Formal Representation of SysML/KAOS Domain Model (Complete Version)

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

November 16, 2017

Abstract. Nowadays, the usefulness of a formal language for ensuring the consistency of requirements is well established. The work presented here is part of the definition of a formally-grounded, model-based requirements engineering method for critical and complex systems. Requirements are captured through the *SysML/KAOS* method and the targeted formal specification is written using the *Event-B* method. Firstly, an *Event-B* skeleton is produced from the goal hierarchy provided by the SysML/KAOS goal model. This skeleton is then completed in a second step by the Event-B specification obtained from system application domain properties that gives rise to the system structure. Considering that the domain is represented using ontologies through the *SysML/KAOS Domain Model* method, is it possible to automatically produce the structural part of system Event-B models ? This paper proposes a set of generic rules that translate SysML/KAOS domain ontologies into an Event-B specification. They are illustrated through a case study dealing with a landing gear system. Our proposition makes it possible to automatically obtain, from a representation of the system application domain in the form of ontologies, the structural part of the Event-B specification which will be used to formally validate the consistency of system requirements.

Keywords: *Event-B*, Domain Modeling, Ontologies, Requirements Engineering, *SysML/KAOS*, Formal Validation

1 Introduction

This article focuses on the development of systems in critical areas such as railway or aeronautics. The implementation of such systems, in view of their complexity, requires several validation steps, more or less formal[4], with regard to the

[4] through formal methods

current regulations. Our work is part of the *FORMOSE* project [4] which integrates industrial partners involved in the implementation of critical systems for which the regulation imposes formal validations. The contribution presented in this paper represents a straight continuation of our research work on the formal specification of systems whose requirements are captured with *SysML/KAOS* goal models. The *Event-B* method [1] has been choosen for the formal validation steps because it involves simple mathematical concepts and has a powerful refinement logic facilitating the separation of concerns. Furthermore, it is supported by many industrial tools. In [11], we have defined translation rules to produce an Event-B specification from *SysML/KAOS* goal models. Nevertheless, the generated Event-B specification does not contain the system state. This is why in [10], we have presented the use of ontologies and *UML* class and object diagrams for domain properties representation and have also introduced a first attempt to complete the Event-B model with specifications obtained from the translation of 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 variable entities. In addition, the proposed translation rules did not take into account several elements of the domain model such as data sets or predicates. We have therefore proposed in [17] a formalism for domain knowledge representation through ontologies. This paper is specifically concerned with establishing correspondence links between this new formalism called *SysML/KAOS Domain Modeling* and Event-B. The proposed approach allows a high-level modeling of domain properties by encapsulating the difficulties inherent in the manipulation of formal specifications. This facilitates system constraining and enables the expression of more precise and complete properties. The approach also allows further reuse and separation of concerns.

The remainder of this paper is structured as follows: Section 2 briefly describes the Event-B formal method, the SysML/KAOS requirements engineering method, the formalization in Event-B of SysML/KAOS goal models and the SysML/KAOS domain modeling formalism. Follows a presentation, in Section 3, of the relevant state of the art on the formalization of domain knowledge representations. In Section 4, we describe and illustrate our matching rules between domain models and Event-B specifications. Finally, Section 5 reports our conclusions and discusses our future work.

2 Background

In this section, we provide a brief overview of the *Event-B* formal method, of the *SysML/KAOS* requirements engineering method, of the formalization in *Event-B* of *SysML/KAOS* goal models and of the *SysML/KAOS* domain modeling approach.

2.1 Event-B

Event-B is an industrial-strength formal method defined by *J. R. Abrial* in 2010 for *system modeling* [1]. It is used to prove the preservation of safety invariants about a system. *Event-B* is mostly used for 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.

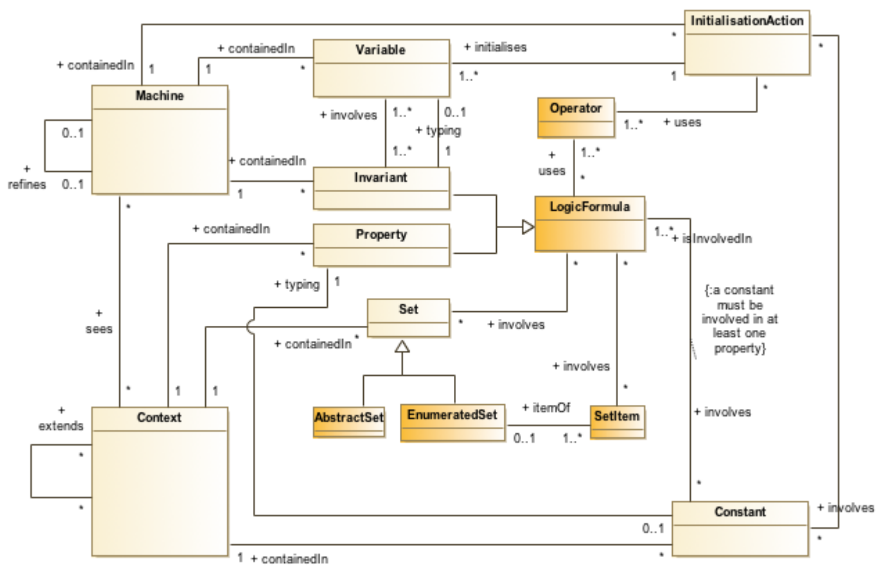


Fig. 1. Event-B Metamodel

Figure 1 is an excerpt from the *Event-B* metamodel. We have represented in orange some categories that do not appear explicitly in Event-B specifications, but which will be useful to better describe our formalization rules. An *Event-B* model includes a static part called ***Context*** and a dynamic part called ***Machine***. The ***Context*** contains the declarations of abstract and enumerated sets, constants and properties. An enumerated set is constructed by specifying its items which are instances of SetItem. The ***Machine*** contains variables, invariants and events. Moreover, a machine can see contexts. Properties and invariants can be categorised as instances of LogicFormula. An instance of LogicFormula consists of a number of operators applied on operands that may be variables, constants, or sets. A machine also contains initialisation actions which are used to define the initial value of each variable. An instance of InitialisationAction references the operator and the operands of the assignment, knowing that the action must initialise all variable operands. some operators and their actions :

- ***Inclusion_ OP*** : it is used to assert that its first operand is a subset of its second operand. The second operand has to be an instance of Set or an instance of Constant or Variable typed as a subset. The first operand can only be an instance of Constant or an instance of Variable.
- ***Belonging_ OP*** : it is used to assert that its first operand is an element of its second operand. The second operand has to be an instance of Set or an instance of Constant or Variable typed as a subset. The first operand can only be an instance of Constant or an instance of Variable.

The system specification can be constructed using stepwise refinement. A machine can refine another machine, by adding new events or by reducing non-determinacy of existing events. A refinement step can also introduce new state variables or replace abstract variables by more concrete ones. Furthermore, a context can extend another one in order to access the elements defined in it and to reuse them for new constructions.

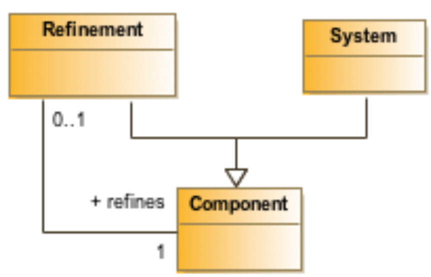


Fig. 2. B System Components

In the rest of this paper, we will illustrate our formal models using *B System*, an *Event-B* variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project, in its integrated development environment *Atelier B* [6]. A *B System* specification considers the notion of Component to specify machines and contexts, knowing that a component can be a system or a refinement (figure 2). Although it is advisable to always isolate the static and dynamic parts of the *B System* formal model, it is possible to define the two parts within the same component, for simplification purposes. In the following sections, our *B System* models will be presented using this facility.

2.2 SysML/KAOS Requirements Engineering Method

Requirements engineering focuses on defining and handling requirements. These and all related activities, in order to be carried out, require the choice of an adequate means for requirements representation. The *KAOS* method [9,10], proposes to represent the requirements in the form of goals, which can be *functional* or *non-functional*, through five sub-models of which the two main ones are : **the object model** which uses the *UML* class diagram for the representation of domain vocabulary and **the goal model** for the determination of requirements to be satisfied by the system and of expectations with regard to the environment through a goals hierarchy. *KAOS* proposes a structured approach to obtaining the requirements based on expectations formulated by stakeholders. Unfortunately, it offers no mechanism to maintain a strong traceability between those requirements and deliverables associated with system design and implementation, 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 those requirements and design diagrams resulting from the system design phase. Unfortunately, OMG has not defined a formal semantics and an unambiguous syntax for requirements specification. *SysML/KAOS* [7] therefore proposes to extend the *SysML* metamodel with a set of concepts allowing to represent requirements in *SysML* models as *KAOS* goals.

Figure 3 is an excerpt from the landing gear system [5] goal diagram focused on the purpose of landing gear expansion. We assume that each aircraft has one landing gear system which is equipped with three landing sets which can be each extended or retracted. We also assume that in the initial state, there is one landing gear named *LG1* which is extended and is associated to one handle named *HD1* which is down and to landing sets *LS1*, *LS2* and *LS3* which are all extended.

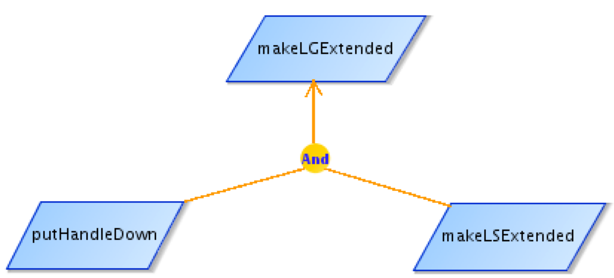


Fig. 3. Excerpt from the landing gear system goal diagram

To achieve the root goal, which is the extension of the landing gear (**makeLGExtended**), the handle must be put down (**putHandleDown**) and landing gear sets must be extended (**makeLSExtended**).

