Event-B Expression and Verification of Translation Rules Between SysML/KAOS Domain Models and B System Specifications

Steve Jeffrey Tueno Fotso[1,3], Amel Mammar[2], Régine Laleau[1], and Marc Frappier[3]

[1] Université Paris-Est Créteil, LACL, Créteil, France,
steve.tuenofotso@univ-paris-est.fr, laleau@u-pec.fr
[2] Télécom SudParis, SAMOVAR-CNRS, Evry, France,
amel.mammar@telecom-sudparis.eu
[3] Université de Sherbrooke, GRIL, Québec, Canada,
Marc.Frappier@usherbrooke.ca

Abstract. This paper is about the extension of the *SysML/KAOS* requirements engineering method with domain models expressed as ontologies. More precisely, it concerns the translation of these ontologies into *B* for system construction. The contributions of this paper are twofold. The first contribution is a formal semantics for the ontology modeling language. The second contribution is the formal definition of translation rules between ontologies and *B system* specifications in order to provide the structural part of the system specification. These translation rules are modeled in *Event-B*. Their consistency and completeness is proved using *Rodin*. We show that the translation rules are structure preserving, by proving various isomorphisms between the ontology and the B system specification. The translation rules are also implemented in an open source tool. This domain modeling approach has been applied to three significant case studies for the formal specification of the corresponding systems (a landing gear system, a localization component of an autonomous electric vehicle, and a component managing the transitions of *hybrid ERTMS/ETCS level 3* virtual sub-section states).

Keywords: *Event-B*, *B System*, Domain Modeling, Ontologies, *SysML/KAOS*, *Rodin*

1 Introduction

Our study, part of the *FORMOSE* project [5], focuses on an approach to facilitate the development of systems in critical areas such as railway or aeronautics. The implementation of such systems, in view of their complexity, requires several verifications and validation steps, more or less formal, with regard to the current regulations. In [21], rules have been defined in order to produce a formal specification from *SysML/KAOS* goal models[13, 20]. Nevertheless, the generated specification was not containing the system state. This is why in [20], we have presented the use of ontologies and *UML* class and object diagrams for domain properties representation; we have also introduced rules to derive the system state from these domain representations. Unfortunately, the proposed approach raised several concerns such as the use of several modeling formalisms for the representation of domain knowledge or the disregard of the variability aspect of domain models. In addition, the proposed rules were incomplete and informal. We have therefore proposed in [34] a language for domain knowledge representation through ontologies that meets the shortcomings of [20]. The language allows a high-level modeling of domain properties. This facilitates system constraining and enables the expression of more precise and complete properties.

In this paper, we propose rules for translating SysML/KAOS domain models into B System specifications. These rules have all been defined and the most relevant have been formally specified with *Event-B* [1] and validated through *Rodin* [9]. The Event-B method has been choosed because it involves intuitive mathematical concepts and has a powerful refinement logic. It has also been chosen because it is supported by industrial-strength tools. This paper contributes to define a formal semantics for the SysML/KAOS domain modeling language, through the definition of its metamodel and its associated constraints in the form of Event-B specifications [4]. The paper also provides the formal definition of the translation rules and summarises the benefits and difficulties of their expression and validation through Rodin. The approach has been used to describe the domain properties associated with the *hybrid ERTMS/ETCS level 3* case study [15] and to obtain the corresponding B System specifications [28]. It has also been applied on the landing gear system case study [8] and for the specification of a localisation software component that uses GPS, Wi-Fi and sensor technologies for the realtime localisation of the *Cycab* vehicle [24]. The models can be found in [29, 30]. A tool has been developed to support the approach [31]. It can be used to model domain ontologies and to automatically translate them into B System specifications in order to provide the structural part of the system formal model. The presentation of the work done on the case studies is out of the scope of this paper, but we use an excerpt from the landing gear system case study to illustrate our work.

The remainder of this paper is structured as follows: Section 2 briefly describes the SysML/KAOS requirements engineering method, the SysML/KAOS domain modeling language and the Event-B and B System formal methods. Follows a presentation, in Section 3, of the formal expression in Event-B, of the B System and SysML/KAOS domain metamodels. In Section 4, we describe the translation rules between domain models and B System specifications and we provide an overview of their formal definition. Section 5 underlines the benefits of using the Event-B method for the expression and validation of rules and some challenges encountered. It ends with a positioning of our work with regard to the state of the art. Finally, Section 6 reports our conclusions and discusses future work.

2 Context

2.1 SysML/KAOS

Requirements engineering focuses on elicitation, analysis, verification and validation of requirements. These activities, in order to be carried out, require the choice of an adequate means for requirements representation. The *KAOS* method [18] proposes to represent the requirements in the form of goals through five sub-models of which the two main ones are : the **goal model** for the representation of requirements to be satisfied by the system and of expectations with regard to the environment through a goals hierarchy and the **object model** which uses the *UML* class diagram for the representation of domain vocabulary. The goal hierarchy is built through a succession of refinements using different operators : *AND* and *OR*. An **AND refinement** decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. Dually, an **OR refinement** decomposes a goal into subgoals such that the achievement of only one of them is sufficient for the accomplishment of the parent goal. Requirements and expectations correspond to the lowest level goals of the model. However, *KAOS* offers no mechanism to maintain a strong traceability between requirements and design deliverables, making it difficult to validate them against the needs formulated. The *SysML UML profile* has been specially designed by the Object Management Group (OMG) for the analysis and specification of complex systems and allows for the capturing of requirements and the maintaining of traceability links between requirements and design deliverables. Despite these advantages, OMG has not defined a precise syntax for requirements specification. *SysML/KAOS* [13, 20] is a requirement engineering method which extends the *SysML* UML profile with a set of elements allowing to represent functional and non-functional requirements, in *SysML* models, as *KAOS* goals [18]. It combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*.

In this paper, we use the landing gear system case study to illustrate some elements of our approach [8, 29]. Figure 1 is an excerpt from its goal diagram focused on the purpose of landing gear expansion (**makeLGExtended**). To achieve it, the handle must be put down (**putHandleDown**) and landing gear sets must be extended (**makeLSExtended**). We assume that each aircraft has one landing gear.

2.2 Domain Modeling in SysML/KAOS

Domain models in SysML/KAOS are represented using ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [33, 34], a language based on *OWL* [25] and *PLIB* [23], two well-known and complementary ontology modeling languages. Figure 2 is an excerpt of its metamodel. The *parent* association represents the hierarchy of domain models. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. A *concept* (instance of metaclass Concept) represents a group of individuals sharing common characteristics. A *concept* can be declared *variable* (*isVariable=true*) when the set of its individuals can be updated through addition or deletion of individuals. Otherwise, it is considered to be *constant* (*isVariable=false*).

