Translation Rules from Ontology-based Domain Models to B System Specifications

Steve Tueno and Marc Frappier
GRIL
Université de Sherbrooke
Sherbrooke, QC J1K 2R1, Canada
Email: Steve.Jeffrey.Tueno.Fotso@USherbrooke.ca
Marc.Frappier@USherbrooke.ca

Régine Laleau
LACL
Université Paris-Est Créteil
94010, CRÉTEIL, France
Email: laleau@u-pec.fr

Amel Mammar
SAMOVAR-CNRS
Télécom SudParis
91000, Evry, France
Email: amel.mammar@telecom-sudparis.eu

***Abstract*—Nowadays, the usefulness of a formal language for ensuring the consistency of requirements is well established, especially for critical systems. In our work, we use a combination of the *SysML/KAOS* requirements engineering method, an extension of *SysML*, with concepts of the *KAOS* goal model, and of the *B System* formal method. Translation rules from a goal model to a formal specification have already been defined. They allow to obtain a skeleton of the formal specification. To complete it, we have defined a language, based on ontologies, to express the domain model associated to the goal model. The translation of this domain model gives the structural part of the formal specification. The contribution of this paper is the description of these rules, based on the definition of the metamodels of the SysML/KAOS domain modeling language and of the *B System* specification language. We also present a summary of the formal verification of these rules and we describe an open source tool that implements the languages and the rules. Finally, we provide a brief review of the application of the approach on case studies such as for the formal specification of the *hybrid ERTMS/ETCS level 3* standard.**

Index Terms*—Domain Modeling, Ontologies, *B System*, Requirements Engineering, *SysML/KAOS*, *Event-B

I. Introduction

Our study is part of the *FORMOSE* project [1] and focuses on the formal requirements modeling of systems in critical areas such as railway or aeronautics. In [2], we have defined translation rules to produce formal specifications from *SysML/KAOS* goal models [3], [4]. Nevertheless, the generated specification does not contain the system state. This is why in [4], we have presented the use of ontologies and *UML* class and object diagrams for domain properties representation. Unfortunately, the proposed approach is case study oriented and is therefore interested only in few elements of domain modeling. Furthermore, it involves several modeling languages for the representation of the domain knowledge. We have therefore proposed in [5] a language for domain knowledge representation through ontologies based on *OWL* [6] and *PLIB* [7]. It meets the shortcomings of [4].

This paper is specifically concerned with establishing correspondence links between this domain modeling language and *B System*, a syntactic variant of *Event-B* [8] supported by the *Atelier B* tool which is developed by *ClearSy* [9], an industrial partner in the *FORMOSE* project. *B System* has been chosen because it involves intuitive mathematical concepts and has a powerful refinement logic. The proposed approach allows a high-level modeling of domain properties by encapsulating the difficulties inherent in the manipulation of formal specifications. This enables the expression of more precise and complete properties. The contribution of the paper is the description of the translation rules, based on the definition of the metamodels of the SysML/KAOS domain modeling language and of the *B System* specification language. The rules have been formally specified and verified using *Event-B*. We have, for instance, proved that they are consistent and structure preserving, by discharging the proof obligations and by proving various isomorphisms between the domain model and the *B System* model. The full *Event-B* specification can be found in [10], [11]. We present in this paper a summary of the formal verification process. We also describe an open source tool [12] that implements the languages and the rules, built through *Jetbrains MPS* [13]. It can be used to construct domain models and to automatically translate them into *B System* specifications. The approach, supported by the tool, has been used for the specification of a landing gear system [14], of a localisation software component for the *Cycab* vehicle [15] and of the *hybrid ERTMS/ETCS level 3* standard [16]. The obtained models can be found in [17]–[19]. 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 the approach and we provide a brief review of the work done for the formal specification of the hybrid ERTMS/ETCS level 3 standard.

The remainder of this paper is structured as follows: Section 2 briefly describes the SysML/KAOS requirements engineering method, the *Event-B* and *B System* formal methods, the formalisation of SysML/KAOS goal models and the SysML/KAOS domain modeling language. Follows a presentation, in Section 3, of the relevant state of the art on the expression of domain knowledge using formal methods. In Section 4, we describe and illustrate our translation rules between domain model ontologies and *B System* specifications and we provide an overview of the domain modeling tool. Section 5 discusses the formal

French National Research Agency (ANR)
Natural Sciences and Engineering Research Council of Canada (NSERC)

verification of rules and outlines the work done for the formal specification of the hybrid ERTMS/ETCS level 3 standard. Finally, Section 6 reports our conclusions and discusses our future work.