2.3 From SysML/KAOS Goal Model to Event-B

The matching between *SysML/KAOS* modeling and *Event-B* specifications is the focus of the work done by [11]. Each layer of abstraction of the goal diagram gives rise to an *Event-B* machine, each goal of the layer giving rise to an event. The refinement links are materialized within the *Event-B* specification through a set of proof obligations and refinement links between machines and between events. Figure 4 represents the *B System* specifications associated with the most abstract layer of the *SysML/KAOS* goal diagram of the Landing Gear System illustrated through Figure 3.

```
SYSTEM
    LandingGearSystem
SETS
CONSTANTS
PROPERTIES
VARIABLES
INVARIANT
INITIALISATION
EVENTS
    makeLGExtended=
    BEGIN /* extension of the landing gear */
    END
END
```

Fig. 4. Formalization of the root level of the Landing Gear System goal model

As we can see, the state of the system and the body of events must be manually completed. The state of a system is composed of variables, constrained by an invariant, and constants, constrained by properties. The objective of our study is to automatically derive this state in the *Event-B* model starting from *SysML/KAOS* domain models.

2.4 SysML/KAOS Domain Modeling

We present, through Figures 5 and 8 the metamodel associated with the *SysML/KAOS* domain modeling approach [17] which is an ontology modeling formalism for the modeling of domain knowledge in the framework of the *SysML/KAOS* requirements engineering method.

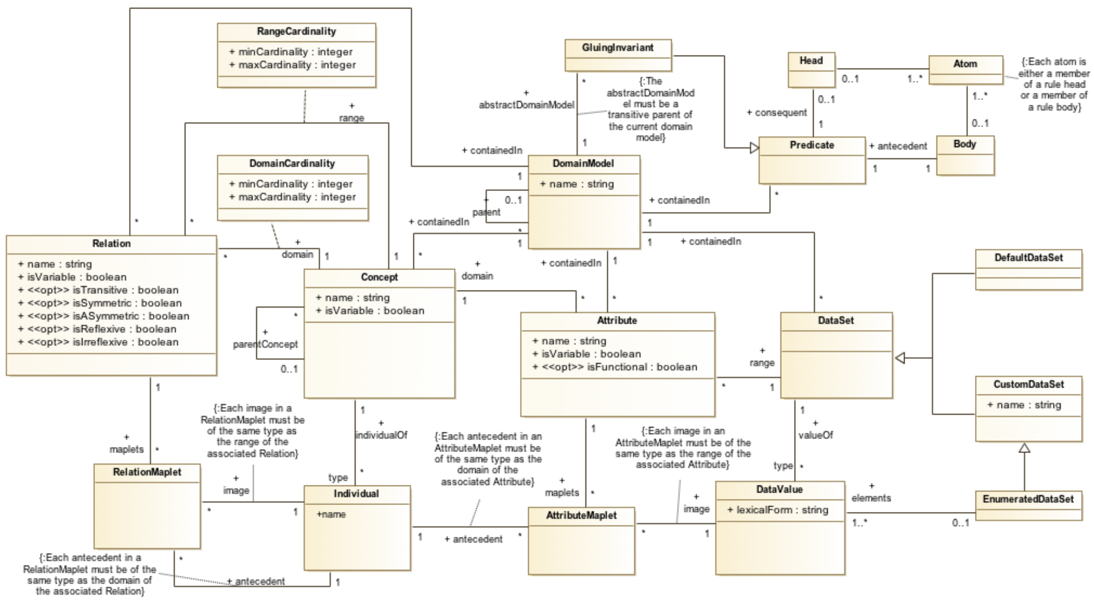


Fig. 5. Metamodel associated with SysML/KAOS domain modeling

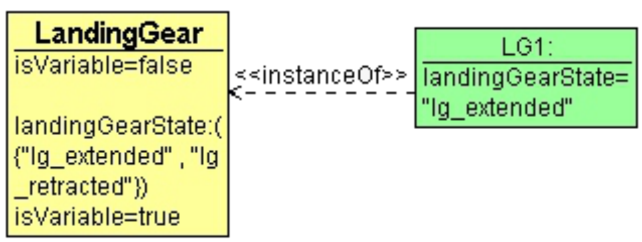


Fig. 6. ***lg_system_ref_0***: ontology associated to the root level of the landing gear goal model

Figure 6 represents the *SysML/KAOS* domain model associated to the root level of the landing gear system goal model of Figure 3, and Figure 7 represents the first refinement level. They are illustrated using the syntax proposed by *OWLGred* [18] and, for readability purposes, we have decided to remove optional characteristics representation. It should be noted that the *individualOf* association is illustrated by *OWLGred* within the figures as a stereotyped link with the tag *«instanceOf»*. The domain model associated to the goal diagram root level is named ***lg_system_ref_0*** and the one associated to the first refinement level is named ***lg_system_ref_1***.

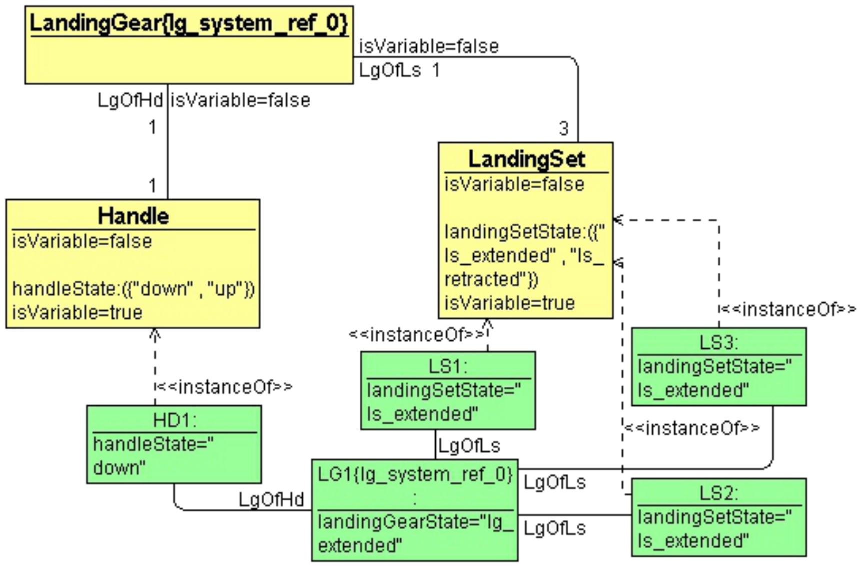


Fig. 7. ***lg_system_ref_1***: ontology associated to the first level of refinement of the landing gear goal model

Each domain model is associated with a level of refinement of the *SysML/KAOS* goal diagram and is likely to have as its parent, through the *parent* association, another domain model. This allows the child domain model to access and extend some elements defined in the parent domain model. For example, in *lg_system_ref_1* (Fig. 7), elements defined in *lg_system_ref_0* (Fig. 6) are imported and reused.

A *concept* (instance of metaclass *Concept* of Figure 5) represents a group of individuals sharing common characteristics. It can be declared *variable* (isVariable=true) when the set of its individuals is likely to be updated through addition or deletion of individuals. Otherwise, it is considered to be *constant* (isVariable=false). A concept may be associated with another, known as its parent concept, through the parentConcept association, from which it inherits properties. For example, in *lg_system_ref_0* (Fig. 6), a *landing gear* is modeled as an instance of Concept named *"LandingGear"*. Since it is impossible to dynamically add or remove a landing gear, the attribute isVariable of ***LandingGear*** is set to *false*. *LG1* is modeled as an instance of Individual (Fig. 5) named *"LG1"* individual of ***LandingGear***.

Instances of Relation are used to capture links between concepts, and instances of Attribute capture links between concepts and data sets, knowing that data sets (instances of DataSet) are used to group data values (instances of DataValue) having the same type. The most basic way to build an instance of DataSet is by listing its elements. This can be done through the DataSet specialization called EnumeratedDataSet. A relation or an attribute can be declared *variable* if the list of maplets related to it is likely to change over time. Otherwise, it is considered to be *constant*. Each instance of DomainCardinality (respectively RangeCardinality) makes it possible to define, for an instance of Relation *re*, the minimum and maximum limits of the number of instances of Individual, having the domain (respectively range) of *re* as *type*, that can be put in relation with one instance of Individual, having the range (respectively domain) of *re* as *type*. The following constraint is associated with these limits : $(minCardinality \geq 0) \wedge (maxCardinality = * \vee maxCardinality \geq minCardinality)$, knowing that if $maxCardinality = *$, then the maximum limit is *infinity*. Instances of RelationMaplet are used to define associations between instances of Individual through instances of Relation. In an identical manner, instances of AttributeMaplet are used to define associations between instances of Individual and instances of DataValue through instances of Attribute. Optional characteristics can be specified for a relation : *transitive* (isTransitive, default *false*), *symmetrical* (isSymmetric, default *false*), *asymmetrical* (isASymmetric, default *false*), *reflexive* (isReflexive, default *false*) or *irreflexive* (isIrreflexive, default *false*). Moreover, an attribute can be *functional* (isFunctional, default *true*). For example, in *lg_system_ref_0* (Fig. 6), the possible states of a landing gear is modeled as an instance of Attribute named *"landingGearState"*, having ***LandingGear*** as domain and as range an instance of EnumeratedDataSet containing two instances of DataValue of type STRING: ***"lg_extended"*** for the extended state and ***"lg_retracted"*** for the retracted state. Since it is possible to dynamically change a landing gear state, its isVariable attribute is set to *true*.