Figure 3 gives an excerpt from the domain model associated to the root level of the landing gear system goal model. It has been represented using the tool supporting the representation of SysML/KAOS domain models [31] which is built through the *Jetbrains Meta Programming System* [16].

In the rest of this paper, *source* is used in place of SysML/KAOS domain model.

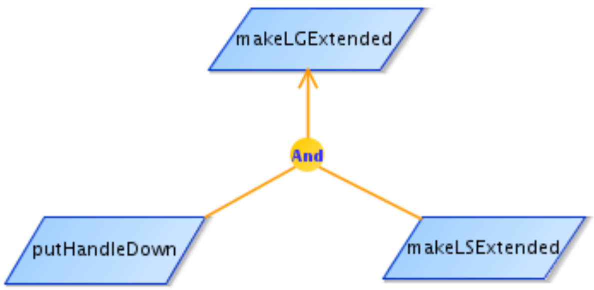

Fig. 1. Excerpt from the landing gear system goal diagram

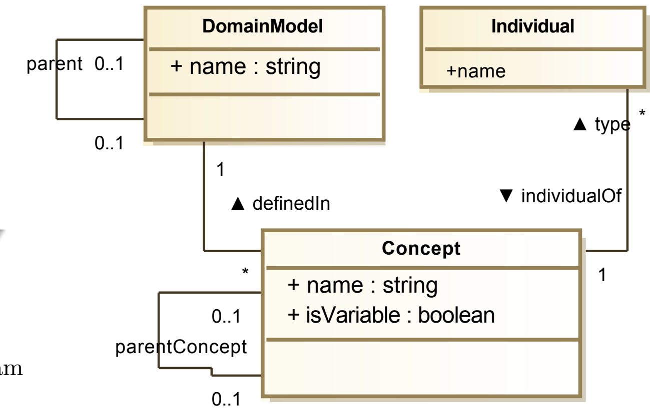

Fig. 2. Excerpt from the Metamodel Associated with the SysML/KAOS domain modeling language

```
domain model landing_gear_system_ref_0 {
concepts:
  concept LandingGear  {
    is variable: false

    individuals:
       LG1
  }
}
```

Fig. 3. ***landing_gear_system_ref_0***: Excerpt from the ontology associated to the root level of the landing gear goal model

2.3 Event-B and B System

Event-B [1] is an industrial-strength formal method for *system modeling*. It is used to incrementally construct a system specification, using refinement, and to prove useful properties. Its main purpose is the modeling of closed systems: the modeling of the system is accompanied by that of its environment and of all interactions likely to occur between them. *B System* is an Event-B syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [5], and supported by *Atelier B* [10].

Figure 4 is a metamodel of the B System language restricted to concepts that are relevant to us. A B System specification consists of components (instances of Component). Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to access the elements defined in it and to reuse them for new constructions. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and initialised through initialisation actions. Properties and invariants can be categorised as instances of LogicFormula. In our case, it is sufficient to consider that logic formulas are successions of operands in relation through operators. Thus, an instance of LogicFormula references its operators (instances of Operator) and its operands that may be instances of Variable, Constant or Set. In the same way, an instance of InitialisationAction references the operator and the operands of the assignment. Operator includes, but not limited to [4], ***Inclusion_OP*** which is used to assert that the first operand is a subset of the second operand $((Inclusion\_OP, [op_1, op_2]) \Leftrightarrow op_1 \subseteq op_2)$, ***Belonging_OP*** which is used to assert that the first operand is an element of the second operand $((Belonging\_OP, [op_1, op_2]) \Leftrightarrow op_1 \in op_2)$ and ***BecomeEqual2SetOf_OP*** which is used to initialize a variable as a set of elements $((BecomeEqual2SetOf\_OP, [va, op_2, ..., op_n]) \Leftrightarrow va := \{op_2, ..., op_n\})$.

[4] The full list can be found in [32]

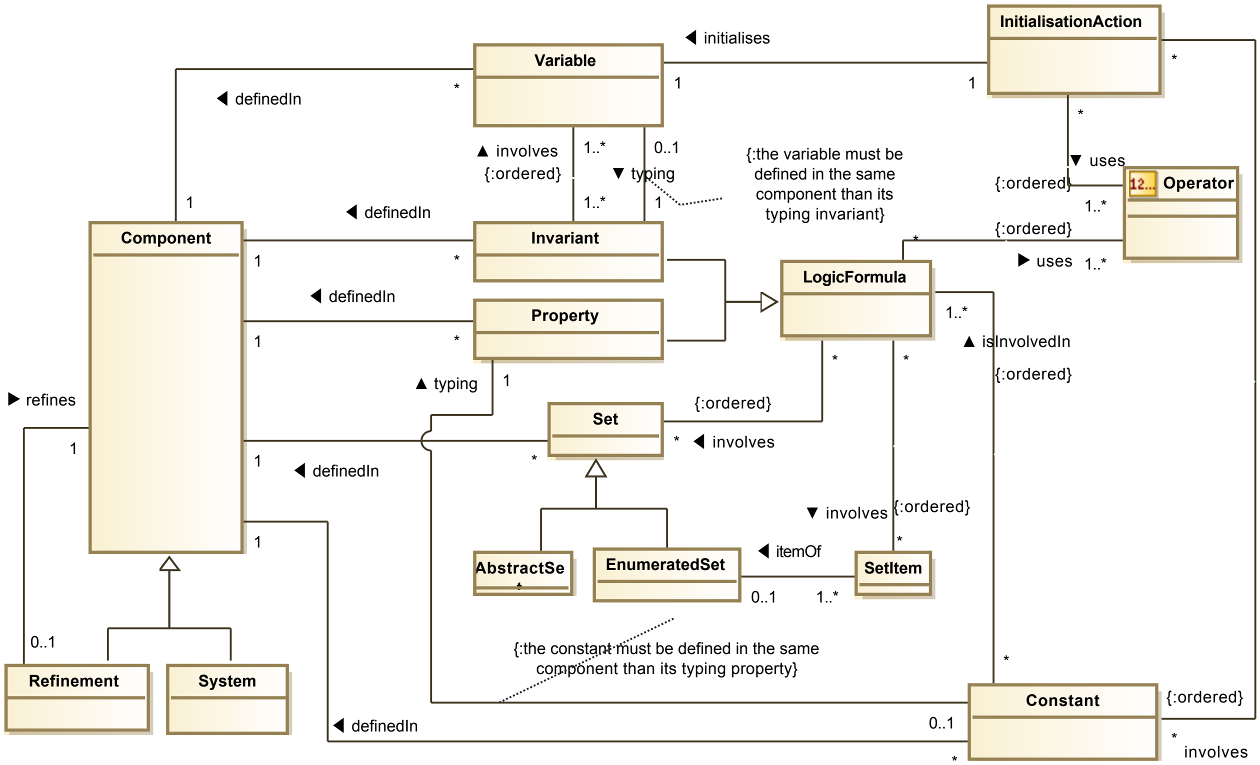

Fig. 4. Metamodel of the B System specification language

In the rest of this paper, *target* is used in place of B System.

