2 Design Models: the Event-B Modeling Language

Event-B [4, 6] has evolved from Classical B [1] for specifying and reasoning about reactive systems. Main motivation to select Event-B is targeted at an incremental modelling style [10, 12] where a system is defined abstractly, and later more properties are introduced in an incremental fashion with a stepwise introduction of safety properties. The use of refinement represents systems at different levels of abstraction and the use of mathematical proof verifies consistency between refinement levels. Event-B is an event-based approach which is defined in terms of a few simple concepts describing a discrete event system and proof obligations that permit verification of properties of the event system. This chapter explains the fundamental concepts and formal notations of Event-B modelling language. Event-B is provided with tool support in the form of an open and extensible Eclipse-based IDE called RODIN [6] which is a platform for Event-B specification and verification.

2.1 Overview of B

Classical B is a state-based method developed by Abrial for specifying, designing and coding software systems. It is based on Zermelo-Fraenkel set theory with the axiom of choice. Sets are used for data modelling, *Generalised Substitutions* are used to describe state modifications, the refinement calculus is used to relate models at varying levels of abstraction, and there are a number of structuring mechanisms (machine, refinement, implementation), which are used in the organisation of a development. The first version of the B method is extensively described in The B-Book [1]. It is supported by the Atelier B tool [29] and by the B Toolkit [43].

Central to the classical B approach is the idea of a software operation which will perform according to a given specification if called within a given pre-condition. Subsequent to the formulation of the classical approach, Abrial and others have developed a more general approach in which the notion of *event* is fundamental. An event has a firing condition (a guard) as opposed to a pre-condition. It may fire when its guard is true. Event based models have proved useful in requirement analysis, modelling distributed systems and in the discovery/design of both distributed and sequential programming algorithms.

After an extensive experience with B, current work by Abrial is proposing the formulation of a second version of the method [3]. This distills experience gained with the event-based approach and provides a general framework for the development of *discrete systems*. Although this widens the scope of the method, the mathematical foundations of both versions of the method are the same.

2.2 Proof-based Development

Proof-based development methods [12, 1, 50] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [12, 1, 26, 11]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination.

A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [29, 30].

At the most abstract level it is obligatory to describe the static properties of a model's data by means of an *invariant* predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations.

The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements.

B Models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [28]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques [49].

2.3 Scope of the B Modelling

The scope of the B method concerns the complete process of software and system development. Initially, the B method was mainly restricted to the development of software systems [13, 41, 36] but a wider scope for the method has emerged with the incorporation of the event based approach [2, 7, 3, 23, 22, 51] and is related to the systematic derivation of reactive distributed systems. Events are simply expressed in the rich syntax of the B language. Abrial and Mussat [7] introduce elements to handle liveness properties. The refinement of the event-based B method does not deal with fairness constraints but introduces explicit counters to ensure the happening of abstract events, while new events are introduced in a refined model. Among case studies developed in B, we can mention the METEOR project [13] for controlling train traffic, the PCI protocol [24], the IEEE 1394 Tree Identify Protocol [5]. Finally, B has been combined with CSP for handling communications systems [22, 21] and with action systems [23, 51]. The proposal can be compared to action systems [10], UNITY programs [26] and TLA [40] specifications but there is no notion of abstract fairness like in TLA or in UNITY.

2.4 The Event-B Modelling Notation

Event-B [4], unlike Classical B [1], does not have a fixed syntax. We summarize the concepts of the Event-B modeling language [25, 4] developed by Abrial and indicate the links with the tool called RODIN [6]. Here, we present the basic notation for Event-B using some syntax. We proceed like this to improve legibility and help the reader remembering the different constructs of Event-B. The syntax should be understood as a convention for presenting Event-B models in a textual form rather than defining a language.

Event-B [4] modeling language has mainly two main constructs *contexts* and *machines*, which are used to model the whole system. Contexts is used to formalise the static parts of the system, while Machines is used to specify dynamic behavior of the system. Context can be extended by other contexts and referenced by machines. A dynamic part of the system, machines can be refined by machines. Fig. 1 depicts basic constructs and their relationship.

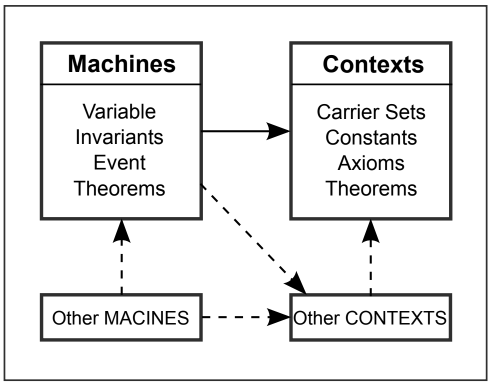

Figure 1: Relationship between constructs: Machine and Contexts

2.4.1 Contexts

Contexts express axiomatic static properties of the models. Contexts may contain carrier sets, constants, axioms, and theorems. Axioms describe properties of carrier sets and constants. Theorems are derived properties that can be proved from the axioms. Proof obligations associated with contexts are straightforward: the stated theorems must be proved, which follow from the predefined axioms and theorems. Additionally, a context may be indirectly seen by machines. Namely, a context C can be seen by a machine M indirectly if the machine M explicitly sees a context which is an extension of the context C.

2.4.2 Machines

Machines express dynamic behavioural properties of the models, which may contain variables, invariants, theorems, events, and variants. Variables v represents the state of the machine. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etcetera. They are constrained by invariants $I(v)$. Invariants are supposed to hold whenever variable values change.

A machine is organizing events modifying state variables, and it uses static informations defined in a context. These basic structure mechanisms are extended by the refinement mechanism which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows us to develop gradually Event-B models and to validate each decision step using the proof tool. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P is simulating M. The final concrete model is close to the behavior of the real system that is executing events using real source code. We give details now on the definition of events, refinement and guidelines for developing complex system models.

2.4.3 Modeling actions over states

The event-driven approach [25, 4] is based on the Classical B notation [1]. It extends the methodological scope of basic concepts to take into account the idea of *formal reactive models*. Briefly, a formal reactive model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*, where an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. Table-1 presents some set-theoretical notations of Event-B [4] and Classical B [1], which are used to formalise the system. We summarize the definitions and principles of formal models and explain how they can be managed by tools [6].

Name	**Syntax**	**Definition**
Binary relation	$s \leftrightarrow t$	$(\mathcal{P})(s \times t)$
Composition of relations	$r_1 \; ; r_2$	$\{x, y \mid x \in a \ \wedge \ y \in b \wedge$
		$\exists z.(z \in c \ \wedge \ x, z \in r_1 \ \wedge \ z, y \in r_2\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \ \wedge \ y \in \mathcal{D}(b) \ \wedge \ a \mapsto b \in r)\}$
Domain	$dom(r)$	$\{a \mid a \in s \ \wedge \ \exists b.(b \in t \ \wedge \ a \mapsto b \in r)\}$
Range	$ran(r)$	$dom(r^{-1})$
Identity	$id(s)$	$\{x, y \mid x \in s \ \wedge \ y \in s \ \wedge \ x = y\}$
Restriction	$s \lhd r$	$id(s); r$
Co-restriction	$r \rhd s$	$r; id(s)$
Anti-restriction	$s ⩤ r$	$(dom(r) - s) \lhd r$
Anti-co-restriction	$r ⩥ s$	$r \rhd (ran(r) - s)$
Image	$r[w]$	$ran(w \lhd r)$
Overriding	$q \mathbin{\lhd\!\!\!-} r$	(dom$(r) \lhd q) \cup r$
Partial function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge \ (r^{-1}; r) \subseteq id(t)\}$

Table 1: Set-theoretical notation

Each event is composed of a guard $G(t, x)$ and an action $R(t, x)$, where t are local variables the event may contain. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. The first general form for an event is as follows:

ANY t **WHERE** $G(x, t)$ **THEN** $x : |(R(x, x', t)$ **END**

Second form of an event (e), when event (e) has not any local variable (t), then an event is represented as follows:

WHEN $G(x)$ **THEN** $x : |(R(x, x')$ **END**

Third form of an event (e), when event (e) has not any guard (G) and local variable (t), then an event is represented as follows:

BEGIN $x : |(R(x, x')$ **END**

The first form for an event means that it is guarded by a guard that states the necessary condition for this event to occur. The guard is represented by $\exists\, t \cdot G(t, x)$. It defines a possibly nondeterministic event where t represents a vector of distinct local variables. It is also semantically equivalent to $\exists\, t \cdot\, (G(t, x) \ \wedge \ R(x, x', t))$. In the first, second and third forms, *before-after* predicate $BA(e)(x, x')$, associated with each event e, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the *execution* of event e. The third form of the event (e) is used for initialisation.