The notion of Predicate is used to represent constraints between different elements of the domain model in the form of *Horn clauses*: each predicate has a body which represents its *antecedent* and a head which represents its *consequent*, body and head designating conjunctions of atoms (Fig. 8). A *typing atom* is used to define the type of a term : ConceptAtom for individuals and DataSetAtom for data values. An *association atom* is used to define associations between terms : RelationshipAtom for the connection of two terms through a *relation*, AttributeAtom for the connection of two terms through an *attribute* and DataFunctionAtom for the connection of terms through a *data function*. A *comparison atom* is used to define comparison relationships between terms : EqualityAtom for equality and InequalityAtom for difference. Built in atoms are some specialized atoms, characterized by identifiers captured through the AtomType enumeration, and used for the representation of particular constraints between several terms. For example, an arithmetic constraint between several integer data values.

GluingInvariant, specialization of Predicate, is used to represent links between variables and constants defined within a domain model and those appearing in more abstract domain models, transitively linked to it through the *parent* association. Gluing invariants are extremely important because they capture relationships between abstract and concrete data during refinement which are used to discharge proof obligations. The following gluing invariant is associated with our case study: if there is at least one landing set having the retracted state,

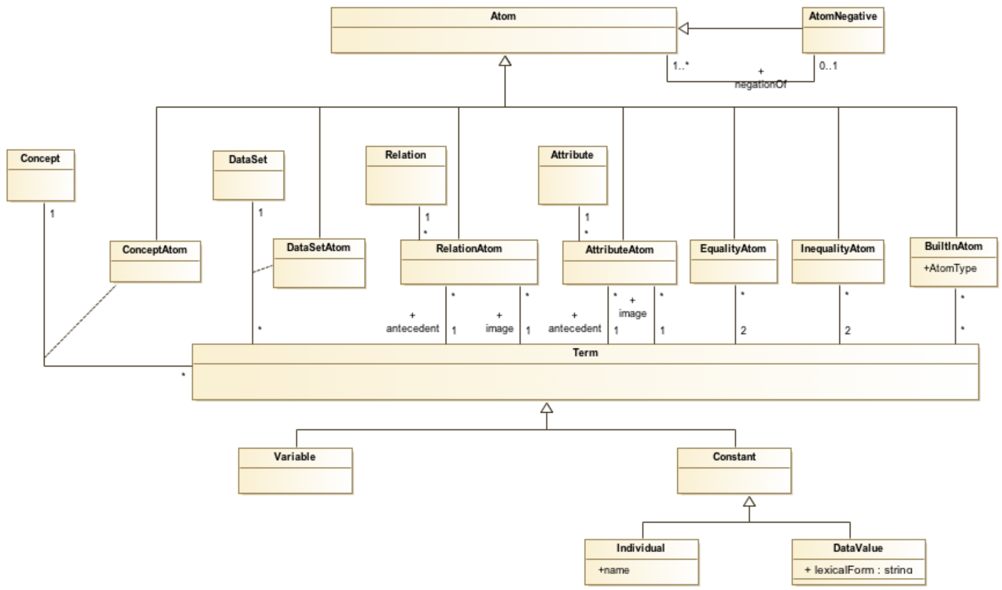


Fig. 8. Extension of the metamodel associated with SysML/KAOS domain modeling for atom specification

then the state of LG1 is retracted

$$\begin{aligned} &landingGearState(LG1, "lg\_retracted") \\ &\quad \leftarrow LandingSet(?\,ls) \wedge landingSetState(?\,ls, "ls\_retracted") \end{aligned} \tag{inv1}$$

An approach for generating an *Event-B* specification from an *OWL* ontology [14] is provided in [3]. The proposed mapping requires the generation of an *ACE (Attempto Controlled English)* version of the *OWL* ontology which serves as the basis for the development of the *Event-B* specification. This is done through a step called *OWL verbalization*. The verbalization method, proposed by [3], transforms *OWL* instances into capitalized proper names, classes into common names, and properties into active and passive verbs. Once the verbalization process has been completed, [3] proposes a set of rules for obtaining the Event-B specification: classes are translated as *Event-B* sets, properties are translated as relations, etc. In addition, [3] proposes rules for the *Event-B* representation of property characteristics and associations between classes or properties. Unfortunately, the proposal makes no distinction between constant and variable : It does not specify when it is necessary to use constants or variables, when it is necessary to express an ontology rule as an invariant or as an axiom. Moreover, the proposal imposes a two-step sequence for the transition from an OWL ontology to an Event-B model, the first step requiring the ontology to be constructed in English. Finally, the approach does not propose anything regarding the referencing from an ontology into another one.

In [13], domain is modeled by defining agents, business entities and relations between them. The paper proposes rules for mapping domain models so designed in *Event-B* specifications : agents are transformed into machines, business entities are transformed into sets, and relations are transformed into *Event-B* variable relations. These rules are certainly sufficient for domain models of interest for [13], but they are very far from covering the extent of *SysML/KAOS* domain modeling formalism.

In [2], domain properties are described through data-oriented requirements for concepts, attributes and associations and through constraint-oriented requirements for axioms. Possible states of a *variable* element are represented using UML state machines. Concepts, attributes and associations arising from data-oriented requirements are modeled as UML class diagrams and translated to Event-B using *UML-B* [15] : nouns and attributes are represented as UML classes and relationships between nouns are represented as UML associations. *UML-B* is also used for the translation of state machines to Event-B variables, invariants and events. Unfortunately, constraints arising from constraint-oriented requirements are modeled using a semi-formal language called *Structured English*, following a method similar to the *Verbalization* approach described in [3] and manually translated to Event-B. Moreover, it is impossible to rely solely on the representation of an element of the class diagram to know if its state is likely to change dynamically. The consequence being that in an Event-B model, the same element can appear as a set, a constant or a variable and its properties are likely to appear both in the *PROPERTIES* and in the *INVARIANT* clauses.

Some rules for passing from an *OWL* ontology representing a domain model to *Event-B* specifications are proposed through a case study in [10]. This case study reveals that each ontology class, having no instance, is modeled as an *Event-B* abstract set. The others are modeled as an enumerated set. Finally, each object property between two classes is modeled as a constant defines as a relation. These rules allow the generation of a first version of an *Event-B* specification from a domain model ontology. Unfortunately, the case study does not address several concerns. For example, object properties are always modeled as constants, despite the fact that they may be variable. Moreover, the case study does not provide any rule for some domain model elements such as datasets or predicates. In the remainder of this paper, we propose to enrich this proposal for a complete mapping of *SysML/KAOS* domain models with *Event-B* specifications.

3 SysML/KAOS Domain Model Formalization

Figures 9 and 10 represents respectively the *B System* specifications associated with the root level of the landing gear system domain model illustrated through Figure 6 and that associated with the first refinement level domain model illustrated through Figure 7.

SYSTEM *lg_system_ref_0*
SETS *LandingGear; DataSet_1= {lg_extended, lg_retracted}*
CONSTANTS *T_landingGearState, LG1*
PROPERTIES
(0.1) $LG1 \in LandingGear$
(0.2) $\wedge\ LandingGear=\{LG1\}$
(0.3) $\wedge\ T\_landingGearState = LandingGear \longrightarrow DataSet\_1$
VARIABLES *landingGearState*
INVARIANT
(0.4) $landingGearState \in T\_landingGearState$
INITIALISATION
(0.5) $landingGearState := \{LG1 \mapsto lg\_extended\}$
EVENTS
●●●
END

Fig. 9. *Formalization of the Root Level of the Landing Gear System Domain Model*

Figures 13, 14 , 15 and 16 are schematizations of correspondence links between domain models and Event-B formal models. Red links represent correspondence links, the part inside the blue rectangle representing the portion of the Event-B metamodel under consideration.

In the following, we describe a set of rules that allow to obtain an *Event-B* specification from domain models associated with refinement levels of a *SysML/KAOS* goal model. They are illustrated using the **B syntax** :

- Regarding the representation of metamodels, we have followed the rules proposed by [15] for the translation of *UML* class diagrams to *B* specifications: for example, 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. Figures 11 and 12 are excerpts from representations of metamodels associated respectively to the SysML/KAOS Domain Modeling method and to the Event-B method. DomainModel, Concept, Relation, Attribute and DataSet of the SysML/KAOS domain metamodel and Component, BSet, LogicFormula and Variable of the Event-B metamodel give rise to abstract sets representing all their possible instances. Variables appear to capture, for each class, all the currently defined instances. Variables are also used to represent attributes and associations such as ParentConcept, Relation_isVariable, Attribute_isFunctional of the SysML/KAOS domain metamodel and Refines of the Event-B metamodel. In case of ambiguity as to the nomenclature of an element, its name is prefixed by that of the class to which it is attached.
- Correspondence links between classes are represented through variables typed as functions having the ***B*** representation of the first class as domain and the ***B*** representation of the second class as range. For example, correspondence links between instances of Concept and instances of AbstractSet illustrated through figure 14, are captured through a variable typed as a function be-