3 Specification of Source and Target Metamodels in Event-B

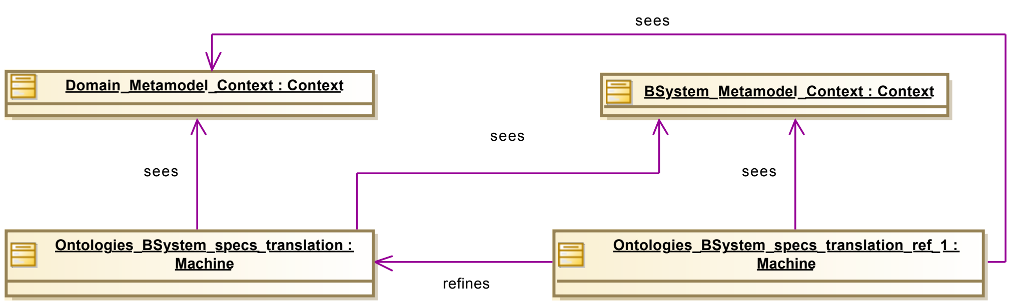

Fig. 5. Structure of the Event-B specification

As we have chosen Event-B to express and verify the translation rules between the source and target metamodels, the first step is to specified them in Event-B. This also allows us to formally define the semantics of SysML/KAOS domain models. Figure 5 represents the structure of the whole Event-B specification. This specification can only be splitten into two abstraction levels because all the translation rules use the class LogicFormula, except those related to the class DomainModel. The first machine, `Ontologies_BSystem_specs_translation`, contains the rules for the translation of instances of DomainModel into instances of Component. The other rules are defined in the machine `Ontologies_BSystem_specs_translation_ref_1`.

We have defined static elements of the target metamodel in a context named `BSystem_Metamodel_Context` and static elements of the source metamodel in the one named `Domain_Metamodel_Context`. The two machines have access to the definitions of the contexts. For the sake of concision, we provide only an illustrative excerpt of these Event-B specifications. For instance, the model `Ontologies_BSystem_specs_translation_ref_1` contains more than a hundred variables, a hundred invariants and fifty events and it gives rise to a thousand proof obligations. The full version can be found in [27, 32].

For the translation of some metamodel elements, we have followed the rules proposed in [17, 26], such as : classes which are not subclasses give rise to abstract sets, each class gives rise to a variable typed as a subset and containing its instances and each association or property gives rise to a variable typed as a relation. For example, in the following specification, class DomainModel of the source metamodel and class Component of the target metamodel give rise to abstract sets representing all their possible instances. Variables are introduced and typed (`inv0_1, inv0_2` and `inv0_3`) to represent sets of defined instances.

CONTEXT Domain_Metamodel_Context
SETS
DomainModel_Set
END

CONTEXT BSystem_Metamodel_Context
SETS
Component_Set
END

MACHINE Ontologies_BSystem_specs_translation
SEES BSystem_Metamodel_Context,Domain_Metamodel_Context
VARIABLES
Component System Refinement DomainModel
INVARIANTS
`inv0_1`: $Component \subseteq Component\_Set$
`inv0_2`: $partition(Component, System, Refinement)$
`inv0_3`: $DomainModel \subseteq DomainModel\_Set$
END

UML enumerations are represented as Event-B enumerated sets. For example, in the following specification, defined in `BSystem_Metamodel_Context`, class Operator of the target metamodel is represented as an enumerated set containing the constants *Inclusion_OP, Belonging_OP* and *BecomeEqual2SetOf_OP*.

SETS Operator
CONSTANTS Inclusion_OP Belonging_OP BecomeEqual2SetOf_OP
AXIOMS `axiom1`: $partition(Operator, \{Inclusion\_OP\}, \{Belonging\_OP\}, \{BecomeEqual2SetOf\_OP\})$

Variables are also used to represent attributes and associations [17, 26] such as the attribute isVariable of the class Concept in the source metamodel (`inv1_5`) and the association definedIn between the classes Constant and Component in the target metamodel (`inv1_7`). To avoid ambiguity, we have prefixed and suffixed each element name with that of the class to which it is attached (**e.g.** `Concept_isVariable` or `Constant_definedIn_Component`). Furthermore, for a better readability of the specification, we have chosen to add *"s"* to the name of all Event-B relations for which an image is a set (**e.g.** `Constant_isInvolvedIn_LogicFormulas` or `Invariant_involves_Variables`).

MACHINE Ontologies_BSystem_specs_translation_ref_1
REFINES Ontologies_BSystem_specs_translation
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
VARIABLES
Concept_isVariable Constant_definedIn_Component Invariant_involves_Variables Constant_isInvolvedIn_LogicFormulas
INVARIANTS
`inv1_5`: $Concept\_isVariable \in Concept \rightarrow BOOL$
`inv1_7`: $Constant\_definedIn\_Component \in Constant \rightarrow Component$
`inv1_11`: $Invariant\_involves\_Variables \in Invariant \rightarrow (\mathbb{N}_1 \nrightarrow Variable)$
`inv1_12`: $ran(union(ran(Invariant\_involves\_Variables))) = Variable$
`inv1_13`: $Constant\_isInvolvedIn\_LogicFormulas \in Constant \rightarrow \mathbb{P}_1(\mathbb{N}_1 \times LogicFormula)$
`inv1_14`: $\forall co \cdot (co \in Constant \Rightarrow ran(Constant\_isInvolvedIn\_LogicFormulas(co)) \cap Property \neq \emptyset)$
END

An association r from a class A to a class B to which the *ordered* constraint is attached is represented as a variable r typed through the invariant $r \in (A \rightarrow (\mathbb{N}_1 \twoheadrightarrow B))$. This is for example the case of the association *Invariant_involves_Variables* of the target metamodel (`inv1_11`). If instances of B have the same sequence number, then the invariant becomes $r \in (A \rightarrow \mathbb{P}_1 (\mathbb{N}_1 \times B))$. This is for example the case of the association *Constant_isInvolvedIn_LogicFormulas* of the target metamodel (`inv1_13`). Invariant `inv1_12` ensures that each variable is involved in at least one invariant and `inv1_14` ensures the same constraint for constants.

4 Translation Rules

4.1 Overview of Translation Rules

Table 1 summarises the translation rules. These rules cover the formalisation of all elements of the source metamodel, from domain models with or without parents to concepts with or without parents, including relations, individuals or attributes. It should be noted that o_x designates the result of the translation of x and that *abstract* is used for "without parent".