Generalized substitutions are also borrowed from the B notation [1]. They provide a means to express changes to state variable values. The action of an event is composed of mainly three kinds of assignments

: $skip$ (do nothing), deterministic assignment and non-deterministic assignment. Where x is a variable, E is an expression and P is a predicate. The value of x in each case depends on its corresponding expression/predicate. For example, $x :\in E(x,t)$, x will be assigned as an element of $E(x,t)$. In the case of $x : |(P(x,x')$, x will be assigned as a value satisfying the predicate P. $x : |(P(x,x')$ is a more general substitution form of an assignment predicate. This should be read as *x is modified in such a way that the value of x afterwards, denoted by x', satisfies the predicate* $P(x,x')$, where x' denotes the *new value* of the vector and x denotes its *old value*.

Type	**Generalized Substitution**
Empty	$skip$
Deterministic	$x := E(x,t)$
Non-deterministic	$x :\in E(x,t)$
	$x : |(P(x,x')$

Table 2: List of Generalized Substitutions

2.4.4 Proof Obligations

Proof obligations are generated by Rodin tool [6]. Different kinds of proof obligations are produced by Rodin tools, which are as follows: WD (well- definedness), INV (Invariant Preservation), GRD (Guard Strengthening), SIM (Action Simulation), FIS (Feasibility), etcetera. WD proof obligations are generated to ensure that formal predicates and expressions are well defined, which covers generally axioms, invariants, event guards/actions. The Rodin tool supports well-definedness to aid the activities of modelling and proving [1]. INV proof obligations are generated to guarantee that the invariants are always preserved whenever the machine state changes. Proof obligations (INV 1 and INV 2) are produced by the RODIN tool [6] from events to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before-after predicate $BA(e)(x,x')$ of each event e and $grd(e)(x)$ is safety of the guard $G(t,x)$ of event e:

(INV1) $Init(x) \Rightarrow I(x)$
(INV2) $I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$

The generated GRD proof obligation ensures that the guard of a concrete event is a correct refinement of the corresponding guard of the abstract event. Finally, the generated SIM proof obligations aim to ensure that the abstract actions are refined correctly by the action of the corresponding concrete event as specified by any gluing invariants. Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. Whenever this is the case, the event is said to be *disabled*.

The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I. By proving feasibility we achieve that $BA(e)(x,y)$ provides an after state whenever $grd(e)(x)$ holds. This means that the guard indeed represents the enabling condition of the event. The intention of specifying a guard of an event is that the event may always occur when the guard is true. There is, however, some interaction between guards and nondeterministic assignments, namely $x : |BA(e)(x,x')$. The predicate $BA(e)(x,x')$ of an action $x : |BA(e)(x,x')$ is not satisfiable or the set S of an action $v :\in S$ is empty. Both cases show violations of the event feasibility proof obligation.

$$\text{(FIS) } I(x) \wedge \mathsf{grd}(e)(x) \Rightarrow \exists y.BA(e)(x,y)$$

We say that an assignment is feasible if there is an after-state satisfying the corresponding before-after predicate. For each event its feasibility must be proved. Note, that for deterministic assignments, the proof of feasibility is trivial. Furthermore, note that feasibility of the initialisation of a machine yields the existence of an initial state of the machine. It is not necessary to require an extra initialisation.

It is sometimes useful to state that the model which has been defined is deadlock free, that it can run for ever. This is very simply done by stating that the disjunction of the event guards always hold under the properties of the constant and the invariant. This is shown as follows, where $\mathsf{G}_1(e)(x), ..., \mathsf{G}_n(e)(x)$ denote the guards of the events.

$$\text{(DKLF) } I(x) \Rightarrow \mathsf{G}_1(e)(x) \vee \mathsf{G}_2(e)(x), ..., \mathsf{G}_n(e)(x)$$

2.4.5 Model refinement

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our correct-by-construction approach [42]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a set of possible concrete version, and by adding new events. The abstract (x) and concrete (y) state variables are linked by means of a *gluing invariant* $J(x, y)$. A number of proof obligations ensure that,

- each abstract event is correctly refined by its corresponding concrete version
- each new event refines $skip$
- no new event takes control for ever
- relative deadlock freedom is preserved.

Details of the formulation of these proofs follow. We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. Event e is in abstract model AM and event f is in concrete model CM. Event f refines event e. $BA(e)(x, x')$ and $BA(f)(y, y')$ are predicates of events e and f respectively, we have to prove the following statement, corresponding to proof obligation (1):

$$\boxed{I(x) \ \wedge\ J(x,y) \ \wedge\ BA(f)(y,y') \ \Rightarrow\ \exists x' \cdot (BA(e)(x,x') \ \wedge\ J(x',y'))}$$

The new events introduced in a refinement step can be viewed as hidden events not visible to the environment of a system and are thus outside the control of the environment. In Event-B, requiring a new event to refine $skip$ means that the effect of the new event is not observable in the abstract model. Any number of executions of an internal action may occur in between each execution of a visible action. Now, proof obligation (2) states that $BA(f)(y, y')$ must refine $skip$ ($x' = x$), generating the following simple statement to prove (2):

$$\boxed{I(x) \ \wedge\ J(x,y) \ \wedge\ BA(f)(y,y') \ \Rightarrow\ J(x,y')}$$

In refining a model, an existing event can be refined by strengthening the guard and/or the before–after predicate (effectively reducing the degree of nondeterminism), or a new event can be added to refine the skip event. The feasibility condition is crucial to avoiding possible states that have no successor, such as division by zero. Furthermore, this refinement guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model. The refinement of an event e by an event f means that the event f simulates the event e.

The Event-B modeling language is supported by the RODIN platform [6] and has been introduced in publications [4, 1, 25], where there are many case studies and discussions about the language itself and the foundations of the Event-B approach. The language of *generalized substitutions* is very rich, enabling the expression of any relation between states in a set-theoretical context. The expressive power of the language leads to a requirement for help in writing relational specifications, which is why we should provide guidelines for assisting the development of Event-B models.

2.5 Contexts and machines for Event-B models

The Event-B modelling language is the main modelling language used for our works. It means that we are experimenting and developing case studies using Event-B as well as the platform Rodin which is providing plugins and the possibility to integrate new plugins.

Event-B is a formal modeling language for expressing state-based models of reactive systems. It is supported by two proof-assistant-based environments, namely Rodin [6] and Atelier-B [29]. The two environments can be used for developing the same models; both environments use the same kernel proof module and any Event-B model developed in one of these environments can be exported in the other environment.