REFINEMENT lg_system_ref_1
REFINES *lg_system_ref_0*
SETS *Handle; LandingSet; DataSet_2={ls_extended, ls_retracted}; DataSet_3={down, up}*
CONSTANTS *T_LgOfHd, LgOfHd, T_LgOfLs, LgOfLs, T_landingSetState, T_handleState, HD1, LS1, LS2, LS3*
PROPERTIES
(1.1) $HD1 \in Handle$
(1.2) $\wedge\ Handle=\{HD1\}$
(1.3) $\wedge\ LS1 \in LandingSet$
(1.4) $\wedge\ LS2 \in LandingSet$
(1.5) $\wedge\ LS3 \in LandingSet$
(1.6) $\wedge\ LandingSet=\{LS1, LS2, LS3\}$
(1.7) $\wedge\ T\_LgOfHd = Handle \leftrightarrow LandingGear$
(1.8) $\wedge\ LgOfHd \in T\_LgOfHd$
(1.9) $\wedge\ \forall\ xx.(xx \in Handle \Rightarrow card(LgOfHd[\{xx\}])=1)$
(1.10) $\wedge\ \forall xx.(xx \in LandingGear \Rightarrow card(LgOfHd^{-1}[\{xx\}])=1)$
(1.11) $\wedge\ LgOfHd = \{HD1 \mapsto LG1\ \}$
(1.12) $\wedge\ T\_LgOfLs = LandingSet \leftrightarrow LandingGear$
(1.13) $\wedge\ LgOfLs \in T\_LgOfLs$
(1.14) $\wedge\ \forall xx.(xx \in LandingSet \Rightarrow card(LgOfLs[\{xx\}])=1)$
(1.15) $\wedge\ \forall xx.(xx \in LandingGear \Rightarrow card(LgOfLs^{-1}[\{xx\}])=3)$
(1.16) $\wedge\ LgOfLs = \{LS1 \mapsto LG1,\ LS2 \mapsto LG1,\ LS3 \mapsto LG1\ \}$
(1.17) $\wedge\ T\_landingSetState = LandingSet \longrightarrow DataSet\_2$
(1.18) $\wedge\ T\_handleState = Handle \longrightarrow DataSet\_3$
VARIABLES *landingSetState, handleState*
INVARIANT
(1.19) $landingSetState \in T\_landingSetState$
(1.20) $\wedge\ handleState \in T\_handleState$
(1.21) $\wedge\ \forall ls.(ls \in LandingSet \wedge landingSetState(ls, ls\_extended) \Rightarrow landingGearState(LG1, lg\_extended))$
INITIALISATION
(1.22) $landingSetState := \{LS1 \mapsto ls\_extended,\ LS2 \mapsto ls\_extended,\ LS3 \mapsto ls\_extended\ \}$
(1.23) $||$ $handleState := \{HD1 \mapsto down\ \}$
EVENTS
●●●
END

Fig. 10. *Formalization of the First Refinement Level of the Landing Gear System Domain Model*

tween Concept and AbstractSet : $correspondenceOf\_Concept\_AbstractSet \in Concept \rightarrowtail AbstractSet$.
- Each rule is represented as an event by following the correspondence links.
- Whereas no additional precision is given, we consider that all *Event-B* content associated with a refinement level is defined within a single component (SYSTEM/REFINEMENT) : it is always possible to separate it into two parts: the context for the static part (SETS, CONSTANTS and PROPER-

SYSTEM *SysMLKAOSDomainMetamodel*
SETS *DomainModel_Set; Concept_Set; Relation_Set; Attribute_Set; DataSet_Set*
VARIABLES *DomainModel, Concept, Relation, Attribute, DataSet, CustomDataSet, EnumeratedDataSet, ParentConcept, Relation_isVariable, Attribute_isFunctional*
INVARIANT
$DomainModel \subseteq DomainModel\_Set$
$\wedge\ Concept \subseteq Concept\_Set$
$\wedge\ Relation \subseteq Relation\_Set$
$\wedge\ Attribute \subseteq Attribute\_Set$
$\wedge\ DataSet \subseteq DataSet\_Set$
$\wedge\ CustomDataSet \subseteq DataSet$
$\wedge\ EnumeratedDataSet \subseteq CustomDataSet$
$\wedge\ Relation\_isVariable \in Relation \longrightarrow BOOL$
$\wedge\ Attribute\_isFunctional \in Attribute \longrightarrow BOOL$
$\wedge\ ParentConcept \in Concept \nrightarrow Concept$
INITIALISATION
$DomainModel := \emptyset$
$||\quad Concept := \emptyset$
$||\quad Relation := \emptyset$
$||\quad Attribute := \emptyset$
$||\quad DataSet := \emptyset$
$||\quad CustomDataSet := \emptyset$
$||\quad EnumeratedDataSet := \emptyset$
$||\quad ParentConcept := \emptyset$
$||\quad Relation\_isVariable := \emptyset$
$||\quad Attribute\_isFunctional := \emptyset$
END

Fig. 11. *Excerpt from the B representation of the SysML/KAOS domain modeling method metamodel*

TIES) and the machine for the dynamic part (VARIABLES, INVARIANT, INITIALIZATION and EVENTS). The correspondences of the elements of a domain model are defined within the Event-B component associated with this domain model.

3.1 Event-B Machines and Contexts

Rule 1: Domain model without parent

Description : correspondence of a domain model not associated to a parent domain model
rule_ 1=
ANY
DM
WHERE
$DM \in DomainModel$

SYSTEM *EventBMetamodel*
SETS *Component_Set; BSet_Set; LogicFormula_Set; Variable_Set*
VARIABLES *Component, BSet, LogicFormula, Variable, Invariant, System, Refinement, Refines*
INVARIANT

$$
\begin{aligned}
& Component \subseteq Component\_Set \\
\wedge\ & BSet \subseteq BSet\_Set \\
\wedge\ & LogicFormula \subseteq LogicFormula\_Set \\
\wedge\ & Variable \subseteq Variable\_Set \\
\wedge\ & Invariant \subseteq LogicFormula \\
\wedge\ & System \subseteq Component \\
\wedge\ & Refinement \subseteq Component \\
\wedge\ & Refines \in Refinement \rightarrowtail Component
\end{aligned}
$$

INITIALISATION

$$
\begin{aligned}
& Component := \emptyset \\
\| \quad & BSet := \emptyset \\
\| \quad & LogicFormula := \emptyset \\
\| \quad & Variable := \emptyset \\
\| \quad & Invariant := \emptyset \\
\| \quad & System := \emptyset \\
\| \quad & Refinement := \emptyset \\
\| \quad & Refines := \emptyset
\end{aligned}
$$

END

Fig. 12. *Excerpt from the B representation of the Event-B method metamodel*

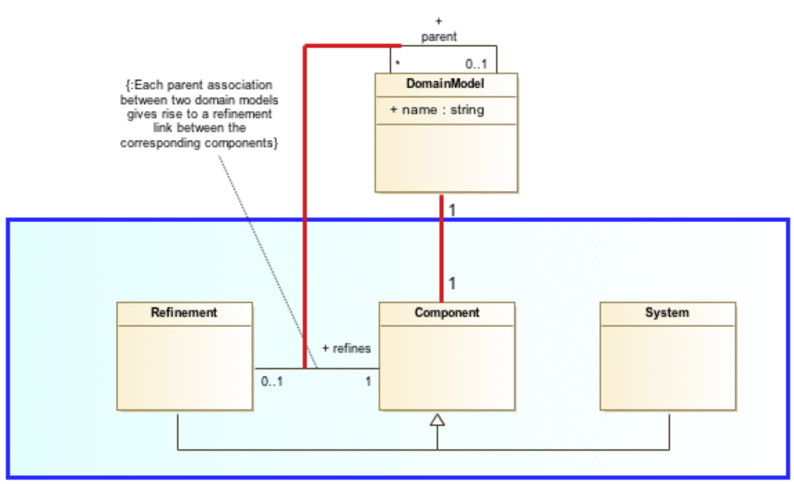


Fig. 13. Correspondence to B System Components

$\wedge\ DM \notin dom(correspondenceOf)$

$\wedge\ DM \notin dom(DomainModel\_Parent)$
THEN
ANY
o_DM
WHERE
$o\_DM \in Component\_Set$
$\wedge\ o\_DM \notin Component$
THEN
$System := System \cup \{o\_DM\}$
$||\ Component := Component \cup \{o\_DM\}$
$||\ correspondenceOf(DM) := o\_DM$
END
END

Any domain model that is not associated with another domain model (Fig. 13), through the *parent* association, gives rise to a system component. This is illustrated in Figure 9 where the root level domain model is translated into a system component named ***lg_system_ref_0***.

Rule 2: Domain model with parent

Description : correspondence of a domain model associated to a parent domain model
rule_ 2$=$
ANY
DM, PDM
WHERE
$DM \in \mathsf{DomainModel}$
$\wedge\ DM \notin dom(correspondenceOf)$
$\wedge\ PDM \in \mathsf{DomainModel}$
$\wedge\ PDM \in dom(correspondenceOf)$
$\wedge\ DomainModel\_Parent(DM) = PDM$
THEN
ANY
o_DM
WHERE
$o\_DM \in Component\_Set$
$\wedge\ o\_DM \notin Component$
THEN
$Refinement := Refinement \cup \{o\_DM\}$
$||\ Component := Component \cup \{o\_DM\}$
$||\ Refines(o\_DM) = correspondenceOf(PDM)$
$||\ correspondenceOf(DM) := o\_DM$
END
END

A domain model associated with another one representing its parent (Fig. 13) gives rise to a refinement component. The refinement component must refine the component corresponding to the parent domain model. This is illustrated in Figure 10 where the first refinement level domain model is translated into a refinement component named ***lg_system_ref_1*** refining ***lg_system_ref_0***.