Table 1: Summary of the translation rules

	Domain Model		**B System**	
Translation Of	**Element**	**Constraint**	**Element**	**Constraint**
Abstract domain model	DM	$DM \in$ DomainModel $DM \notin dom(DomainModel\_parent\_DomainModel)$	o_DM	$o\_DM \in$ System
Domain model with parent	DM PDM	$\{DM, PDM\} \subseteq$ DomainModel $DomainModel\_parent\_DomainModel(DM) = PDM$ PDM has already been translated	o_DM	$o\_DM \in$ Refinement $Refinement\_refines\_Component(o\_DM) = o\_PDM$
Abstract concept	CO	$CO \in$ Concept $CO \notin dom(Concept\_parentConcept\_Concept)$	o_CO	$o\_CO \in$ AbstractSet
Concept with parent	CO PCO	$\{CO, PCO\} \subseteq$ Concept $Concept\_parentConcept\_Concept(CO) = PCO$ PCO has already been translated	o_CO	$o\_CO \in$ Constant $o\_CO \subseteq o\_PCO$
Relation	RE CO1 CO2	$\{CO1, CO2\} \subseteq$ Concept $RE \in$ Relation $Relation\_domain\_Concept(RE) = CO1$ $Relation\_range\_Concept(RE) = CO2$ $CO1$ and $CO2$ have already been translated	o_RE	**IF** $Relation\_isVariable(RE) = FALSE$ **THEN** $o\_RE \in$ Constant **ELSE** $o\_RE \in$ Variable **END** $o\_RE \in o\_CO1 \leftrightarrow o\_CO2$[5]
Attribute	AT CO DS	$CO \in$ Concept $DS \in$ DataSet $AT \in$ Attribute $Attribute\_domain\_Concept(AT) = CO$ $Attribute\_range\_Concept(AT) = DS$ CO and DS have already been translated	o_AT	**IF** $Attribute\_isVariable(AT) = FALSE$ **THEN** $o\_AT \in$ Constant **ELSE** $o\_AT \in$ Variable **END** **IF** $Attribute\_isFunctional(AT) = FALSE$ **THEN** $o\_AT \in o\_CO \leftrightarrow o\_DS$ **ELSE** $o\_AT \in o\_CO \longrightarrow o\_DS$ **END**
Concept variability	CO	$CO \in$ Concept $Concept\_isVariable(CO) = TRUE$ CO has already been translated	X_CO	$X\_CO \in$ Variable $X\_CO \subseteq o\_CO$
Individual	Ind CO	$Ind \in$ Individual $CO \in$ Concept $Individual\_individualOf\_Concept(Ind) = CO$ CO has already been translated	o_Ind	$o\_Ind \in$ Constant $o\_Ind \in o\_CO$
Data value	Dva DS	$Dva \in$ DataValue $DS \in$ DataSet $DataValue\_valueOf\_DataSet(Dva) = DS$ DS has already been translated	o_Dva	$o\_Dva \in$ Constant $o\_Dva \in o\_DS$
Relation symmetry[6]	RE	$RE \in$ Relation $Relation\_isSymmetric(RE) = TRUE$ RE has already been translated		$o\_RE^{-1} = o\_RE$
Relation maplets	RE $(M_j)_{j=1..n}$ $(a_j, i_j)_{j=1..n}$	$RE \in$ Relation $(M_j)_{j=1..n} = RelationMaplet\_mapletOf\_Relation^{-1}\{RE\}$ $\forall j \in 1..n, a_j = RelationMaplet\_antecedent\_Individual(M_j)$ $\forall j \in 1..n, i_j = RelationMaplet\_image\_Individual(M_j)$ RE and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** $Relation\_isVariable(RE) = FALSE$ **THEN** $o\_RE = \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **ELSE** $o\_RE := \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **END**
Attribute maplets	AT $(M_j)_{j=1..n}$ $(a_j, i_j)_{j=1..n}$	$AT \in$ Attribute $(M_j)_{j=1..n} = AttributeMaplet\_mapletOf\_Attribute^{-1}\{AT\}$ $\forall j \in 1..n, a_j = AttributeMaplet\_antecedent\_Individual(M_j)$ $\forall j \in 1..n, i_j = AttributeMaplet\_image\_DataValue(M_j)$ AT and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** $Attribute\_isVariable(AT) = FALSE$ **THEN** $o\_AT = \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **ELSE** $o\_AT := \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **END**

[5] As usual, this relation becomes a *function*, an *injection*, ... according to the cardinalities of *RE*.

[6] All other optional properties of an instance of Relation are translated in the same way (transitivity, ...) [32]

We are not interested in validating the transformation rules of predicates because both source and target metamodels express them using first-order logic notations.

SYSTEM *landing_gear_system_ref_0*
SETS *LandingGear*
CONSTANTS *LG1*

PROPERTIES
$LG1 \in LandingGear$
$\wedge$ $LandingGear = \{LG1\}$
END

Fig. 6. Part of the B system specification obtained by translation of the domain model of Fig. 3

Figure 6 represents an excerpt from the B System specifications obtained by translation of the root domain model of the landing gear system of Fig. 3. The root domain model is translated into a system component named `landing_gear_system_ref_0`, following the rule in line 1 of table 1. The abstract set `LandingGear` appears because *LandingGear* is an instance of the class Concept (rule in line 3). The individual `LG1` gives rise to a constant $LG1 \in LandingGear$ (line 8 of table 1). The property $LandingGear = \{LG1\}$ translates the fact that the *isVariable* property of *LandingGear* is set to *false*.

4.2 Event-B Specification of Translation Rules

The correspondence links between instances of a class A of the source metamodel and instances of a class B of the target metamodel are captured in a variable named *A_corresp_B* typed by the invariant $A\_corresp\_B \in A \rightarrowtail\!\!\!\!\!\!\!\shortmid\ B$. It is an injection because each instance, on both sides, must have at most one correspondence. The injection is partial because the elements are not translated at the same time. Thus, it is possible that at an intermediate state of the system, there are elements not yet translated. For example, correspondence links between instances of Concept and instances of AbstractSet are captured as follows

INVARIANTS `inv1_8`: $Concept\_corresp\_AbstractSet \in Concept \rightarrowtail\!\!\!\!\!\!\!\shortmid\ AbstractSet$

Translation rules have been modeled as *convergent* events. Each event execution translates an element of the source into the target. Variants and event guards and type have been defined such that when the system reaches a state where no transition is possible (deadlock state), all translations are done (see Section 5.1). Up to fifty events have been specified. The rest of this section provides an overview of the specification of some of these events in order to illustrate the formalisation process and some of its benefits and difficulties. The full specification can be found in [27, 32].

Translating a Domain Model with Parent (line 2 of table 1) The corresponding event is called *domain_model_with_parent_to_component*. It states that a domain model, associated with another one representing its parent, gives rise to a refinement component.