The Rodin platform is an open toolset platform, which is used for developing, analysing, validating and experimenting the Event-B methodology. Atelier-B is freely distributed but remains an industrial tool. Moreover, Atelier-B has first provided functionalities for building classical B models, which are models for developing only software. The construction of an Event-B model is based on concepts like sets, constants, axioms, theorems, variables, invariants, events; these syntactic constructions are organised in two kinds of structures, namely contexts and machines. Fig.2 contains the general form of each possible component. Contexts express axiomatic static properties of the models and machines express dynamic behavioral properties (state-based features) of the models, which may contain variables, invariants, theorems and events. A machine $\mathcal{M}$ *sees* a context $\mathcal{D}$.

Contexts may contain carrier sets, constants, axioms, and theorems. Axioms describe properties of carrier sets and constants. Theorems derive properties that can be proved from the axioms. Proof obligations associated with contexts are straightforward: the stated theorems must be proved, which follow from the predefined axioms and theorems. Additionally, a context C can be seen by a machine M indirectly if the machine M explicitly sees a context D which is an extension of the context C. Variables v represents the state of the machine M. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etcetera. They are constrained by invariants $I(v)$. Invariants are supposed to hold whenever variable values change.

A machine is organizing events modifying the state variables and it uses static informations defined in a context. These basic structure mechanisms are extended by the refinement mechanism which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows us to develop gradually Event-B models and to validate each decision step using the proof tools. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P is simulating M. The final concrete model is close to the behavior of real system that is executing events using real source code. We give details now on the definition of events, refinement and guidelines for developing complex system models.

The consistency of a context or a machine in Event-B is achieved by proving *proof obligations* generated by tools [29, 6] and sound with respect to the results of the previous section. If these proof obligations are discharged, then the structure (context or machine) is correct at least with respect to the typing.

An Event-B model organizes a set of *events* stating how state-variables may be modified, when observing the occurence of one of them. Intuitively, an event is triggered when guard (the triggering condition of an event) evaluates to true. Due to non determinism, if a given guard evaluates to true does not mean that the corresponding event is triggered. Indeed, several events may have guards evaluating to true and only one of them is triggered (interleaving of events).

Each event can be defined by a relationship before-after denoted as $BA(x, x')$. An event is characterized by its guard which is determined at the modeling phase and it can only be triggered if the guard is true. We will detail proof obligations generated for a given event e and explain the meaning of these proof obligations. For each event e, proof obligations are generated and discharged by the environment Rodin [6]. These two concepts allows us to illustrate the relationship between *ontologies* and *formal models* and what may be the gain of ontologies in the formal modeling process. The modeling process deals with various languages, as seen by considering the triptych of Bjørner [15, 17, 18, 19]: $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$. Here, the domain $\mathcal{D}$ deals with properties, axioms, sets, constants, functions, relations, and theories and it is written as a context enriched by ontological informations. The system model $\mathcal{S}$ expresses a model or a refinement-based chain of models of the system. Finally, $\mathcal{R}$ expresses requirements for the system to be designed. Considering the Event-B modeling language, we notice that the language can express *safety* properties, which are either *invariants* or *theorems* in a machine corresponding to the system.

A context (see Fig. 2) provides the definition of the sets, constants, axioms for sets and constants, and theorems that can be derived from the axioms of the context $\mathcal{D}$. The context $\mathcal{AD}$ is a previous context that has already been defined, and it is extended to the current context . A context is validated when sets $S_1, \ldots, S_n$, constants $C_1, \ldots, C_m$, and axioms $ax_1, \ldots, ax_p$ are well formed and when all theorems $th_1, \ldots, th_q$ are proved.

A context clearly states the static properties of the (system) model under construction. The *extends* construct enables re-use by extending a previously defined context.

The proof process is based on the management of sequents, with an associated environment for proof called $\Gamma(\mathcal{D})$. The proof environment includes axioms, properties, and theorems already proved. An environment is initially provided, but the intention is to add new theorems. This means that we intend to prove the following properties in the sequent calculus style:

```
CONTEXT $\mathcal{D}$
EXTENDS  $\mathcal{AD}$
SETS
  $S_1, \dots S_n$
CONSTANTS
  $C_1, \dots, C_m$
AXIOMS
  $ax_1 : P_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $ax_p : P_p(S_1, \dots S_n, C_1, \dots, C_m)$
THEOREMS
  $th_1 : Q_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $th_q : Q_q(S_1, \dots S_n, C_1, \dots, C_m)$
```

```
MACHINE $\mathcal{M}$
REFINES  $\mathcal{AM}$
SEES  $\mathcal{D}$
VARIABLES  $x$
INVARIANTS
  $inv_1 : I_1(x, S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $inv_r : I_r(x, S_1, \dots S_n, C_1, \dots, C_m)$
THEOREMS
  $th_1 : SAFE_1(x, S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $th_s : SAFE_s(x, S_1, \dots S_n, C_1, \dots, C_m)$
EVENTS
  EVENT initialisation
    BEGIN
      $x : |(P(x'))$
    END
  $\dots$
  EVENT e
    ANY $t$
    WHERE
      $G(x, t)$
    THEN
      $x : |(P(x, x', t))$
    END
  $\dots$
END
```

Figure 2: Context and Machine

$$\text{for any } j \text{ in } \{1..q\}, \Gamma(\mathcal{D}) \vdash th_j : Q_j(S_1, \dots S_n, C_1, \dots, C_m).$$

Theorems for the context are proved using the Rodin tool, but it is clear that the process for constructing the domain $\mathcal{D}$ is crucial to modeling the system, from consideration of the triptych of Bjørner [15, 17, 18, 19] and variations of this methodology.

The possibility of re-using former definitions is crucial, but we do not consider this point in this paper. Instead, we *simulate* the re-use of theories by manipulating the contexts directly. Among the requirements, we can list the theorems of the context, and we can, in fact, interpret the triptych as follows: for any j in $\{1..q\}$, $\mathcal{D} \longrightarrow th_j : Q_j(S_1, \dots S_n, C_1, \dots, C_m)$. Here, it appears that the system is not mentioned, and this is the case for static properties. Therefore, we have an interpretation of the triptych for the static information, which can be re-used later for any system.

The dynamic part of a model is expressed using a *machine* (see Fig. 2). A machine is either a basic machine or a refinement of an abstract machine. A machine models a state via a list of variables x that are assumed to be modifiable by events listed in the machine. The view is assumed to be closed with respect to events. Each event maintains an assertion called an *invariant*, which is a conjunction of logical statements called inv_j. Each reached state satisfies properties of the theorem part called safety properties. Proof obligations are given in the last section, and they are generated and checkable by the RODIN framework. The validation of the machine M leads to the validation of the safety and invariance properties.

We can obtain a variation of the triptych ($\Gamma(\mathcal{D}, M)$ is an associated environment for proof) as follows.

- For any j in $\{1..r\}$,
  $\Gamma(\mathcal{D}, M) \vdash INITIALISATION(x') \Rightarrow I_j(x', S_1, \dots S_n, C_1, \dots, C_m)$
- For any j in $\{1..r\}$, for any event e of M,

$$\Gamma(\mathcal{D}, M) \vdash \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \dots S_n, C_1, \dots, C_m) \right) \wedge BA(e)(x, x') \Rightarrow I_j(x', S_1, \dots S_n, C_1, \dots, C_m)$$

- For any k in $\{1..s\}$,
$$\Gamma(\mathcal{D}, M) \vdash \left(\bigwedge_{j\in\{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \Rightarrow SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)$$

We use the temporal operator $\Box P$ to express the safety and invariant properties. This operator expresses that the property P is true in all the system states [45, 46].

- For any j in $\{1..r\}$, $\mathcal{D}, M \longrightarrow \Box I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.
- For any k in $\{1..s\}$, $\mathcal{D}, M \longrightarrow \Box SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.