3.2 Event-B Sets

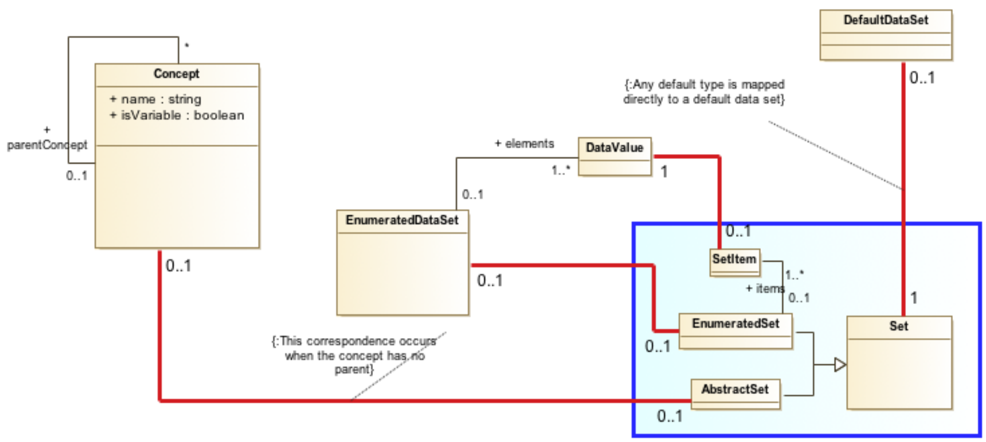


Fig. 14. Correspondence to Sets

Rule 3: Concept without parent

Description : correspondence of a concept not associated to a parent concept
rule_3$=$
ANY
$\quad CO$
WHERE
$\quad CO \in Concept$
$\quad \wedge\ CO \notin dom(correspondenceOf)$
$\quad \wedge\ CO \notin dom(parentConcept)$
THEN
$\quad$**ANY**
$\quad\quad o\_CO$
$\quad$**WHERE**
$\quad\quad o\_CO \in BSet\_Set$

$\wedge\ o\_CO \notin BSet$
THEN
$AbstractSet := AbstractSet \cup \{o\_CO\}$
$||\ Set := Set \cup \{o\_CO\}$
$||\ ConstantBSet := ConstantBSet \cup \{o\_CO\}$
$||\ BSet := BSet \cup \{o\_CO\}$
$||\ correspondenceOf(CO) := o\_CO$
END
END

Any concept that is not associated with another one known as its parent concept (Fig. 14), through the parentConcept association, gives rise to an *Event-B* abstract set. For example, in Figure 9, abstract set named *LandingGear* appears because of Concept instance ***LandingGear***.

Rule 4: Enumerated data set

Description : correspondence of an instance of *EnumeratedDataSet*
rule_4=
ANY
$EDS, (e_i)_{i=1..n}$
WHERE
$EDS \in EnumeratedDataSet$
$\wedge\ \{e_1, e_2, ..., e_n\} \subset DataValue$
$\wedge\ EDS \notin dom(correspondenceOf)$
$\wedge\ elements([\{EDS\}]) = \{e_1, e_2, ..., e_n\}$
THEN
ANY
$o\_EDS, (o\_e_i)_{i=1..n}$
WHERE
$o\_EDS \in BSet\_Set$
$\wedge\ o\_EDS \notin BSet$
$\wedge\ \{o\_e_1, o\_e_2, ..., o\_e_n\} \subset SetItem\_Set)$
$\wedge\ \{o\_e_1, o\_e_2, ..., o\_e_n\} \cap SetItem = \emptyset$
THEN
$EnumeratedSet := EnumeratedSet \cup \{o\_EDS\}$
$||\ Set := Set \cup \{o\_EDS\}$
$||\ ConstantBSet := ConstantBSet \cup \{o\_EDS\}$
$||\ BSet := BSet \cup \{o\_EDS\}$
$||\ correspondenceOf(EDS) := o\_EDS$
$||\ SetItem := SetItem \cup \{o\_e_1, o\_e_2, ..., o\_e_n\}$
$||\ correspondenceOf(e_1) := o\_e_1\ ...\ ||\ correspondenceOf(e_n) := o\_e_n$
$||\ items([\{o\_EDS\}]) := \{o\_e_1, o\_e_2, ..., o\_e_n\}$
END
END

Any instance of CustomDataSet, defined through an enumeration, gives rise to an *Event-B* enumerated set. For example, in Figure 9, the data set ***{"lg_extended",***

"lg_retracted"}, defined in domain model represented in Figure (Fig. 6), gives rise to the enumerated set *DataSet_1={lg_extended, lg_retracted}*.

Any instance of DefaultDataSet is mapped directly to an *Event-B* default data set : NATURAL, INTEGER, FLOAT, STRING or BOOL.

3.3 Event-B Constants

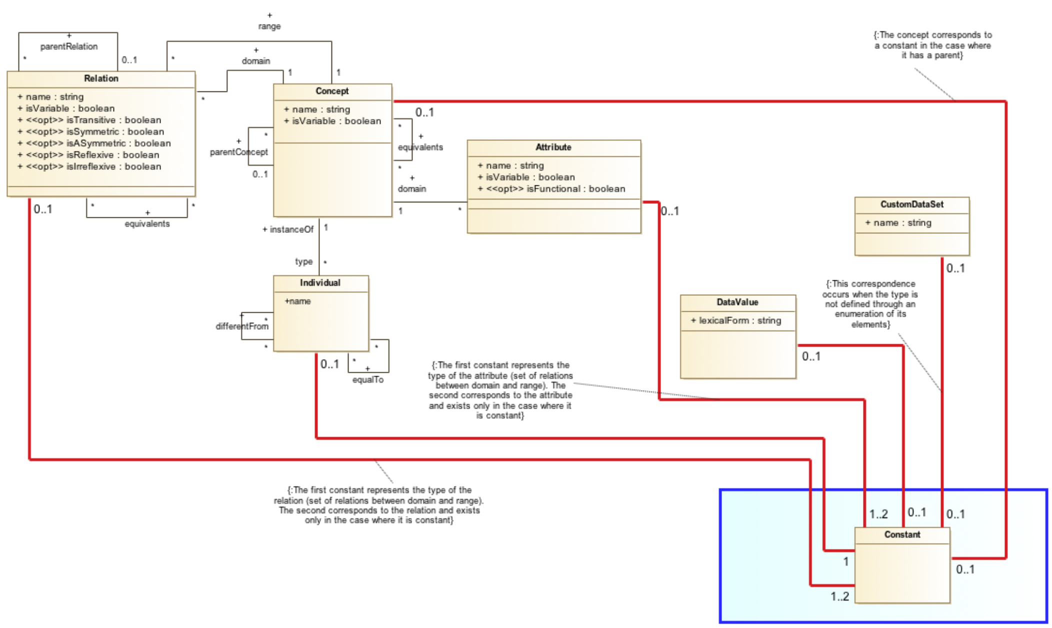


Fig. 15. Correspondence to Constants

Rule 5 : Concept with parent

Description :	correspondence of a concept associated to a parent concept
Parameters :	$Co \in$ Concept ; $PCo \in$ Concept
Constraint :	$Co.parentConcept = PCo$

gives rise to

Output : $o\_Co \in$ CONSTANTS
Property 1 : $o\_Co = correspondenceOf(Co)$
Property 2 : $o\_Co \subseteq correspondenceOf(PCo)$

Any concept associated with another one known as its parent concept (Fig. 13), through the parentConcept association, gives rise to a constant typed as a subset of the *Event-B* element corresponding to the parent concept.

Rule 6 : Constant relation

Description : correspondence of an instance of Relation having its *isVariable* property set to *false*
Parameters : $RE \in$ Relation ; $CO1 \in$ Concept; $CO2 \in$ Concept
Constraint 1 : $RE.domain = CO1$
Constraint 2 : $RE.range = CO2$
Constraint 3 : $RE.isVariable = false$

gives rise to

Output : $o\_RE \in$ CONSTANTS; $T\_RE \in$ CONSTANTS
Property 1 : $o\_RE = correspondenceOf(RE)$
Property 2 : $T\_RE = correspondenceOf(CO1) \leftrightarrow correspondenceOf(CO2)$
Property 3 : $o\_RE \in T\_RE$

Each relation gives rise to a constant representing the type of its associated *Event-B* element and defined as the set of relations between the *Event-B* element corresponding to the relation domain and the one corresponding to the relation range. Moreover, if the relation has its isVariable attribute set to *false*, it is translated through a second constant. This is illustrated in Figure 10 where ***LgOfHd***, for which isVariable is set to *false*, is translated into a constant named *LgOfHd* and having as type ***T_LgOfHd*** defined as the set of relations between ***Handle*** and ***LandingGear*** (assertions *1.7* and *1.8*).

Rule 7 : Constant non-functional attribute

Description : correspondence of an instance of Attribute having its *isVariable* and *isFunctional* properties set to *false*
Parameters : $AT \in$ Attribute ; $CO \in$ Concept; $DS \in$ DataSet
Constraint 1 : $AT.domain = CO$
Constraint 2 : $AT.range = DS$
Constraint 3 : $AT.isVariable = false$
Constraint 4 : $AT.isFunctional = false$

gives rise to

Output : $o\_AT \in$ CONSTANTS; $T\_AT \in$ CONSTANTS **Property 1 :** $o\_AT = correspondenceOf(AT)$ **Property 2 :** $T\_AT = correspondenceOf(CO) \leftrightarrow correspondenceOf(DS)$ **Property 3 :** $o\_AT \in T\_AT$

Rule 8 : Constant functional attribute

Description : correspondence of an instance of Attribute having its *isVariable* property set to *false* and its *isFunctional* property set to *true* **Parameters :** $AT \in$ Attribute ; $CO \in$ Concept; $DS \in$ DataSet **Constraint 1 :** $AT.domain = CO$ **Constraint 2 :** $AT.range = DS$ **Constraint 3 :** $AT.isVariable = false$ **Constraint 4 :** $AT.isFunctional = true$

gives rise to

Output : $o\_AT \in$ CONSTANTS; $T\_AT \in$ CONSTANTS **Property 1 :** $o\_AT = correspondenceOf(AT)$ **Property 2 :** $T\_AT = correspondenceOf(CO) \longrightarrow correspondenceOf(DS)$ **Property 3 :** $o\_AT \in T\_AT$

Similarly to relations, each attribute gives rise to a constant representing the type of its associated *Event-B* element and, in the case when isVariable is set to *false*, to another constant having its name. However, when the isFunctional attribute is set to *true*, the constant representing the type is defined as the set of functions between the *Event-B* element corresponding to the attribute domain and the one corresponding to the attribute range. The *Event-B* element corresponding to the attribute is then typed as a function. For example, in Figure 9, *landingGearState* is typed as a function (assertions *0.3* and *0.4*) since its type is the set of functions between ***LandingGear*** and ***DataSet_1*** (***DataSet_1***={lg_extended, lg_retracted}).