MACHINE Ontologies_BSystem_specs_translation
INVARIANTS
`inv0_6`: $Refinement\_refines\_Component \in Refinement \rightarrowtail Component$
`inv0_7`:
$\forall xx, px \cdot ((xx \in dom(DomainModel\_parent\_DomainModel) \wedge px = DomainModel\_parent\_DomainModel(xx)$
$\wedge\ px \in dom(DomainModel\_corresp\_Component) \ \wedge xx \notin dom(DomainModel\_corresp\_Component))$
$\Rightarrow DomainModel\_corresp\_Component(px) \notin ran(Refinement\_refines\_Component))$
Event domain_model_with_parent_to_component ⟨convergent⟩ $\widehat{=}$
correspondence of a domain model associated to a parent domain model
any
DM PDM o_DM
where
`grd0`: $dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) \neq \emptyset$
`grd1`: $DM \in dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component)$
`grd2`: $dom(DomainModel\_corresp\_Component) \neq \emptyset$

grd3: $PDM \in dom(DomainModel\_corresp\_Component)$
grd4: $DomainModel\_parent\_DomainModel(DM) = PDM$
grd5: $Component\_Set \setminus Component \neq \emptyset$
grd6: $o\_DM \in Component\_Set \setminus Component$

then

act1: $Refinement := Refinement \cup \{o\_DM\}$
act2: $Component := Component \cup \{o\_DM\}$
act3: $Refinement\_refines\_Component(o\_DM) := DomainModel\_corresp\_Component(PDM)$
act4: $DomainModel\_corresp\_Component(DM) := o\_DM$

end

END

The refinement component must be the one refining the component corresponding to the parent domain model. Guard `grd1` is the main guard of the event. It is used to ensure that the event will only handle instances of DomainModel with parent and only instances which have not yet been translated. It also guarantee that the event will be enabled until all these instances are translated. Action `act3` states that *o_DM* refines the correspondent of *PDM*. To discharge, for this event, the proof obligation related to the invariant `inv0_6`, it is necessary to guarantee that, given a domain model m not translated yet, and its parent pm that has been translated into component *o_pm*, then *o_pm* has no refinement yet. The invariant `inv0_7` then appears accordingly to encode this constraint.

Translating a Concept with Parent (line 4 of table 1) This rule leads to two events : the first one for when the parent concept corresponds to an abstract set (the parent concept does not have a parent : line 3 of table 1) and the second one for when the parent concept corresponds to a constant (the parent concept has a parent : line 4 of table 1). Below is the specification of the first event.

MACHINE Ontologies_BSystem_specs_translation_ref_1
Event concept_with_parent_to_constant_1 ⟨convergent⟩ $\widehat{=}$

case when the parent concept corresponds to an abstract set

any

CO o_CO PCO o_lg o_PCO

where

grd0: $dom(Concept\_parentConcept\_Concept) \setminus dom(Concept\_corresp\_Constant) \neq \emptyset$
grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus dom(Concept\_corresp\_Constant)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $PCO \in dom(Concept\_corresp\_AbstractSet)$
grd4: $Concept\_parentConcept\_Concept(CO) = PCO$
grd5: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$
grd7: $o\_CO \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $o\_PCO \in AbstractSet$
grd11: $o\_PCO = Concept\_corresp\_AbstractSet(PCO)$

then

act1: $Constant := Constant \cup \{o\_CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component($
$Concept\_definedIn\_DomainModel(CO))$
act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_CO) := \{1 \mapsto o\_lg\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_PCO\}$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component($
$Concept\_definedIn\_DomainModel(CO))$
act10: $Constant\_typing\_Property(o\_CO) := o\_lg$

end

END

The rule asserts that any concept, associated with another one known as its parent concept, through the parentConcept association, gives rise to a constant. The constant must be typed as a subset of the B System element corresponding to the parent concept. We use an instance of LogicFormula, named o_lg, to capture this constraint linking the concept and its parent correspondents (o_CO and o_PCO). Guard grd3 constrains the parent correspondent to be an instance of AbstractSet : $PCO \in dom(Concept\_corresp\_AbstractSet)$. Guard grd5 ensures that the event will not be triggered until the translation of the domain model containing the definition of the concept. Action act3 ensures that o_CO is defined in the component corresponding to the domain model where CO is defined. Action act6 defines the operator used by o_lg. Because the parent concept corresponds to an abstract set, o_CO is the only constant involved in o_lg (act7); o_PCO, the second operand, is a set (act8). Finally, action act9 ensures that o_lg is defined in the same component than o_CO and act10 defines o_lg as its typing predicate.

Example : concept *co*, with parent *pco* is translated into a constant typed as a subset of the correspondent of its parent.

SysML/KAOS domain model	**B System specification**
`concept` pco	**SETS** *pco*
	CONSTANTS *co*
`concept` co `parent concept` pco	**PROPERTIES** co $\subseteq$ pco

The specification of the second event (when the parent concept corresponds to a constant) is different from the specification of the first one in some points. The three least trivial differences appear at guard grd3 and at actions act7 and act8. Guard grd3 constrains the parent correspondent to be an instance of Constant : $PCO \in dom(Concept\_corresp\_Constant)$. Thus, the first and the second operands involved in o_lg are constants :

act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Lleftarrow \{ (o\_CO \mapsto \{1 \mapsto o\_lg\}), o\_PCO \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PCO) \cup \{2 \mapsto o\_lg\}\}$

act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

This approach to modeling logic formulas allows us to capture all the information conveyed by the predicate which can then be used to make inferences and semantic analysis. It is especially useful when we deal with rules to propagate changes made to a generated B system specification back to the domain model (ie, propagate changes made to the target into the source). The study of these propagation rules will be the next step in our work. The following Event-B specification allows for example, from a B System constant o_CO, to evaluate the typing predicate o_lg in order to build its correspondent CO within the domain model.

any CO o_CO PCO o_lg o_PCO
where
grd1: $o\_CO \in dom(Constant\_typing\_Property) \setminus ran(Concept\_corresp\_Constant)$
grd2: $o\_lg = Constant\_typing\_Property(o\_CO)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) \neq \emptyset$
grd5: $(2 \mapsto o\_PCO) \in LogicFormula\_involves\_Sets(o\_lg)$
grd6: $o\_PCO \in ran(Concept\_corresp\_AbstractSet)$
grd7: $PCO = Concept\_corresp\_AbstractSet^{-1}(o\_PCO)$
grd8: $CO \in Concept\_Set \setminus Concept$
then
act1: $Concept := Concept \cup \{CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Concept\_parentConcept\_Concept(CO) := PCO$

Abstract set o_PCO turns out to be the parent of o_CO because it appears as second operand in o_lg (grd5). Its correspondent PCO is then set as the parent of CO (act3).