We summarize the requirements expressed by the machine M as follows.

$$\mathcal{D}, M \longrightarrow \Box \left(\begin{array}{l} \left(\bigwedge_{j\in\{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \\ \left(\bigwedge_{k\in\{1..s\}} SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \end{array} \right)$$

We will use the notation $\mathcal{I}(M)$ to stand for the invariants of the machine M and $\mathcal{SAFE}(M)$ to stand for the safety properties of the machine M. We have shown that requirements $\mathcal{R}$ are first expressed using the *always* temporal operator. In next subsections, we illustrate how ontological informations can be used for enriching contexts as well as machines following the refinement-based process. We present two simple examples which are illustrating benefits for formal modeling when using ontologies.

2.6 The OntoEventB plugin

The report [39] describes the OntoEventB Eclipse/Rodin plug-in that implements the proposed approaches. This plug-in has been developed to automatically support the formalisation of ontologies, described with ontology description languages like OWL, using set theory and predicate logic supported by the Event-B method. The plugin completes the already existing tools of the Rodin platform and is a key point for implementing the IMPEX methodology in the task 3 from the works of the task 2.

3 Learning from examples

The annotation concept is emerging from a list of case studies and we are summarizing some case studies to illustrate how the idea can be implemented.

3.1 Motivating the annotation concept using a simple example from program verification in EVENT-B

A simple example is an annotated algorithm called *Example* and setting a value y to a value. The *explicit* knowledge of the designer of the algorithm is allowing him to state that the final value of y will be B. The *explicit* knowledge is required for infering the correctness of the algorithm which is transformed into an EVENT-Bmachine listing the verification conditions to check derived from Floyd's annotations. Two knowledges are missing in the context in the first attempt to discharge the proof obligations generated by the environment: *an1* and *an2*:

- *an1* is added for inferring that there exists at least one value in S different from A.
- *an2* is added for stating that there is no other value different from A in S.

an1 and *an2* are two explicit knowledges of the designer of the algorithm. The prover is requiring new assumptions that are given by the designer, who is the expert of the problem.

```
ALGORITHM $Example$
CONSTANTS $x$
VARIABLES $y$
PRE $x \in S \land x = A \land y \in S$
POST $y \in S \land y = B$
BEGIN
$\ell_1 : \{x = A\}$
$y : |(y' \in S \land y' \neq x)$;
$\ell_2 : \{y = B\}$
END
```

```
CONTEXT $C1$
SETS $S, L$
CONSTANTS $l1, l2, A, B, x$
AXIOMS
  $axm1 : partition(L, \{l1\}, \{l2\})$
  $axm2 : A \in S$
  $axm3 : B \in S$
  $axm4 : x \in S$
  $axm5 : x = A$
  $an1 : A \neq B$ annotation1
  $an2 : S \subseteq \{A, B\}$ annotation2
END
```

```
MACHINE $M1$
SEES $C1$
VARIABLES $y, pc$
INVARIANTS
  $inv1 : pc \in L \land y \in S$
  $inv2 : pc = l1 \Rightarrow y \in S$ Floyd's annotation
  $inv3 : pc = l2 \Rightarrow y = B$ Floyd's annotation
  $th1 : x \in S \land x = A \land y \in S \Rightarrow y \in S$ checking precondition
  $th2 : y = B \Rightarrow y \in S \land y = B$ checking postcondition
EVENTS
EVENT INITIALISATION
  BEGIN
    $act1 : y :\in S$
    $act2 : pc := l1$
  END
EVENT e
  WHEN
    $grd1 : pc = l1$
  THEN
    $act1 : pc := l2$
    $act2 : y : |(y' \in S \land y' \neq x)$
  END
END
```

When considering the statement of the triptych of Dines Bjoerner [16], we have the following relationship in the equation 1. It states that under the proof obligations generated and proved by the Rodin environment, the algorithm is partially correct. The explicit knowledges of the designer are encoded by the prover as annotations which are axioms. Why do we use axioms? In fact, the explicit knowledge is trusted by the prover and annotations are derived from expertise.

$$C1, M1 \longrightarrow (\text{PRE } x \in S \land x = A \land y \in S, \text{POST } y \in S \land x = B) \qquad (1)$$

3.2 Second example: a simple avionic system

Let us consider a simple system issued from avionic system design. We identify two sub-systems: the first one is part of the flight management system acting in the closed world (heart of the avionic systems), it produces flight informations, like *altitude* and *speed*; and the second is the display part of a passenger information system (open world). It displays, to the passengers, information issued from the closed world, here altitude and speed. The information is transmitted from the closed world to the open world within a communication bus. Communications are *unidirectional from the closed world to the open world only*.

The development of this system considers a formally expressed specification which is refined twice. Figure 3 shows the structure of the development for this case study, and the whole Event-B developments are given in appendix **??**.

The development, presented below, introduces the explicit knowledge carried out by ontologies, it is used for coding the ternary relationship called triptych. In Event-B, it is formalized within *contexts*. The

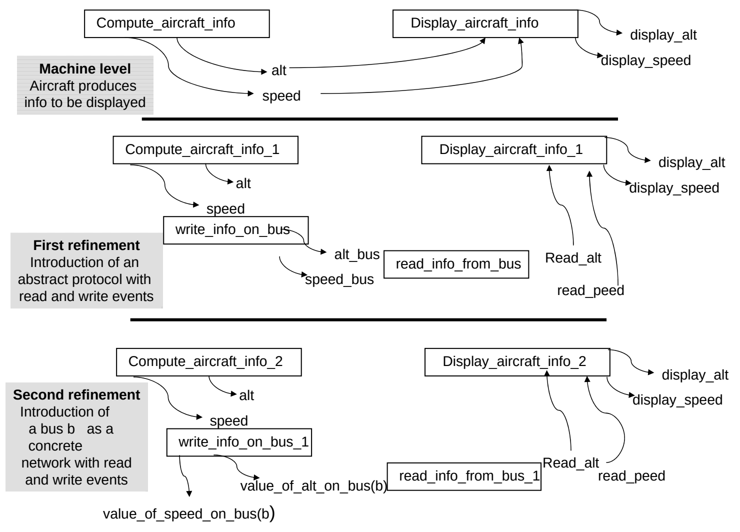

Figure 3: A global view of the formal development.

ternary relationship is obtained by *annotation* i.e. linking the model elements, variables in our case, to the explicit knowledge.

3.2.1 Ontologies: Contexts for Defining Explicit Domain Knowledge

The first step consists of introducing the explicit domain knowledge through a formal model for ontologies. In the simple case, this knowledge is defined by contexts. In this case, we are concerned by the description of the units that may be associated to the $altitude$ and to the $speed$ in the context DOMAIN_KNOWLEDGE_FOR_UNITS.

Meters, inches, kilometers per hour, and miles per hour are introduced to define distance speed measures. Conversion functions, that define equivalences in terms of ontology definitions, are described by the functions $inchtometer$ and $mphtokph$. We do not detail these two functions but they can be made more precise by an implementation step at a later phase in the process.

```
CONTEXT $DOMAIN\_KNOWLEDGE\_FOR\_UNITS$
SETS
  $STATUS, inch, meter, mph, kph$
CONSTANTS
  $OK, KO, inchtometer, mphtokph, valt, vspeed$
AXIOMS
  $axm1 : partition(STATUS, \{OK\}, \{KO\})$
  $onto - axm2 : inchtometer \in inch \rightarrow meter$
  $onto - axm3 : mphtokph \in mph \rightarrow kph$
  $axm4 : valt \in inch$
  $axm5 : vspeed \in mph$
END
```

Figure 4: The ontological context for units

3.2.2 Annotation: Associating Explicit Knowledge to Model Variables.

Once the explicit knowledge has been formalized, it becomes possible to annotate available concepts in the obtained formal models. In our case, the variables are annotated by explicitly referring to the ontology defined in the context of figure 4. Measurement units are introduced in an explicit way.