Rule 9 : individual

Description : correspondence of an instance of Individual **Parameters :** $in \in$ Individual; $CO \in$ Concept **Constraint :** $in.type = CO$

gives rise to

Output : $o\_in \in$ CONSTANTS **Property 1 :** $o\_in = correspondenceOf(in)$ **Property 2 :** $o\_in \in correspondenceOf(CO)$

Rule 10 : data value

Description :	correspondence of an instance of DataValue
Parameters :	$dv \in$ DataValue; $DS \in$ DataSet
Constraint :	$dv.type = DS$

gives rise to

Output :	$o\_dv \in$ CONSTANTS
Property 1 :	$o\_dv = correspondenceOf(dv)$
Property 2 :	$o\_dv \in correspondenceOf(DS)$

Finally, each individual (or data value) gives rise to a constant having its name (or with his *lexicalForm* typed as value) and each instance of CustomDataSet, not defined through an enumeration of its elements, unlike ***DataSet_1*** of Figure 9, gives rise to a constant having its name. For example, in Figure 10, the constant named *HD1* is the correspondent of the individual ***HD1***.

3.4 Event-B Variables

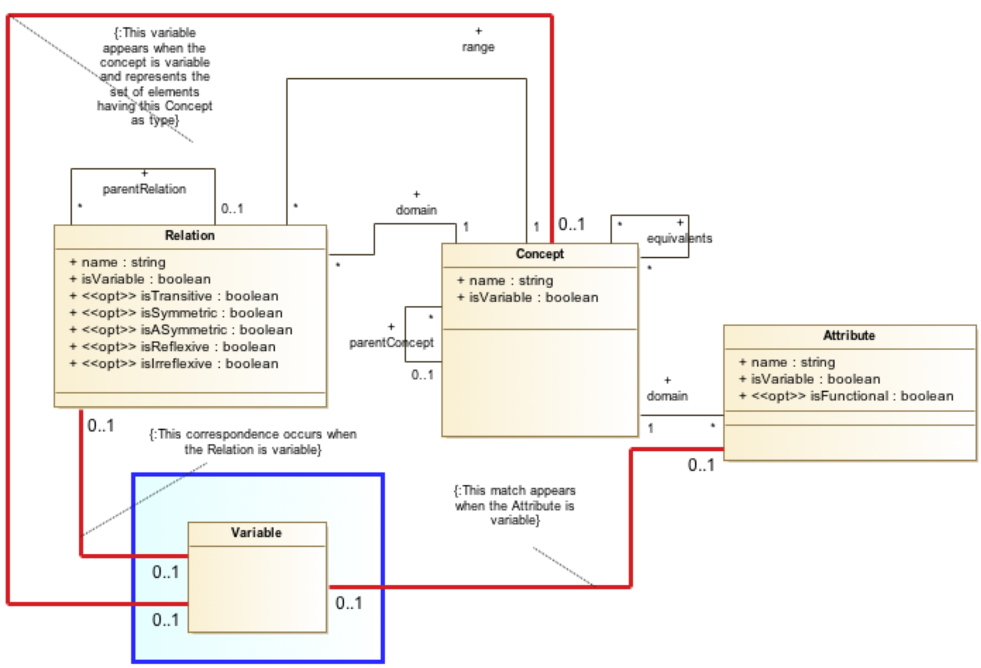


Fig. 16. Correspondence to Variables

Rule 11 : Variable concept

Description : correspondence of an instance of Concept having its *isVariable* property set to *true*
Parameter : $CO \in$ Concept
Constraint : $CO.isVariable = true$

gives rise to

Output : $X\_Co \in$ VARIABLES
Invariant : $X\_CO \subseteq correspondenceOf(CO)$
Description : X_CO represents the set of *Event-B* elements having the correspondence of CO as type.

Rule 12 : variable relation

Description : correspondence of an instance of Relation having its *isVariable* property set to *true*
Parameters : $RE \in$ Relation ; $CO1 \in$ Concept; $CO2 \in$ Concept
Constraint 1 : $RE.domain = CO1$
Constraint 2 : $RE.range = CO2$
Constraint 3 : $RE.isVariable = true$

gives rise to

Output : $o\_RE \in$ VARIABLES; $T\_RE \in$ CONSTANTS
Property 1 : $o\_RE = correspondenceOf(RE)$
Property 2 : $T\_RE = correspondenceOf(CO1) \leftrightarrow correspondenceOf(CO2)$
Invariant : $o\_RE \in T\_RE$

Rule 13 : variable non-functional (respectively functional) attribute

Description : correspondence of an instance of Attribute having its *isVariable* property set to *true* and its *isFunctional* property set to *false* (respectively *true*)
Parameters : $AT \in$ Attribute ; $CO \in$ Concept; $DS \in$ DataSet
Constraint 1 : $AT.domain = CO$
Constraint 2 : $AT.range = DS$
Constraint 3 : $AT.isVariable = true$
Constraint 4 : $AT.isFunctional = false$ (respectively $true$)

gives rise to

Output : $o\_AT \in$ VARIABLES; $T\_AT \in$ CONSTANTS
Property 1 : $o\_AT = correspondenceOf(AT)$
Property 2 : $T\_AT = correspondenceOf(CO) \leftrightarrow correspondenceOf(DS)$

(respectively $T\_AT = correspondenceOf(CO) \longrightarrow correspondenceOf(DS)$)
Invariant : $o\_AT \in T\_AT$

An instance of Relation, of Concept or of Attribute, having its isVariable property set to *true* gives rise to a variable (Fig. 16). For a concept, the variable represents the set of *Event-B* elements having this concept as type. For a relation or an attribute, it represents the set of links between individuals (in case of relation) or between individuals and data values (in case of attribute) defined through it. For example, in Figure 10, variables named *landingSetState* and *handleState* appear because of Attribute instances ***landingSetState*** and ***handleState*** for which the isVariable property is set to *true* (Fig. 7).

3.5 Invariants and Properties

In this section, we are interested in the correspondences between the domain model and the *Event-B* model that are likely to give rise to *invariants* or *properties*. Throughout this section, we will denote by *assertion* any *invariant* or *property*, knowing that an *assertion* is a *property* when it involves only *Event-B* elements corresponding to domain model elements that do not have an attribute *isVariable* or for which the *isVariable* attribute is set to *false* : any other *assertion* is an *invariant*.

The individualOf association between an instance of Individual i and an instance of Concept C is translated through the assertion « $i \in C$ ». For example, in Figure 9, the property $LG1 \in LandingGear$ appears because of the instanceOf association between ***LG1*** and ***LandingGear*** (Fig. 6). The same rule is applicable within the scope of the valueOf association between an instance of DataValue and an instance of DataSet.

Rule 14 : optional characteristics of relations

Rule 14.1 : transitive relation

Description : Assertion raised when an instance of Relation has its *isTransitive* property set to *true*
Parameter : $RE \in$ Relation
Constraint : $RE.isTransitive = true$

gives rise to

ASSERTION : $(correspondenceOf(RE) \; ; \; correspondenceOf(RE)) \subseteq correspondenceOf(RE)$

Rule 14.2 : symmetric relation

Description : Assertion raised when an instance of Relation has its *isSymmetric* property set to *true*
Parameter : $RE \in$ Relation
Constraint : $RE.isSymmetric = true$

gives rise to

ASSERTION : $correspondenceOf(RE)^{-1} = correspondenceOf(RE)$

Rule 14.3 : asymmetric relation

Description : Assertion raised when an instance of Relation has its *isASymmetric* property set to *true*
Parameter : $RE \in$ Relation
Constraint : $RE.isASymmetric = true$

gives rise to

ASSERTION : $(correspondenceOf(RE)^{-1} \cap correspondenceOf(RE)) \subseteq id(correspondenceOf(RE.domain))$

Rule 14.4 : reflexive relation

Description : Assertion raised when an instance of Relation has its *isReflexive* property set to *true*
Parameter : $RE \in$ Relation
Constraint : $RE.isReflexive = true$

gives rise to

ASSERTION : $id(correspondenceOf(RE.domain)) \subseteq correspondenceOf(RE)$

Rule 14.5 : irreflexive relation

Description : Assertion raised when an instance of Relation has its *isIrreflexive* property set to *true*
Parameter : $RE \in$ Relation
Constraint : $RE.isIrreflexive = true$

gives rise to

ASSERTION : $id(correspondenceOf(RE.domain)) \cap correspondenceOf(RE) = \emptyset$

When the isTransitive property of an instance of Relation re is set to *true*, the assertion « $(re\ ;\ re) \subseteq re$ » must appear in the *Event-B* component corresponding to the domain model, knowing that *";"* is the composition operator for relations. For the isSymmetric property, the assertion is « $re^{-1} = re$ ». For the isASymmetric property, the assertion is « $(re^{-1} \cap re) \subseteq id(dom(re))$ ». For the isReflexive property, the assertion is « $id(dom(re)) \subseteq re$ » and for the isIrreflexive property, the assertion is « $id(dom(re)) \cap re = \emptyset$ », knowing that *"id"* is the *identity* function and *"dom"* is an operator that gives the *domain* of a relation (*"ran"* is the operator that gives the *range*).