II. Context

A. SysML/KAOS

Requirements engineering focuses on elicitation, analysis, verification and validation of requirements. The *KAOS* method [20] proposes to represent the requirements in the form of goals described 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 hierarchy of goals 1 and the **object model** which uses the *UML* class diagram for the representation of the domain vocabulary. The hierarchy is built through a succession of refinements using two main 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.

KAOS offers no mechanism to maintain a strong traceability between requirements and design deliverables, making it difficult to validate them against the needs formulated. *SysML/KAOS* [4], [21] addresses this issue by adding to KAOS the *SysML* UML profile specially designed by the Object Management Group (OMG) for the analysis and specification of complex systems. *SysML* allows for the capturing of requirements and the maintaining of traceability links between those requirements and design deliverables, but it does not define a precise syntax for requirements specification. SysML/KAOS therefore proposes to extend the SysML metamodel in order to allow the representation of requirements using the KAOS expressivity.

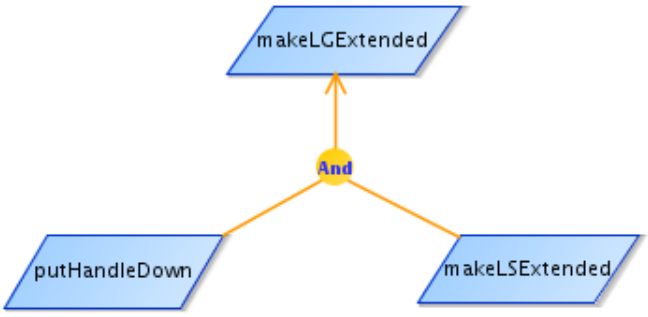

Fig. 1. Excerpt from the landing gear system goal diagram

Our case study deals with the landing gear system of an aircraft which can be retracted (respectively extended) through a handle [14]. 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 system.

B. Event-B and B System

Event-B is an industrial-strength formal method for *system modeling* [8]. It is used to incrementally construct a system specification, using refinement, and to prove properties. 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 axioms. The machine contains variables, invariants and events. A machine can refine another machine, a context can extend others contexts and a machine can see contexts. Gluing invariants are invariants that capture links between variables defined within a machine and those appearing in more abstract ones. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [1], and supported by *Atelier B* [9].

Figure 2 is an excerpt from a metamodel of the *B System* language. A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. A component defining static elements corresponds to an *Event-B* context and the one defining dynamic elements corresponds to a machine. Thus, 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 updated through events. Although it is advisable to always isolate the static and dynamic parts, it is possible to define the two parts within the same component. In the following sections, our *B System* models will be presented using this facility.

C. Formalisation of SysML/KAOS Goal Models

The formalisation of SysML/KAOS goal models is the focus of the work done by [2]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between formal components, [2] has defined new proof obligations to express it. They complete the classic proof obligations for invariant preservation and for event feasibility.

Nevertheless, the generated specification does not contain the system state, composed of variables, constrained by an invariant, and constants, constrained by properties. They must be manually provided. The objective of our study is to automatically derive the system state elements starting from SysML/KAOS domain models.

D. The SysML/KAOS Domain Modeling Language

Domain models in SysML/KAOS are represented using ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [5], [22], built based on