The variables are then defined as follows:

$$\begin{array}{l} inv1 : speed \in mph \\ inv2 : alt \in inch \\ inv4 : display\_speed \in kph \\ inv5 : display\_alt \in meter \end{array}$$

When the annotations have been specified, the new invariant defines the ontological constraints that should be satisfied by the events. For example, one of the generated proof obligations for checking the preservation of $inv5 : display\_alt \in meter$ by the event Display_Aircraft_Info fails to prove that $alt \in meter$. Thus, we should modify the event Display_Aircraft_Info by adding the ontological information provided by the two functions $inchtometer$ and $mphtokph$. The example is simple and gives an obvious way to solve the possible unproved proof obligation: without refinement it may be much more difficult to discover why similar proof obligations are not discharged. We think that the refinement is one of the key issues for *distilling* requirements and ontological information. This method highlights the interest of handling explicit knowledge for checking model correctness.

Consequently, the following events — Display_Air craft_Info and Compute_aircraft_info — require further description. In Particular, Display_aircraft_info has been modified in order to handle converted values issued from Compute_aircraft_info.

```
EVENT Compute_Aircraft_Info
  WHEN
    grd1 : computing = KO
  THEN
    act1 : alt :∈ inch
    act2 : speed :∈ mph
    act3 : computing := OK
  END
```

```
EVENT Display_Aircraft_Info
  WHEN
    grd1 : computing = OK
  THEN
    act1 : computing := KO
    act2 : display_speed := mphtokph(speed)
    act3 : display_alt := inchtometer(alt)
  END
```

3.2.3 First refinement: Introducing an Abstract Communication Protocol

As a next step, we can add new features in the current model COM1 by refining it into COM10.

The new model COM10 performs the same extension of the state as in the previous case using implicit knowledge. This is quite natural since none of these state variables (i.e. $step$) are annotated). Two new events model the reading to and the writing from the bus. The invariant is extended by sub-invariants $inv3 \ldots inv9$. Notice the introduction of new kinds of invariants, labeled $onto - inv7$, $onto - inv8$ and $onto - inv9$, borrowed from the context where the explicit knowledge is described. They define **ontological invariants**.

$$\begin{array}{l} inv3 : alt\_bus \in meter \\ inv4 : speed\_bus \in kph \\ inv5 : read\_speed \in kph \\ inv6 : read\_alt \in meter \\ onto - inv7 : step = written \Rightarrow alt\_bus = inchtometer(alt) \land speed\_bus = mphtokph(speed) \\ onto - inv8 : step = read \Rightarrow \left(\begin{array}{l} read\_alt = alt\_bus \land alt\_bus = inchtometer(alt) \\ \land read\_speed = speed\_bus \land speed\_bus = mphtokph(speed) \end{array} \right) \\ onto - inv9 : step = tocompute \Rightarrow \left(\begin{array}{l} display\_alt = read\_alt \land read\_alt = alt\_bus \\ \land alt\_bus = inchtometer(alt) \land display\_speed = read\_speed \\ \land read\_speed = speed\_bus \land speed\_bus = mphtokph(speed) \end{array} \right) \end{array}$$

The two abstract events are refined by strengthening guards with respect to the new control variable ($step$). The new model introduces an abstract protocol for the bus.

```
EVENT Compute_Aircraft_Info_1
REFINES Compute_Aircraft_Info
WHEN
  $grd1 : computing = KO$
  $grd2 : step = tocompute$
THEN
  $act1 : alt :\in inch$
  $act2 : speed :\in mph$
  $act3 : computing := OK$
  $act4 : step := computed$
END
```

```
EVENT Display_Aircraft_Info_1
REFINES Display_Aircraft_Info
WHEN
  $grd1 : computing = OK$
  $grd2 : step = read$
THEN
  $act1 : computing := KO$
  $act2 : display\_speed := read\_speed$
  $act3 : display\_alt := read\_alt$
  $act4 : step := tocompute$
END
```

The two next events model the abstract protocol for exchanging the data. The abstract protocol manages the relationship between the measurement units. The ontological annotation appears in the invariant $inv13$: the protocol ensures the *correct* communication.

```
EVENT Write_Info_On_Bus
  WHEN
    $grd1 : step = computed$
  THEN
    $act1 : alt\_bus := inchtometer(alt)$
    $act2 : speed\_bus := mphtokph(speed)$
    $act3 : step := written$   END
```

```
EVENT Read_Infor_From_Bus
  WHEN
    $grd1 : step = written$
  THEN
    $act1 : step := read$
    $act2 : read\_alt := alt\_bus$
    $act3 : read\_speed := speed\_bus$
  END
```

3.2.4 Context extension: Need for Explicit Knowledge on the Bus.

The current system is still abstract and we have to add details concerning the bus. Following good engineering practice, the communication bus should be described independently of any usage in a given model. Here again, an ontology of communication media is needed. It is defined in a context that extends the one defined for measure units. The bus has specific properties that are expressed in a new context domain_knowledge_for_protocols(in figure 5).

```
CONTEXT $domain\_knowledge\_for\_protocols$
EXTENDS $domain\_knowledge\_for\_units$
SETS
  $bus, tbus$
CONSTANTS
  $unidirectional, bidirectional, tbus$
AXIOMS
  $axm1 : partition(bus\_type, \{unidirectional\}, \{bidirectional\})$
  $axm2 : tbus \in bus \to bus\_type \wedge abus \in bus \wedge tbus(abus) = unidirectional$
  $axm5 : s \in bus \to kph \wedge s(abus) = mphtokph(vspeed)$
  $axm6 : a \in bus \to meter \wedge a(abus) = inchtometer(valt)$
END
```

Figure 5: The ontolological context for bus protocols

Notice that the definition of explicit knowledge is modular. It uses contexts that import only those ontologies that are needed for a given development. Moreover, it is flexible since contexts can be changed, if the domain knowledge or the nature of the manipulated concepts evolves. The whole formal development of the system does not need to be rewritten.

3.2.5 Second refinement: Concretizing the Bus for Communication.

```
$inv1 : speedbus \in bus \rightarrow kph$
$inv2 : altbus \in bus \rightarrow meter$
$onto - inv3 : altbus(b) = alt\_bus$
$onto - inv4 : speedbus(b) = speed\_bus$
$onto - inv6 : tbus(b) = unidirectional$
$onto - inv7 : b \in bus$
```

The new invariant extends the previous one, whilst integrating the state of the bus. It also asserts that the bus is *unidirectional*, which is a very important issue for ensuring security over the communications. The invariant $onto - inv7 : b \in bus$ is an ontological invariant and the context enriches the description of the domain. It explicitly expresses that the bus is *unidirectional*. Finally, the four events of the model COM10 are refined to concretize the actions over the bus b. The two first events are directly related to the computation and display components.

```
EVENT Compute_Aircraft_Info_2
REFINES Compute_Aircraft_Info_1
WHEN
  $grd1 : computing = KO$
  $grd2 : step = tocompute$
THEN
  $act1 : alt :\in inch$
  $act2 : speed :\in mph$
  $act3 : computing := OK$
  $act4 : step := computed$
END
```

```
EVENT Display_Aircraft_Info_2
REFINES Display_Aircraft_Info_1
WHEN
  $grd1 : computing = OK$
  $grd2 : step = read$
THEN
  $act1 : computing := KO$
  $act2 : display\_speed := read\_speed$
  $act3 : display\_alt := read\_alt$
  $act4 : step := tocompute$
END
```