Rule 15 : domain cardinality

Rule 15.1 : domain cardinality ($maxCardinality \geq 0 \wedge minCardinality \neq maxCardinality$)

Description : Assertion raised when an instance of DomainCardinality associated to an instance of Relation, has its *maxCardinality* greater or equal to *0* and different from its *minCardinality*
Parameters : $dc \in$ DomainCardinality; $RE \in$ Relation
Constraint 1 : $dc.relation = RE$
Constraint 2 : $dc.maxCardinality \geq 0$
Constraint 3 : $dc.minCardinality \neq dc.maxCardinality$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.range) \Rightarrow card(correspondenceOf(RE)^{-1}[\{x\}]) \in dc.minCardinality..dc.maxCardinality)$

Rule 15.2 : domain cardinality ($minCardinality = maxCardinality$)

Description : Assertion raised when an instance of DomainCardinality associated to an instance of Relation, has its *maxCardinality* equal to its *minCardinality*
Parameters : $dc \in$ DomainCardinality; $RE \in$ Relation
Constraint 1 : $dc.relation = RE$
Constraint 2 : $dc.minCardinality = dc.maxCardinality$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.range) \Rightarrow card(correspondenceOf(RE)^{-1}[\{x\}]) = dc.minCardinality)$

Rule 15.3 : domain cardinality ($maxCardinality = *$)

Description : Assertion raised when an instance of DomainCardinality associated to an instance of Relation, has its *maxCardinality* set to *null*
Parameters : $dc \in$ DomainCardinality; $RE \in$ Relation
Constraint 1 : $dc.relation = RE$
Constraint 2 : $dc.maxCardinality = *$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.range) \Rightarrow card(correspondenceOf(RE)^{-1}[\{x\}]) \geq dc.minCardinality)$

Rule 16 : range cardinality

Rule 16.1 : range cardinality ($maxCardinality \geq 0 \wedge minCardinality \neq maxCardinality$)

Description : Assertion raised when an instance of RangeCardinality associated to an instance of Relation, has its *maxCardinality* greater or equal to *0* and different from its *minCardinality*
Parameters : $rc \in$ RangeCardinality; $RE \in$ Relation
Constraint 1 : $rc.relation = RE$
Constraint 2 : $rc.maxCardinality \geq 0$
Constraint 3 : $rc.minCardinality \neq rc.maxCardinality$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.domain) \Rightarrow card(correspondenceOf(RE)[\{x\}]) \in rc.minCardinality..rc.maxCardinality)$

Rule 16.2 : range cardinality ($minCardinality = maxCardinality$)

Description : Assertion raised when an instance of RangeCardinality associated to an instance of Relation, has its *maxCardinality* equal to its *minCardinality*
Parameters : $rc \in$ RangeCardinality; $RE \in$ Relation
Constraint 1 : $rc.relation = RE$
Constraint 2 : $rc.minCardinality = rc.maxCardinality$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.domain) \Rightarrow card(correspondenceOf(RE)[\{x\}]) = rc.minCardinality)$

Rule 16.3 : range cardinality ($maxCardinality = *$)

Description : Assertion raised when an instance of RangeCardinality associated to an instance of Relation, has its *maxCardinality* set to *null*
Parameters : $rc \in$ RangeCardinality; $RE \in$ Relation
Constraint 1 : $rc.relation = RE$
Constraint 2 : $rc.maxCardinality = *$

gives rise to

ASSERTION : $\forall x.(x \in correspondenceOf(RE.domain) \Rightarrow card(correspondenceOf(RE)[\{x\}]) \geq rc.minCardinality)$

An instance of DomainCardinality (respectively RangeCardinality) associated to an instance of Relation *re*, with bounds minCardinality and maxCardinality ($maxCardinality \geq 0$), gives rise to the assertion

$$\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) \in minCardinality..maxCardinality)$$

(respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) \in minCardinality..maxCardinality)$).

When $minCardinality = maxCardinality$, then the assertion is

$$\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) = minCardinality)$$

(respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) = minCardinality)$).

Finally, when $maxCardinality = *$, then the assertion is

$$\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) \geq minCardinality)$$

(respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) \geq minCardinality)$).

For example, in Figure 10, assertions *1.9* and *1.10* appear because of instances of RangeCardinality and DomainCardinality associated to the instance of Relation *LgOfHd* (Fig. 6).

Rule 17 : constant relation (respectively attribute) maplets

Description : Assertion raised due to instances of RelationMaplet (respectively AttributeMaplet) associated to an instance of Relation (respectively Attribute having its *isVariable* property set to *false*
Parameters : $ASSOC \in$ Relation (respectively Attribute) ; $(a_j, i_j)_{j=1..n} / \forall j \in 1..n, (a_j, i_j) \in$ RelationMaplet (respectively $AttributeMaplet$)
Constraint 1 : $ASSOC.maplets = \{(a_1, i_1), (a_2, i_2), ..., (a_n, i_n)\}$
Constraint 2 : $ASSOC.isVariable = false$

gives rise to

PROPERTIES : $correspondenceOf(ASSOC) =$
$\{correspondenceOf(a_1) \mapsto correspondenceOf(i_1), ...,$
$correspondenceOf(a_n) \mapsto correspondenceOf(i_n)\}$

Rule 18 : variable relation (respectively attribute) maplets

Description : substitution raised due to instances of RelationMaplet (respectively AttributeMaplet) associated to an instance of Relation (respectively Attribute having its *isVariable* property set to *true*
Parameters : $ASSOC \in$ Relation (respectively Attribute) ; $(a_j, i_j)_{j=1..n} / \forall j \in 1..n, (a_j, i_j) \in$ RelationMaplet (respectively AttributeMaplet)
Constraint 1 : $ASSOC.maplets = \{(a_1, i_1), (a_2, i_2), ..., (a_n, i_n)\}$
Constraint 2 : $ASSOC.isVariable = true$

gives rise to

INITIALISATION : $correspondenceOf(ASSOC) :=$
$\{correspondenceOf(a_1) \mapsto correspondenceOf(i_1), ...,$
$correspondenceOf(a_n) \mapsto correspondenceOf(i_n)\}$

Instances of RelationMaplet (respectively AttributeMaplet) associated to an instance of Relation (respectively Attribute) *ra* give rise, in the case where the isVariable property of *ra* is set to *false*, to the property «$re = \{an_1 \mapsto im_1, an_2 \mapsto im_2, ..., an_k \mapsto im_k, ..., an_t \mapsto im_t\}$», where an_k designates the instance of Individual linked to the k-th instance of RelationMaplet (respectively AttributeMaplet), through the antecedent association, and im_k designates the instance of Individual (respectively DataValue) linked through the image association. When the isVariable property of *ra* is set to *true*, it is the substitution «$re := \{an_1 \mapsto im_1, an_2 \mapsto im_2, ..., an_k \mapsto im_k, ..., an_t \mapsto im_t\}$» which is rather defined in the *INITIALISATION* clause of the *Event-B* component. For example, in Figure 10, the property *1.11* appears because of the association between ***LG1*** and ***HD1*** through ***LgOfHd*** (Fig. 7). Furthermore, the substitution *1.23* appears in the *INITIALISATION* clause because the *handleState* attribute, for which isVariable is *true*, is set to *down*, for the individual ***HD1*** (through an instance of AttributeMaplet).

Rule 19 : predicates

Description : Assertion raised due to an instance of Predicate
Parameters : $Bo \in$ Body; $He \in$ Head; $(x_i)_{i=1..n} / \forall i \in 1..n, x_i \in$ Variable
Constraint 1 : $\forall i \in 1..n, x_i$ is typed in Bo
Constraint 2 : $Bo.predicate.consequent = He$
Constraint 3 : $\forall i \in 1..n, x_i$ is involved in He

gives rise to

Output : $(o\_x_i)_{i=1..n} / \forall i \in 1..n, o\_x_i \in$ Ident; $o\_Bo \in$ Predicate; $o\_He \in$ Predicate
Property 1 : $o\_Bo = correspondenceOf(Bo)$
Property 2 : $o\_He = correspondenceOf(He)$
Property 3 : $\forall i \in 1..n, o\_x_i = correspondenceOf(x_i)$
ASSERTION : $\forall(o\_x_1, ..., o\_x_n).o\_Bo \Rightarrow o\_He$
Description : o_Bo is the conjunction of *Event-B* predicates corresponding to body atoms and o_He is the conjunction of *Event-B* predicates corresponding to head atoms. If a variable is involved only in the *antecedent* part or in the *consequent* part, then it is existentially quantified (*rule 19.4*).

Any instance of Predicate gives rise to an assertion of the form

$$\forall(x_1, ..., x_n).Bo(x_1, ..., x_n) \Rightarrow He(x_1, ..., x_n)$$

where $(x_i)_{i \in 1..n}$ are variable terms introduced in the predicate, $Bo(x_1, ..., x_n)$ is the conjunction of *Event-B* assertions corresponding to predicate body atoms and $He(x_1, ..., x_n)$ is the conjunction of *Event-B* assertions corresponding to predicate head atoms.

Rule 19.1 : concept (respectively data set) atom

Description : Assertion raised due to an instance of ConceptAtom (respectively DataSetAtom)
Parameters : $CD \in$ Concept (respectively DataSet); $tt \in$ Term; $at \in$ ConceptAtom (respectively DataSetAtom)
Constraint 1 : $at.concept = CD$ (respectively $at.dataset = CD$)
Constraint 2 : $at.term = tt$

gives rise to

ASSERTION : $correspondenceOf(tt) \in correspondenceOf(CD)$

Any instance of ConceptAtom (respectively DataSetAtom) relating an instance of Concept (respectively DataSet) CD and an instance of Term tt gives rise to the assertion « $tt \in CD$ ».

Rule 19.2 : relation (respectively attribute) atom

Description : Assertion raised due to an instance of RelationAtom (respectively AttributeAtom)
Parameters : $RA \in$ Relation (respectively Attribute); $an \in$ Term; $im \in$ Term; $at \in$ RelationAtom (respectively AttributeAtom)
Constraint 1 : $at.relation = RA$ (respectively $at.attribute = RA$)
Constraint 2 : $at.antecedent = an$
Constraint 3 : $at.image = im$

gives rise to

ASSERTION : $(correspondenceOf(an) \mapsto correspondenceOf(im)) \in correspondenceOf(RA)$

Any instance of RelationAtom (respectively AttributeAtom) relating an instance of Relation (respectively Attribute) RA and two instances of Term, an for antecedent and im for image, gives rise to the assertion « $(an \mapsto im) \in RA$ ».