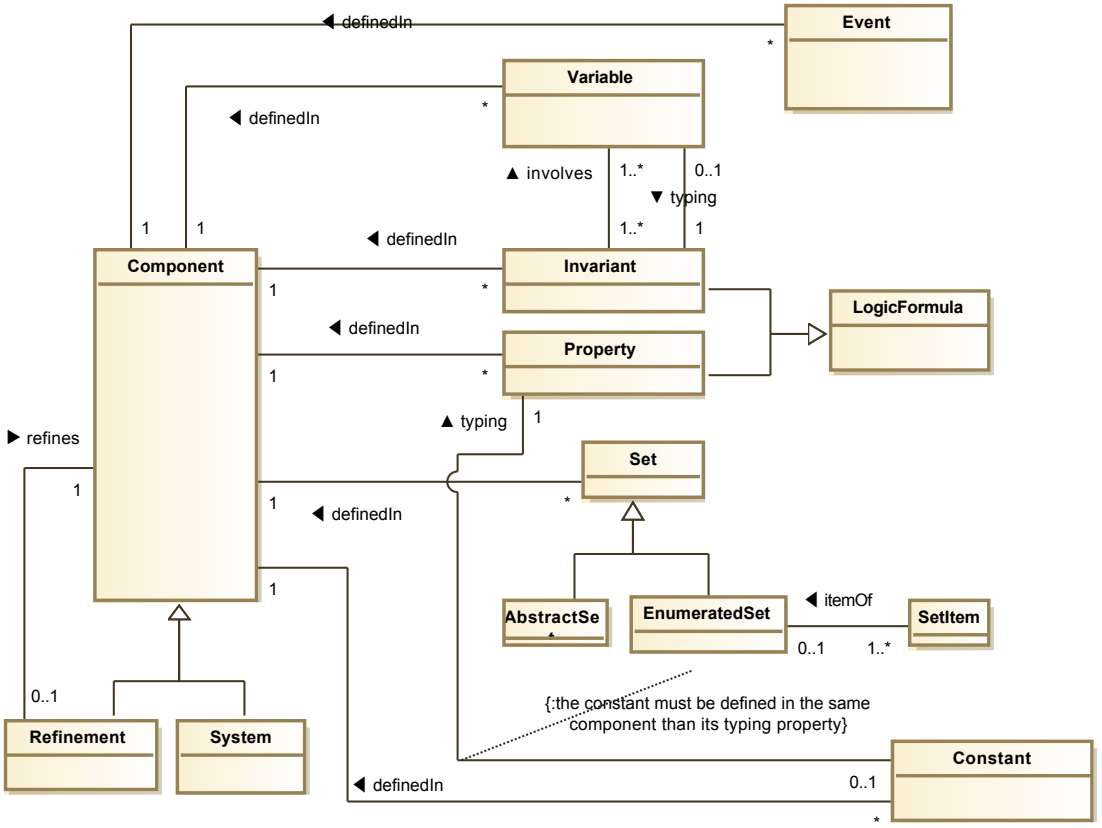

Fig. 2. Excerpt from a metamodel of the *B System* specification language

OWL [6] and *PLIB* [7], two well-known and complementary ontology modeling formalisms.

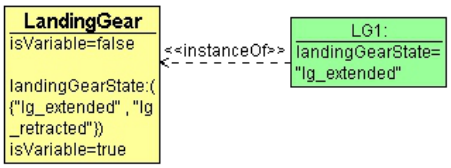

Fig. 3. ***lg_system_ref_0***: ontology associated with the root level of the landing gear goal model

Figure 3 represents the SysML/KAOS domain model associated with the root level of the landing gear system goal model of Fig. 1, and Fig. 4 represents the one associated with the first refinement level. They are illustrated using the syntax proposed by *OWLGred* [23] and, for readability purposes, we have decided to hide optional characteristics representation. It should be noted that the *individualOf* association is illustrated, through *OWLGred*, as a stereotyped link with the tag *«instanceOf»*.

Figure 5 is an excerpt from the metamodel associated with the SysML/KAOS domain modeling language. 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. For example, the domain model `lg_system_ref_1` (Fig. 4)

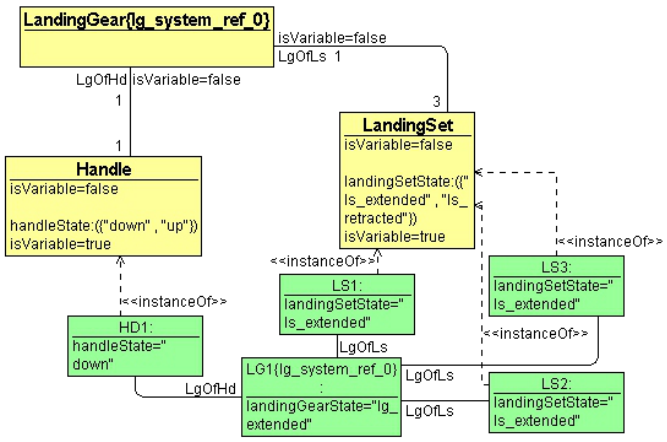

Fig. 4. ***lg_system_ref_1***: ontology associated with the first refinement level of the landing gear goal model

refines `lg_system_ref_0` (Fig. 3). We use the notion of *concept* (instance of Concept) to designate an instantiable universal or a collection of individuals with common properties. A *concept* can be declared *variable* (*isVariable=TRUE*) when the set of its individuals can be updated by adding or deleting individuals.. Otherwise, it is considered to be *constant* (*isVariable=FALSE*). For example, in `lg_system_ref_0`, the