The two next events — Write_Info_On_Bus and Read_Infor_From_Bus — model operations over the bus. They both deal with ontological annotations, where the more detailed characteristics of the bus are necessary for guaranteeing the safety of the global system.

```
EVENT Write_Info_On_Bus_1
REFINES Write_Info_On_Bus
WHEN
  $grd1 : step = computed$
THEN
  $act1 : altbus(b) := inchtometer(alt)$
  $act2 : speedbus(b) := mphtokph(speed)$
  $act3 : step := written$
END
```

```
EVENT Read_Infor_From_Bus_1
REFINES Read_Infor_From_Bus
WHEN
  $grd1 : step = written$
THEN
  $act1 : step := read$
  $act2 : read\_alt := altbus(b)$
  $act3 : read\_speed := speedbus(b)$
END
```

3.3 A didactic case study: the information system

Authors [33] have chosen a didactic case study describing a simple information system. This information system results from requirements and is described through a set of concepts, actions and constraints as it is the case for applications in the engineering domain. The defined case study deals with the management of students diplomas and registration in the European higher education system. This system offers two kinds of curricula: first the Bachelor (Licence), Master and Phd , LMD for short, and second the Engineer curriculum. Each diploma of the LMD curricula corresponds to a given level: Bachelor/Licence (high school degree + 6 semesters/180 ECTS credits), Master (Bachelor + 4 semesters/120 credits) and PhD (Master + 180 credits). The engineer curricula offers the engineer diploma five years after the high school degree. So both Master and Engineer diplomas are obtained five years after the high school degree. The document [33] shows how the ontological informations are required to ensure the refinement of models.

In the studied information system, students register to prepare their next expected diploma. This registration action takes into account the last hold academic degree (or last diploma) as a pre-requisite to register for the next diploma. Constraints on the registration action require that the information system does not allow a student to register for a new diploma if he/she does not have the necessary qualifications. Therefore, the designed information system must check the logical sequence of obtained diplomas before allowing a student to register. For example, Phd degree registration is authorized only if the last obtained degree corresponds to a Master degree. The studied information system prescribes the necessary conditions for registering students for preparing diplomas.

```
MACHINE $REGISTRATION$
SEES $ONTO$
VARIABLES
  $phdregister$
...
INVARIANTS
  $inv1 : \forall x.(x \in STUDENTS \land x \mapsto p \in phdregister \Rightarrow previousdiplom[\{x\}] \cap \{m\} \neq \varnothing)$
...
EVENTS
...
EVENT Phd_Register
  ANY
    $Dip, student$
  WHERE
    $grd1 : Dip \in \{m\}$
    $grd2 : student \in STUDENTS$
    $grd3 : Dip \in previousdiplom[\{student\}]$
  THEN
    $act1 : phdregister := phdregister \cup \{student \mapsto phd\}$
  END
...
END
```

The main problem is to extend the possible registration of students who have an engineering degree which is equivalent to have a master degree: $annotation = \{Master \mapsto m, Engineer \mapsto e\}$. We define the equivalence relationship in the next context which is extending the context $BASIS$. Contexts are derived from the ontological informations.

```
CONTEXT $ANNOTATION$
EXTENDS $BASIS$
AXIOMS
  $axm1 : annotation \in ANNOTATION\_CLASS$
  $axm2 : annotation = \{Master \mapsto m, Engineer \mapsto e\}$
END
```

Using the annotation relationship, we annotate the model $REGISTRATION$ by using the equivalence of degrees in the invariant part. The new event Phd_Register is modified by extending the set of possible degrees (Master, Engineering) including both m and e. The annotation relationship is crucial and allows to maintain the invariant. The binary relation eq over the set of degrees $\{Bachelor, Master, Engineering, PhD\}$ and $\{Master \mapsto Master, Master \mapsto Engineering, Engineering \mapsto Master, Engineering \mapsto Engineering\} \subseteq eq$. Moreover, eq is an equivalence relation and the two degrees $Master$ and $Engineering$ are equivamlent in the context $ONTO$ derived from the ontology DONTOLOGY using the plugin OntoEventB [39]: the condition $previousdiplom[\{x\}] \cap \{m\} \neq \varnothing$ is translated into $previousdiplom[\{x\}] \cap annotation[eq[annotation^{-1}[\{m\}] \neq \varnothing$.

```
MACHINE $REGISTRATION$
SEES $ONTO$
VARIABLES
  $phdregister$
...
INVARIANTS
  $inv1 : \forall x.(x \in STUDENTS \wedge x \mapsto p \in phdregister$
          $\Rightarrow previousdiplom[\{x\}] \cap annotation[eq[annotation^{-1}[\{m\}] \neq \varnothing$
...
EVENTS
...
EVENT Phd_Register
  ANY
    $Dip, student$
  WHERE
    $grd1 : Dip \in \{m, e\}$
    $grd2 : student \in STUDENTS$
    $grd3 : Dip \in previousdiplom[\{student\}]$
  THEN
    $act1 : phdregister := phdregister \cup \{student \mapsto phd\}$
  END
...
END
```

The last example shows how the Event-B models can be enriched by ontologies and how contexts can be extended using knowledges from ontologies using a mechanism called *annotation*.

4 The stepwise methodology in IMPEX

Our approach advocates the exploitation of domain knowledge, carried out by ontologies, in design models. We propose a stepwise methodology, composed of four steps, to establish a formal link between these two models. The approach is based on the definition of an annotation mechanism that represents this link. The definition of this mechanism depends on the used modeling languages for both ontologies and design models. Fig. 6 shows the overall schema of the approach.

1. **Formalization of domain knowledge.** Domain information are formalized in an ontology modeling language. Concepts, entities, relationships, constraints, rules, etc. are explicitly defined. The result is a formal ontology expressed in the chosen ontology modeling language. The semantics of this language and the associated verification techniques are used to establish properties of the ontology. The expressive power of this language has an impact on the defined ontologies (e.g. different constraint description languages may be used). Finally, the ontology shall be defined independently of any context of use. It may also be built from already existing ontologies (e.g. standard ontologies).

2. **Definition of design models.** Any formal modeling language is used to describe design models. Within this formal modeling language, users define and formalize specific design models corresponding to a given specification. Different analyses allowed by the modeling language and its associated verification technique may be performed on the designed model.

3. **Annotation of design model by references to ontologies.** Using specific mechanisms available in the formal modeling language, annotation of design models are explicitly described. Annotation consists of defining specific relationships between design models entities and ontology concepts. Different relationships are available, they have their specific properties. For example, the Is_a relationship can be used to assert that a given design model entity *is a*n ontology concept (annotation by $subsumption$). Annotation is made explicit in the design models thanks to the use of these relationships.

4. **Expression and verification of properties.** Once design models are annotated by domain ontologies, the proof context of the design models is enriched by the domain properties expressed in the

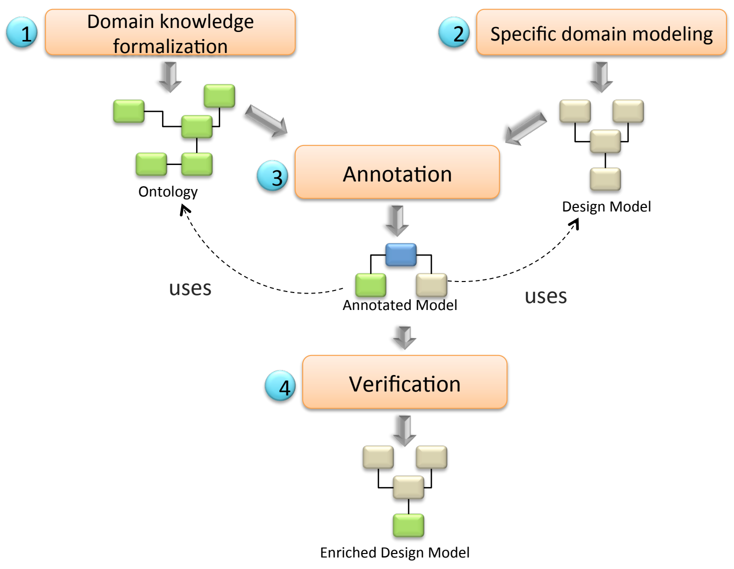

Figure 6: A four steps methodology for integrating domain knowledge and design models.

ontology. It becomes possible to check on the one side the consistence of the design models already established before annotation (they may be no longer correct after annotation) and on the other hand other properties that emerged after annotation.