Handling the Variability : Concept[isVariable=TRUE] (line 7 of table 1) Any instance of the class Concept having its isVariable property set to *true* gives rise to a variable representing the set of its defined elements. The variable must be typed as a subset of the B System element corresponding to the concept and it must be initialised to the set of constants corresponding to the instances of the class Individual linked to the concept. This rule leads to two events : the first one for when the concept corresponds to an abstract set and the second one for when the concept corresponds to a constant. Below is the specification of the first event.

Event variable_concept_to_variable_1 ⟨convergent⟩ $\widehat{=}$
any CO x_CO o_lg o_CO o_ia o_inds bij_o_inds
where
grd1: $CO \in (dom(Concept\_corresp\_AbstractSet) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Concept\_corresp\_Variable)$
grd2: $Individual\_individualOf\_Concept^{-1}[\{CO\}] \subseteq dom(Individual\_corresp\_Constant)$
grd3: $x\_CO \in Variable\_Set \setminus Variable$
grd4: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd5: $o\_CO = Concept\_corresp\_AbstractSet(CO)$
grd6: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$
grd7: $o\_inds = Individual\_corresp\_Constant[Individual\_individualOf\_Concept^{-1}[\{CO\}]]$
grd8: $finite(o\_inds)$
grd9: $bij\_o\_inds \in 1\,..\,card(o\_inds) \rightarrowtail\!\!\!\!\twoheadrightarrow o\_inds$
then
act1: $Variable := Variable \cup \{x\_CO\}$
act2: $Concept\_corresp\_Variable(CO) := x\_CO$
act3: $Invariant := Invariant \cup \{o\_lg\}$
act4: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act5: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act6: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto x\_CO\}$
act7: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_CO\}$
act8: $InitialisationAction := InitialisationAction \cup \{o\_ia\}$
act9: $InitialisationAction\_uses\_Operators(o\_ia) := \{1 \mapsto BecomeEqual2SetOf\_OP\}$
act10: $Variable\_init\_InitialisationAction(x\_CO) := o\_ia$
act11: $InitialisationAction\_involves\_Constants(o\_ia) := bij\_o\_inds$
act12: $Variable\_typing\_Invariant(x\_CO) := o\_lg$

Guard `grd1` is used to select an instance of Concept CO, which does not correspond to a variable and for which the *isVariable* property is set to *true*. Guard `grd2` ensures that the rule will only be active when all concept individuals have been translated. Variable o_inds represents the set of their correspondents. We use an instance of LogicFormula, named o_lg, to capture the first constraint linking the variable to the concept correspondent ($x\_CO \subseteq o\_CO$). Furthermore, we use an instance of InitialisationAction, named o_ia, to capture the second constraint, initialising x_CO to o_inds, the set of constants corresponding to concept individuals. Action `act5` defines the operator used by o_lg. Action `act10` defines x_CO as the variable to initialise and action `act11` defines the operands of the initialisation using bij_o_inds, a bijection used to define an order on items of o_inds.

5 Discussion and Experience

The rules that we propose allow the automatic translation of domain properties, modeled as ontologies, to B System specifications. They allow to drastically close the gap between the system textual description and its formal specification. It is thus possible to benefit from all the advantages of a high-level modeling approach within the framework of the formal specification of systems : decoupling between formal specification handling difficulties and system modeling; better reusability and readability of models; strong traceability between the system structure and stakeholder needs. Applying the approach on case studies [28–30] allowed us to quickly build the refinement hierarchy of the system and to determine and express the safety invariants, without having to manipulate the formal specifications. Furthermore, it allows us to limit our formal specification to the perimeter defined by the expressed needs. This step also allowed us to enrich the domain modeling language expressiveness. The rest of this section consists of a brief overview of the benefits and challenges associated with the Event-B formalisation of the rules under Rodin. Follows a positioning of our work with regard to the state of the art.

5.1 Benefits

Formally defining the SysML/KAOS domain modeling language, using Event-B, allowed us to completely fulfill the criteria for it to be an ontology modeling formalism [4]. Furthermore, formally defining the rules in Event-B and discharging the associated proof obligations allowed us to prove their consistency, to animate them using *ProB* [19] and to reveal several constraints (guards and invariants) that were missing when designing the rules informally or when specifying the metamodels. For instance:

- If an instance of Concept x, with parent px does not have a correspondent yet and if px does, then, the correspondent of px should not be refined by any instance of Component (`inv0_7` defined in`Ontologies_BSystem_specs_translation` and described in Section 4.2).
- Elements of an enumerated data set should have correspondents if and only if the enumerated data set does.

- If a concept, given as the domain of an attribute (instance of Attribute), is variable, then the attribute must also be variable. If this is not the case, then it will be possible to reach a state where an attribute maplet (instance of AttributeMaplet) is defined for a non-existing individual (because the individual has been dynamically removed).
- If the domain or the range of a relation is variable, then the relation must also be variable.

These constraints have been integrated in the SysML/KAOS domain modeling language in order to strengthen its semantics.

There are two essential properties that the specification of the rules must ensure and that we have proved using Rodin. The first one is that the rules are isomorphisms and it guarantees that established links between elements of the ontologies are preserved between the corresponding elements in the B System specification and vice versa. To do this, we have introduced, for each link between elements, an invariant guaranteeing the preservation of the corresponding link between the correspondences and we have discharged the associated proof obligations. This leads to fifty or so invariants. For example, to ensure that for each domain model pxx, parent of xx, the correspondent of xx refines the correspondent of pxx and vice versa, we have defined the invariants `inv0_8` and `inv0_9`:

inv0_8: $\forall xx, pxx \cdot (\ (xx \in dom(DomainModel\_parent\_DomainModel)\ \wedge\ pxx = DomainModel\_parent\_DomainModel(xx)\ \wedge \{xx, pxx\} \subseteq dom(DomainModel\_corresp\_Component)) \Rightarrow (DomainModel\_corresp\_Component(xx) \in dom(Refinement\_refines\_Component) \wedge Refinement\_refines\_Component(DomainModel\_corresp\_Component(xx)) = DomainModel\_corresp\_Component(pxx))\ )$

inv0_9: $\forall o\_xx, o\_pxx \cdot (\ (o\_xx \in dom(Refinement\_refines\_Component)\ \wedge\ o\_pxx = Refinement\_refines\_Component(o\_xx)\ \wedge \{o\_xx, o\_pxx\} \subseteq ran(DomainModel\_corresp\_Component)) \Rightarrow (DomainModel\_corresp\_Component^{-1}(o\_xx) \in dom(DomainModel\_parent\_DomainModel) \wedge DomainModel\_parent\_DomainModel(DomainModel\_corresp\_Component^{-1}(o\_xx)) = DomainModel\_corresp\_Component^{-1}(o\_pxx))\ )$