Rule 19.3 : equality (respectively inequality) atom

Description : Assertion raised due to an instance of EqualityAtom (respectively InequalityAtom)
Parameters : $tt_1 \in$ Term; $tt_2 \in$ Term; $at \in$ EqualityAtom (respectively InequalityAtom)
Constraint : $at.terms = \{tt_1, tt_2\}$

gives rise to

ASSERTION : $correspondenceOf(tt_1) = correspondenceOf(tt_2)$ (respectively $correspondenceOf(tt_1) \neq correspondenceOf(tt_2)$)

Any instance of EqualityAtom (respectively InequalityAtom) relating two instances of Term tt_1 and tt_2 gives rise to the assertion « $tt_1 = tt_2$ » (respectively « $tt_1 \neq tt_2$ »).

For instances of BuiltInAtom, the correspondence rules are specific and depend on the value taken by the AtomType enumeration.

Rule 19.4 : existential quantification

Description : Assertion raised due to an instance of Variable which is involved only in the *antecedent* part or in the *consequent* part of the predicate
Parameters : $xx \in$ Variable
Constraint : xx is involved only in the *antecedent* part or in the *consequent* part

gives rise to

Output : $o\_xx \in$ Ident; $o\_Co \in$ Predicate
Property 1 : $o\_xx = correspondenceOf(xx)$
Property 2 : o_Co is the conjunction of Event-B predicates corresponding to atoms involving xx
ASSERTION : $\exists(o\_xx).(o\_Co)$

Rule 19.5 : negation of atoms

Description : Assertion raised due to an instance of AtomNegative **Parameters :** $(A_i)_{i=1..n} / \forall i \in 1..n, A_i \in$ Atom; $at \in$ AtomNegative **Constraint :** $at.negationOf = \{A_1, A_2, ..., A_n\}$

gives rise to

ASSERTION : $not(correspondenceOf(A_1) \wedge correspondenceOf(A_2) \wedge ... \wedge correspondenceOf(A_n))$ **Description :** $(correspondenceOf(A_i))_{i=1..n}$ are Event-B predicates corresponding to $(A_i)_{i=1..n}$

When the predicate is an instance of GluingInvariant, the assertion raised is an Event-B gluing invariant. For example, in Figure 10, assertion *(1.21)* appears because of the gluing invariant (inv1).

3.6 The SysML/KAOS Domain Model Parser Tool

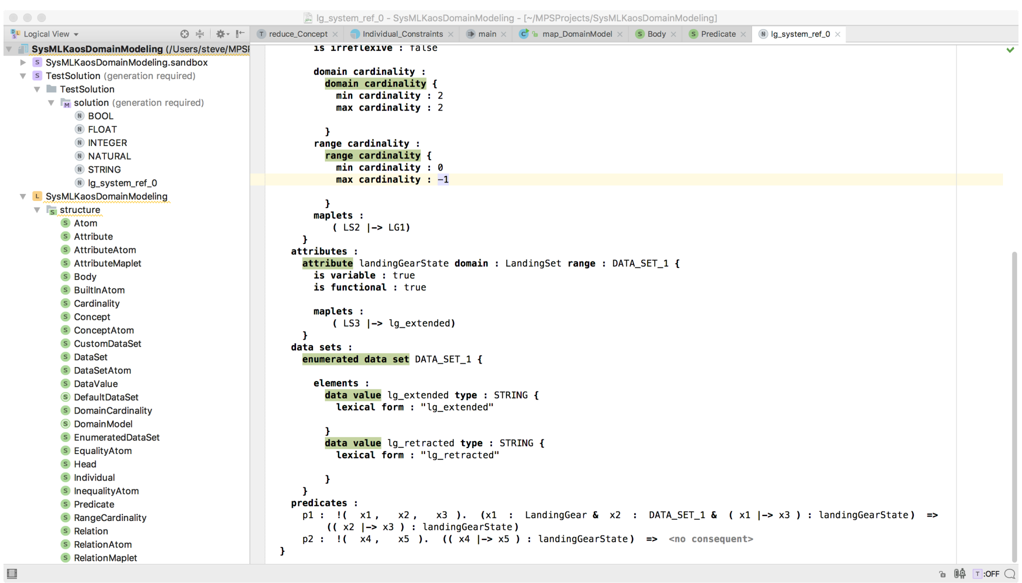
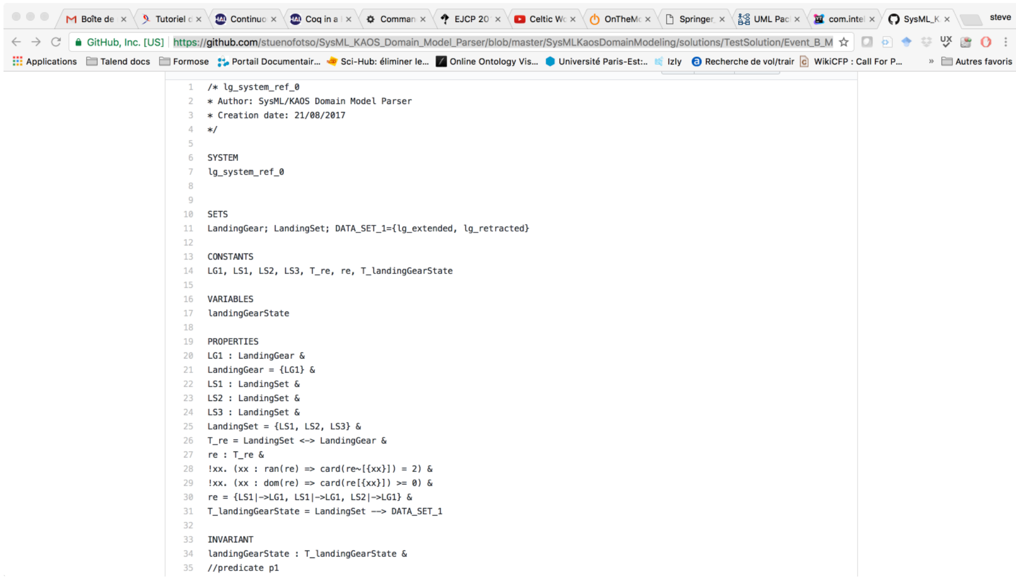


Fig. 17. Preview of the SysML/KAOS Domain Model Parser Tool

The correspondence rules outlined here have been implemented within an open source tool called *SysML/KAOS Domain Model Parser* [16]. It allows the construction of domain models *(Fig. 17)* and generates the corresponding *Event-B* specifications *(Fig. 18)*. It is build through *Jetbrains Meta Programming*



```
/* lg_system_ref_0
* Author: SysML/KAOS Domain Model Parser
* Creation date: 21/08/2017
*/

SYSTEM
lg_system_ref_0


SETS
LandingGear; LandingSet; DATA_SET_1={lg_extended, lg_retracted}

CONSTANTS
LG1, LS1, LS2, LS3, T_re, re, T_landingGearState

VARIABLES
landingGearState

PROPERTIES
LG1 : LandingGear &
LandingGear = {LG1} &
LS1 : LandingSet &
LS2 : LandingSet &
LS3 : LandingSet &
LandingSet = {LS1, LS2, LS3} &
T_re = LandingSet <-> LandingGear &
re : T_re &
!xx. (xx : ran(re) => card(re~[{xx}]) = 2) &
!xx. (xx : dom(re) => card(re[{xx}]) >= 0) &
re = {LS1|->LG1, LS1|->LG1, LS2|->LG1} &
T_landingGearState = LandingSet --> DATA_SET_1

INVARIANT
landingGearState : T_landingGearState &
//predicate p1
```

Fig. 18. Preview of *B System* Specifications Generated by the SysML/KAOS Domain Model Parser Tool for the Landing Gear System Case Study

System [8], a tool to design domain specific languages using language-oriented programming.

4 Conclusion and Future Works

This paper was focused on a presentation of mapping rules between SysML/KAOS domain models and Event-B specifications illustrated through a case study dealing with a landing gear system. The specifications thus obtained can also be seen as a formal semantics for SysML/KAOS domain models. They complement the formalization of the SysML/KAOS goal model by providing a description of the state of the system.

Work in progress is aimed at integrating our approach, implemented through the *SysML/KAOS Domain Model Parser* tool, within the open-source platform *Openflexo* [12] and at evaluating the impact of updates on Event-B specifications on domain models.

Acknowledgment

This work is carried out within the framework of the *FORMOSE* project [4] funded by the French National Research Agency (ANR).

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. ANR-14-CE28-0009: Formose ANR project (2017), http://formose.lacl.fr/
5. Boniol, F., Wiels, V.: The landing gear system case study. pp. 1–18. ABZ, Springer (2014)
6. ClearSy: Atelier B: B System (2014), http://clearsy.com/
7. Gnaho, C., Semmak, F.: Une extension SysML pour l'ingénierie des exigences dirigée par les buts. In: 28e Congrès INFORSID, France. pp. 277–292 (2010)
8. Jetbrains: Jetbrains mps (2017), https://www.jetbrains.com/mps/
9. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
10. 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)
11. 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)
12. Openflexo: Openflexo project (2015), http://www.openflexo.org
13. Poernomo, I., Umarov, T.: A mapping from normative requirements to Event-B to facilitate verified data-centric business process management. CEE-SET LNCS, vol. 7054, pp. 136–149. Springer (2009)
14. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)
15. Snook, C., Butler, M.: UML-B: Formal Modeling and Design Aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (Jan 2006)
16. Tueno, S.: SysML/KAOS Domain Model Parser (2017), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser
17. 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)
18. UL, I.: Owlgred home (2017), http://owlgred.lumii.lv/