At the end of this process, a new design model enriched with new information of the domain knowledge is obtained. This model makes an explicit representation of domain concepts and properties borrowed from the ontology thanks to the annotation.

4.1 Comments on the Methodology

1. It is important that the specified and used ontologies are defined in a consensual manner by the stakeholders involved in the system under design. Moreover, they should have relationships with the domain of the design model.

2. Steps 1 and 2 of the previous methodology are independent. They may be run in parallel. Ontologies may be defined prior to the design model or they may preexist.

3. In the semantic web area, lot of efforts are devoted to the definition of automatic annotation mechanisms [20, 31, 32, 34, 35, 27]. There the annotated models are documents in general and ontologies usually exploit terms rather than concepts. The definition of the annotations may be realized either by manual, semi-automatic or automatic processes[44, 47, 14, 52, 8]. In this paper, we are concerned with formal design models targeting system design. Therefore, our approach relies on an interactive annotation performed by the designer.

4. The situation where a design model is an extension (by specialization of the ontology or the design model is subsumed by the ontology) of the defined ontology may occur. This situation is ideal, it occurs when the design model is an extension of the domain ontology. In this case, reasoning by subsumption is possible. However, other situations may occur. For example in case the specific design

model is cross to several domain ontologies (i.e. multi-domain model) and uses its own concepts, such a specialization is not striaghtforward. More complex mappings (e.g. algebraic expressions, or other structural relationships) are required. The reasoning capabilities offered by subsumption may be lost and more complex reasoning mechanims may be needed.

4.2 Mathematical and logical Theories

Following the previously defined stepwise methodology to make explicit domain knowledge expressed by ontologies in design models, we propose a general formal setting in which such a methodology can be deployed for specific formal methods.

As we do not address heterogeneous semantics, the rest of this paper considers that the same satisfaction $\models$ and entailment $\vdash$ relationships are used for both of the ontology modeling language and for the design modeling language.

1. **Formalization of Domain Knowledge.** An ontology is described with an ontology modeling language. It defines axioms $A_{O_1}, \ldots, A_{O_m}$ and proof deduction rules from which properties i.e. theorems $T_{O_1}, \ldots, T_{O_n}$ may be deduced.
   - Ontologies shall be sound (healthiness of the ontology). This means that there exists a model M_O that satisfies the axioms of the ontology. We write $M_O \models A_{O_i}$ for $1 \leq i \leq m$
   - Each theorem can be deduced from the axioms and the other theorems. We write $A_{O_1}, \ldots, A_{O_m} \vdash T_{O_1}$ and $A_{O_1}, \ldots, A_{O_m}, T_{O_1}, \ldots T_{O_{i-1}} \vdash T_{O_i}$ for all $2 \leq i \leq n$
2. **Definition of design models.** The studied systems are described in the chosen modeling language. If the modeling language supports properties verification, then properties may be expressed and checked. Axioms $A_1, \ldots A_k$ and theorems $T_1, \ldots T_l$ describing the model properties are defined.
   - Described system models shall be sound (healthiness of the design model). This means that there exists a model M_D that satisfies the axioms defined for the system model. We write $M_D \models A_i$ for $1 \leq i \leq k$
   - Each theorem expressing properties on the design model can be deduced from the axioms and the other demonstrated theorems. We write $A_1, \ldots, A_k \vdash T_1$ and $A_1, \ldots, A_m, T_1, \ldots T_{i-1} \vdash T_i$ for all $2 \leq i \leq l$
3. **Annotation of design model by references to ontologies.** Annotation consists in integrating domain knowledge expressed by ontologies in the design model.
   - Integrated axioms define a sound annotation (healthiness of the annotated model). There exists a model M satisfying the axioms of both the ontology and the design model. We write $M \models A_1 \wedge \ldots \wedge A_k \wedge A_{O_1} \wedge \ldots \wedge A_{O_m}$.
4. **Expression and verification of properties.** The properties $T_1, \ldots, T_l$ shall be re-proved again once the model has been enriched by ontologies. Moreover, new emerging properties $P_1, \ldots, P_t$ may be inferred from the annotated model.
   - The properties of the design model before annotation need to be re-proved again. Indeed, the ontology may have brought relevant information that falsify 0 or more properties. We write $A_1, \ldots, A_k, A_{O_1}, \ldots, A_{O_m} \vdash T_1$ and $A_1, \ldots, A_k, A_{O_1}, \ldots, A_{O_m}, T_{O_1}, \ldots T_{O_n}, T_1, \ldots T_{i-1} \vdash T_i$ for all $2 \leq i \leq l$
   - When domain knowledge described by ontologies is embedded in the design models, new properties $P_1, \ldots P_t$ may arise. They should be proved. We write: $A_1, \ldots, A_m, A_{O_1}, \ldots, A_{O_m}, \vdash P_i$ for all $0 \leq i \leq t$

Remark. As mentioned in section **??**, we have assumed that the same deduction logic (with $\vdash$ and $\models$) is associated to both the ontology and the design models. If this is not the case, alignment of the semantics of the ontologies and of the models should be performed. This is out of the scope of this paper.

5 Conclusion and perspectives

Handling formally the domain knowledge in design models is a challenge in system and software engineering. When the capability to enrich design models with relevant knowledge and properties mined from the domain where a system evolves or a software runs is offered, the quality of the obtained design models is increased.

At the end of the task 2, we propose a generic methodology to make explicit the domain knowledge in formal system/software developments. This methodology advocates the use of formalized ontologies to model domain knowledge on the one hand, and a formally defined annotation relationship to link domain knowledge concepts with formal design models entities on the other hand. Ontologies are formalized as theories exploited by the formal modeling technique used to express design models. The annotation relationship [9, 33] enriches the proof contexts of the design models with axioms and theorems borrowed from the domain ontology.

Compared to the classical approaches of the semantic web or information retrieval, this approach is different because it addresses resources that are formal models with a formal semantics and which support formal validation and verification of properties. Such formal semantics definition and properties verification and validation are not required in case of web pages and more generally for documents. In general no semantics is associated to such documents. One may define weak XML models for these documents, but not a rigorously defined semantics. In our case, two formal semantics are defined, one for the ontologies and another for the design models. We make a clear separation between the semantics carried by the design models and the one carried by the domain ontology.

The defined methodology is not tied to a specific formal method nor to a particular ontology modeling language, but it is generic enough to be followed by different formal development approaches. It has been set up in two cases.

- Model checking of labelled transitions systems to check substitutability of design models of systems and/or software expressed with labelled transition systems. The annotation consisted in exploiting a relationship, explicitly defined in a domain ontology, between the labels of a labelled transition system. This relationship permitted to define a rewriting of labels. It has been applied to the case of plastic human computer interfaces and of semantic web services. More generally, it could be applied to study adaptive systems properties.

- Proof and refinement-based development methods illustrated by the Event-B formal method. Here, ontologies have been modeled by Event-B contexts and the annotation relationship has been formalized thanks to specific invariants, namely ontological invariants, expressing the annotation relationship. The proposed approach has been applied to the development of an avionic system involving hardware, network and software components characterized by a domain ontology formalized in an Event-B context. Ontological invariants were defined to define relevant safety properties and gluing invariants for refinements.

We have specifically addressed the case of system and/or software engineering requiring validation and verification of properties. To formalize the semantics of the manipulated models *for both ontologies and design models*, *a single* specific theoretical setting has been considered: first order logic and set theory to express both domain ontologies and design models and the underlying proof and verification procedures under a closed world assumption. This theoretical setting corresponds to the formal development methods we have experimented: model checking and proof and refinement with Event-B.