To discharge the proof obligations related to `inv0_8` and `inv0_9`, invariants `inv0_10` and `inv0_11` have been defined to guarantee an order between translation rules : the parent of xx is always translated before xx.

inv0_10: $\forall xx, pxx \cdot (\ (xx \in dom(DomainModel\_parent\_DomainModel)\ \wedge\ pxx = DomainModel\_parent\_DomainModel(xx)\ \wedge pxx \notin dom(DomainModel\_corresp\_Component)) \Rightarrow xx \notin dom(DomainModel\_corresp\_Component)\ )$

inv0_11: $\forall o\_xx, o\_pxx \cdot (\ (o\_xx \in dom(Refinement\_refines\_Component)\ \wedge\ o\_pxx = Refinement\_refines\_Component(o\_xx)\ \wedge o\_pxx \notin ran(DomainModel\_corresp\_Component)) \Rightarrow o\_xx \notin ran(DomainModel\_corresp\_Component)\ )$

The second essential property is to demonstrate that the system will always reach a state where all translations have been established (*P0*). To manually demonstrate *P0*, we have proven that all events can be disabled if and only if all translations have been done. For example, let's consider rules dealing about the translation of instances of DomainModel (lines 1 and 2 of table 1); the negation of guards results in

$$DomainModel \setminus (dom(DomainModel\_corresp\_Component) \cup dom(DomainModel\_parent\_DomainModel)) = \emptyset \wedge dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) = \emptyset$$
$$\Leftrightarrow$$
$$DomainModel \subseteq (dom(DomainModel\_corresp\_Component) \cup dom(DomainModel\_parent\_DomainModel)) \wedge dom(DomainModel\_parent\_DomainModel) \subseteq dom(DomainModel\_corresp\_Component)$$
$$\Leftrightarrow$$
$$DomainModel \subseteq dom(DomainModel\_corresp\_Component)$$
$$\Leftrightarrow$$
$$DomainModel = dom(DomainModel\_corresp\_Component)$$
because $dom(DomainModel\_corresp\_Component) \subseteq DomainModel$

To automatically prove *P0*, we have introduced, within each machine, a *variant* defined as the difference between the set of elements to be translated and the set of elements already translated. Then, each event representing a translation rule has been marked as *convergent* and we have discharged the proof obligations ensuring that each of them decreases the *variant*. For example, in the machine `Ontologies_BSystem_specs_translation` containing the definition of translation rules from domain models to B System components, the variant was defined as $DomainModel \setminus dom(DomainModel\_corresp\_Component)$. Thus, at the end of system execution, we will have $dom(DomainModel\_corresp\_Component) = DomainModel$, which will reflect the fact that each domain model has been translated into a component.

5.2 Challenges

There is no predefined type for ordered sets in Event-B. This problem led us to the definition of composition of functions in order to define relations on ordered sets. Moreover, because of the size of our model (about one hundred invariants and about fifty events for each machine), we noted a rather significant performance reduction of *Rodin* during some operations such as the execution of auto-tactics or proof replay on undischarged proof obligations that have to be done after each update in order to discharge all previously discharged proofs.

Table 3 summarises the key characteristics of the Rodin project corresponding to the Event-B specification of metamodels and rules.

Table 3. Key Characteristics of the Event-B Specification Rodin Project

Characteristics	**Ontologies_BSystem_-specs_translation**	**Ontologies_BSystem_specs_-translation_ref_1**
Events	**3**	**50**
Invariants	**11**	**98**
Proof Obligations (PO)	**37**	**990**
Automatically Discharged POs	**27**	**274** (86 for the *INITIALISATION* event)
Interactively Discharged POs	**10**	**716** (Most used provers: *ML, PP, SMTs*)

The automatic provers seemed least comfortable with functions ($\mapsto\!\!\!\!\!\!+, \rightarrowtail\!\!\!\!\!\!+, \rightarrow, \twoheadrightarrow\!\!\!\!\!\!+$) and become almost useless when those operators are combined in definitions as for ordered associations ($r \in (A \rightarrow (\mathbb{N}_1 \mapsto\!\!\!\!\!\!+ B))$).

5.3 Related Work

The study of correspondence links between domain models or ontologies and formal methods has been the subject of numerous works. The work presented in [6] is interested in describing entities, their mereology, their behaviours and their transformations. Rules are provided for the formalisation of these elements. On the other hand, our study is focused on the description of entities of a system application domain and their instances, of their constraints and of their attributes and associations. Moreover, our modeling is done through successive refinements and the translation rules integrate the refinement links between modules. In [35], an approach is proposed for the automatic extraction of domain knowledge, as *OWL* ontologies, from *Z/Object-Z (OZ)* models [11]. : OZ types and classes are transformed into OWL classes. Relations and functions are transformed into OWL properties. OZ constants are translated into OWL individuals. Rules are also proposed for subsets and state schemas. A similar approach is proposed in [12], for the extraction of *DAML* ontologies [14] from *Z* models. These approaches are interested in correspondence links between formal methods and ontologies, but their rules are restricted to the extraction of domain model elements from formal specifications. Furthermore, all elements extracted from a formal model are defined within a single ontology component, while in our approach, each ontology refinement level corresponds to a formal model component. Some rules for passing from an *OWL* ontology representing a domain model to Event-B specifications are proposed in [2], in [3] and through a case study in [20]. The approaches in [2] and [3] require a manual transformation of the ontology before the possible application of translation rules to obtain the formal specifications. In [2], it is necessary to convert OWL ontologies into UML diagrams. In [3], the proposal requires the generation of a controlled English version of the *OWL* ontology which serves as the basis for the development of the Event-B specification. Furthermore, for this to be completed, the names of ontology elements must necessarily be expressed in English. Moreover, since the OWL formalism supports weak typing and multiple inheritance, the approaches define a unique Event-B abstract set named *Thing*. Thus, all sets, corresponding to OWL classes, are defined as subsets of *Thing*. Our formalism, on the other hand, imposes strong typing and simple inheritance; which makes it possible to translate some concepts into

Event-B abstract sets. Several shortcomings are common to these approaches : the provided rules do not take into account the refinement links between model parts. Furthermore, they are provided in an informal way and they are not supported by tools. Finally, the approaches are only interested in static domain knowledge : they do not distinguish what gives rise to formal constants or variables.

Many studies have been done on the translation of UML diagrams into B specifications such as [17, 26]. They inspired many of our rules, like those dealing with the translation of concepts (classes) and of attributes and relations (associations). But, our work differs from them because of the distinctions between ontologies and UML diagrams : within an ontology, concepts or classes and their instances are represented within the same model as well as the predicates defining domain constraints. Moreover, these studies are most often interested in the translation of model elements and not really in handling links between models. Finally, in the case of the SysML/KAOS domain modeling language, the variability properties (attributes characterising the belonging of an element to the static or dynamic knowledge) are first-class citizens, as well as association characteristics. As a result, they are explicitly represented. In [7], an approach for modeling the theoretical foundations of Event-B using Event-B is sketched in order to validate Event-B plugins related to distribution and Event-B extensions related to composition and decomposition. However, the proposal considers neither Event-B contexts (Sets, Constants, Properties) nor refinement links and the definition of predicates makes their representation too abstract.