To the best of our knowledge, this work is the first attempt towards handling formally domain knowledge in formal development in an explicit manner. Indeed, the proposed approach makes a clear separation between the formalized domain ontology and the design model. The approach is completely modular. It clearly separates the design model from the domain ontology. This separation means that both ontologies and design models may evolve separately and asynchronously. Obviously, property checking shall be replayed each time an evolution of the domain ontology and/or of the design models occurs.

References

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

[2] J.-R. Abrial. Extending b without changing it (for developing distributed systems). In H. Habrias, editor, *1^{st} Conference on the B method*, pages 169–190, November 1996.

[3] J.-R. Abrial. $B^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer, June 2003.

[4] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[5] J.-R. Abrial, D. Cansell, and D. Méry. A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3):215–227, 2003.

[6] 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.

[7] Jean-Raymond Abrial and Louis Mussat. Introducing Dynamic Constraints in B. In *B98*, pages 83–128, 1998.

[8] Yamine Aït-Ameur, J. Paul Gibson, and Dominique Méry. On implicit and explicit semantics: Integration issues in proof-based development of systems - version to read. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803, pages 604–618. Springer Verlag, 2014.

[9] Yamine Aït Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Sci. Comput. Program.*, 121:100–127, 2016.

[10] R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1998.

[11] R. Back and J. von Wright. *Refinement Calculus A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[12] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[13] P. Behm, P. Benoit, A. Faivre, and J.-M.Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387, 1999.

[14] Nabil Belaid, Stéphane Jean, Yamine Aït-Ameur, and Jean-François Rainaud. An ontology and indexation based management of services and workflows application to geological modeling. *IJEBM*, 9(4):296–309, 2011.

[15] Dines Bjorner. *Software Engineering 1 Abstraction and Modelling*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006. ISBN: 978-3-540-21149-5.

[16] Dines Bjørner. *Software Engineering 1 Abstraction and Modelling; Software Engineering 2 Specification of Systems and Languages ; Software Engineering 3 Domains, Requirements, and Software Design*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006.

[17] Dines Bjorner. *Software Engineering 2 Specification of Systems and Languages*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006. ISBN: 978-3-540-21150-1.

[18] Dines Bjorner. *Software Engineering 3 Domains, Requirements, and Software Design*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006. ISBN: 978-3-540-21151-8.

[19] Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.

[20] Kalina Bontcheva, Valentin Tablan, Diana Maynard, and Hamish Cunningham. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Engineering*, 10(3/4):349–373, 2004.

[21] M. Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27:139–173, 1996.

[22] M. Butler. CSP2B: A Practical Approach To Combining CSP and B. *Formal Aspects of Computing*, 12:182–196, 200.

[23] M. Butler and M. Walden. Parallel Programming with the B Method. In *Program Development by Refinement Cases Studies Using the B Method*, volume [51] of *FACIT*, pages 183–195. Springer, 1998.

[24] D. Cansell, G. Gopalakrishnan, M. D. Jones, D. Méry, and Airy Weinzoepflen. Incremental proof of the producer/consumer property for the pci protocol. In *ZB [**?**]*, pages 22–41, 2002.

[25] Dominique Cansell and Dominique Méry. *The Event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [19].

[26] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[27] Artem Chebotko, Yu Deng, Shiyong Lu, Farshad Fotouhi, and Anthony Aristar. An ontology-based multimedia annotator for the semantic web of language engineering. *Int. J. Semantic Web Inf. Syst.*, 1(1):50–67, 2005.

[28] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[29] ClearSy, Aix-en-Provence (F). *Atelier B*, 2002. http://www.atelierb.eu.

[30] ClearSy, Aix-en-Provence (F). *B4FREE*, 2004. http://www.b4free.com.

[31] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. 2011.

[32] Sylvie Desprès and Sylvie Szulman. Terminae method and integration process for legal ontology building. In Moonis Ali and Richard Dapoigny, editors, *Advances in Applied Artificial Intelligence, 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2006, Annecy, France, June 27-30, 2006, Proceedings*, volume 4031 of *Lecture Notes in Computer Science*, pages 1014–1023. Springer, 2006.

[33] Kahina Hacid and Yamine Aït Ameur. Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 340–357, 2016.

[34] Siegfried Handschuh and Steffen Staab. CREAM: creating metadata for the semantic web. *Computer Networks*, 42(5):579–598, 2003.

[35] Siegfried Handschuh, Raphael Volz, and Steffen Staab. Annotation for the deep web. *IEEE Intelligent Systems*, 18(5):42–48, 2003.

[36] J. Hoare, J. Dick, D. Neilson, and I. Holm Sørensen. Applying the B technologies on CICS. In *FME 96*, pages 74–84. Springer, 1996.

[37] ANR IMPEX. Formal models for ontologies t1.1_d1.1. Technical Report T1.1_D1.1, Consortium IMPEX & ANR, May 2015.

[38] ANR IMPEX. Formal models for ontologies t1.2_d1.2. Technical Report T1.2_D1.2, Consortium IMPEX & ANR, June 2016.

[39] ANR IMPEX. The OntoEventB plug-in t1.3_d1.3. Technical Report T1.3_D1.3, Consortium IMPEX & ANR, December 2016.

[40] L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, May 1994.

[41] K. Lano, J. Bicarregui, and A. Sanchez. Invariant-based synthesis and composition of control algorithms using B. In *FM'99 – B Users Group Meeting – Applying B in an industrial context: Tools, Lessons and Techniques*, pages 69–86, 1999.

[42] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[43] B-Core(UK) Ltd. *B-Toolkit User's Manual*, relase 3.2 edition, 1996.

[44] Yan Lu, Hervé Panetto, Yihua Ni, and Xinjian Gu. Ontology alignment for networked enterprise information system interoperability in supply chain environment. *Int. J. Computer Integrated Manufacturing*, 26(1-2):140–151, 2013.

[45] Z. Manna and A. Pnueli. *The temporal logics of reactive and concurrent systems - Specification*. Springer-Verlag, 1992.

[46] Z. Manna and A. Pnueli. *The temporal logics of reactive and concurrent systems - Safety*. Springer-Verlag, 1995.

[47] Laura S. Mastella, Yamine Aït-Ameur, Stéphane Jean, Michel Perrin, and Jean-François Rainaud. Semantic exploitation of engineering models: An application to oilfield models. In Alan P. Sexton, editor, *Dataspace: The Final Frontier, 26th British National Conference on Databases, BNCOD 26, Birmingham, UK, July 7-9, 2009. Proceedings*, volume 5588 of *Lecture Notes in Computer Science*, pages 203–207. Springer, 2009.

[48] Dominique Méry, Rushikesh Sawant, and Anton Tarasyuk. Integrating domain-based features into event-b: A nose gear velocity case study. In Ladjel Bellatreche and Yannis Manolopoulos, editors, *Model and Data Engineering - 5th International Conference, MEDI 2015, Rhodes, Greece, September 26-28, 2015, Proceedings*, volume 9344 of *Lecture Notes in Computer Science*, pages 89–102. Springer, 2015.

[49] L. Moreau and J. Duprat. A Construction of Distributed Reference Counting. *Acta Informatica*, 37:563–595, 2001.

[50] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[51] E. Sekerinski and K. Sere, editors. *Program Development by Refinement - Cases Studies Using the B Method*. FACIT. Springer, 1998.

[52] David Simon Zayas, Anne Monceaux, and Yamine Aït-Ameur. Knowledge models to reduce the gap between heterogeneous models: Application to aircraft systems engineering. In Radu Calinescu, Richard F. Paige, and Marta Z. Kwiatkowska, editors, *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010*, pages 355–360. IEEE Computer Society, 2010.