6 Conclusion and Future Works

This paper proposes an Event-B formalisation of translation rules between domain ontologies and B System specifications. Their consistency has been proved through Rodin [9], which allowed us to prove some properties regarding rules such as isomorphisms and to determine some guards and invariants that were missed during their initial specification. The translation rules have been implemented within an open source tool [31], allowing the construction of domain models and the generation of the corresponding B System specifications. It is built through *Jetbrains Meta Programming System* [16], a tool to design domain specific languages using language-oriented programming. It has been used to apply the approach on three significant case studies [28–30]. Our work allows the complete extraction of the structural part of the system formal specification from domain models. We also extract the initialisation of system variables. The specification obtained completes models resulting from the formalisation of SysML/KAOS goal diagrams. However, it remains necessary to manually provide the body of events, which can lead to updates on the structure of the system.

Work in progress is aimed at evaluating the impact of updates on formal specifications within domain models. We are also working on integrating the translation rules within the open-source platform *Openflexo* [22] which federates the various contributions of *FORMOSE* project partners [5] and which currently supports the construction of SysML/KAOS goal diagrams and domain models.

Acknowledgment

This work is carried out within the framework of the *FORMOSE* project [5] funded by the French National Research Agency (ANR). It is also partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Alkhammash, E., Butler, M.J., Fathabadi, A.S., Cîrstea, C.: Building traceable Event-B models from requirements. Sci. Comput. Program. 111, 318–338 (2015)
3. Alkhammash, Eman H.: Derivation of Event-B Models from OWL Ontologies. MATEC Web Conf. 76, 04008 (2016)
4. Ameur, Y.A., Baron, M., Bellatreche, L., Jean, S., Sardet, E.: Ontologies in engineering: the OntoDB/OntoQL platform. Soft Comput. 21(2), 369–389 (2017)
5. ANR-14-CE28-0009: Formose ANR project (2017)
6. Bjørner, D., Eir, A.: Compositionality: Ontology and mereology of domains. Essays in Honor of Willem-Paul de Roever, LNCS, vol. 5930, pp. 22–59. Springer (2010)

7. Bodeveix, J., Filali, M., Bhiri, M.T., Siala, B.: An event-b framework for the validation of event-b refinement plugins. CoRR abs/1701.00960 (2017), http://arxiv.org/abs/1701.00960
8. Boniol, F., Wiels, V.: The landing gear system case study. pp. 1–18. ABZ, Springer (2014)
9. Butler, M.J., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project], Lecture Notes in Computer Science, vol. 4157. Springer (2006)
10. ClearSy: Atelier B: B System (2014), http://clearsy.com/
11. Doberkat, E.: The Object-Z specification language. Softwaretechnik-Trends 21(1) (2001)
12. Dong, J.S., Sun, J., Wang, H.H.: Z approach to semantic web. In: Formal Methods and Software Engineering - ICFEM, LNCS. vol. 2495, pp. 156–167. Springer (2002)
13. Gnaho, C., Semmak, F., Laleau, R.: Modeling the impact of non-functional requirements on functional requirements. In: Parsons, J., Chiu, D.K.W. (eds.) Advances in Conceptual Modeling - ER 2013 Workshops, LSAWM, MoBiD, RIGiM, SeCoGIS, WISM, DaSeM, SCME, and PhD Symposium, Hong Kong, China, November 11-13, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8697, pp. 59–67. Springer (2013), https://doi.org/10.1007/978-3-319-14139-8_8
14. van Harmelen, F., Patel-Schneider, P.F., Horrocks, I.: Reference description of the DAML+ OIL ontology markup language (2001)
15. Hoang, T.S., Butler, M., Reichl, K.: The Hybrid ERTMS/ETCS Level 3 Case Study. ABZ pp. 1–3 (2018), http://users.ecs.soton.ac.uk/asf08r/ABZ2018/ERTMS_L3_Hybrid.pdf
16. Jetbrains: Jetbrains mps (2017), https://www.jetbrains.com/mps/
17. Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from UML notations. In: The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11-15, 2000. pp. 269–272. IEEE Computer Society (2000), https://doi.org/10.1109/ASE.2000.873675
18. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
19. Leuschel, M., Butler, M.J.: Prob: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer (2003)
20. Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based Event-B specifications. In: ISoLA 2016, LNCS. pp. 325–339. Springer (2016)
21. Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: ICECCS 2011. pp. 139–148. IEEE Computer Society (2011)
22. Openflexo: Openflexo project (2015), http://www.openflexo.org
23. Pierra, G.: The PLIB ontology-based approach to data integration. In: IFIP 18th World Computer Congress. IFIP, vol. 156, pp. 13–18. Kluwer/Springer (2004)
24. Sekhavat, S., Valadez, J.H.: The Cycab robot: a differentially flat system. In: IROS 2000. pp. 312–317. IEEE (2000)
25. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)
26. Snook, C., Butler, M.: UML-B: Formal Modeling and Design Aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (Jan 2006)
27. Tueno, S.: Event-B Specification of Translation Rules (2017), https://github.com/stuenofotso/SysML_KAOS_-Domain_Model_Parser/tree/master/SysMLKAOSDomainModelRules_ordered_attempt2
28. Tueno, S.: SysML/KAOS Domain Modeling Approach on ERTMS/ETCS Level 3 Hybrid (2017), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/ABZ18_ERTMS
29. Tueno, S.: SysML/KAOS Domain Modeling Approach on Landing Gear (2017), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/LandingGear
30. Tueno, S.: SysML/KAOS Domain Modeling Approach on Localisation Component of Cycab (2017), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/Tacos_Vehicle
31. Tueno, S.: SysML/KAOS Domain Modeling Tool (2017), https://github.com/stuenofotso/SysML_KAOS_-Domain_Model_Parser
32. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Formal Representation of SysML/KAOS Domain Model (Complete Version). ArXiv e-prints, cs.SE, 1712.07406 (Dec 2017)
33. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Approach. ArXiv e-prints, cs.SE, 1710.00903 (Sep 2017)
34. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Towards Using Ontologies for Domain Modeling within the SysML/KAOS Approach. IEEE proceedings of MoDRE workshop, 25th IEEE International Requirements Engineering Conference (2017)
35. Wang, H.H., Damljanovic, D., Sun, J.: Enhanced semantic access to formal software models. In: Formal Methods and Software Engineering - ICFEM, LNCS. vol. 6447, pp. 237–252. Springer (2010)