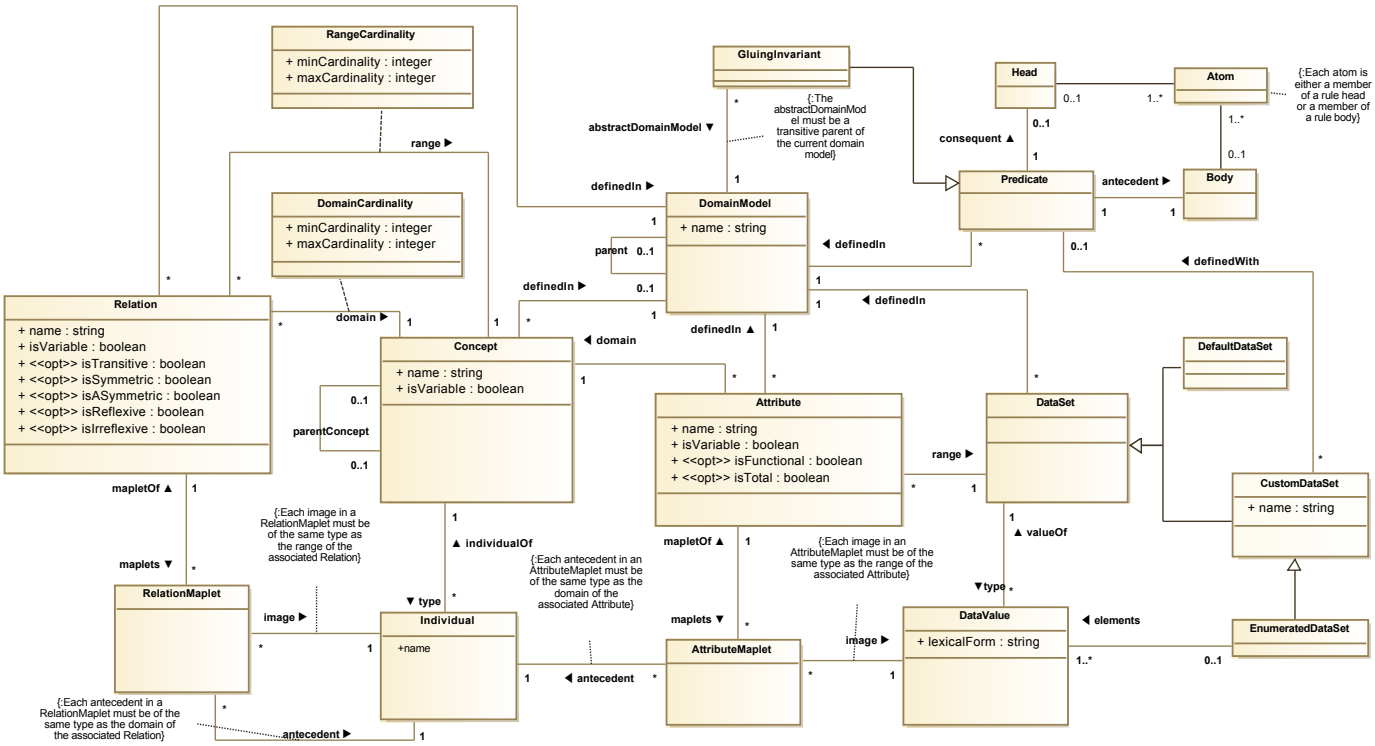

Fig. 5. Metamodel associated with the SysML/KAOS domain modeling language

landing gear entity is modeled as an instance of Concept named `LandingGear`. As in the case study adding or deleting a landing gear is not considered, the property isVariable of `LandingGear` is set to `false`. Instances of Relation are used to capture links between concepts, and instances of Attribute capture links between concepts and data sets. 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*. For example, the possible states of a landing gear are modeled by 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. Its `isVariable` property is set to *true*, since it is possible to dynamically change the state of a landing gear. Furthermore, the association between landing sets and landing gears, in `lg_system_ref_1`, is modeled as an instance of Relation named `LgOfLs`. Since the association of a landing set to a landing gear cannot be changed dynamically, the property isVariable of `LgOfLs` is set to `false`.

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. For example, in `lg_system_ref_1`, the instance of DomainCardinality associated with `LgOfLs` has its minCardinality and maxCardinality properties set to *1*. Instances of RelationMaplet are used to define associations between instances of Individual through instances of Relation. Instances of AttributeMaplet play the same role for attributes. For example, in `lg_system_ref_1`, there are three instances of RelationMaplet to model the association of the landing gear `LG1` to the landing sets `LS1`, `LS2` and `LS3`, each having as image `LG1` and as antecedent the corresponding `LandingSet` individual.

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. A data set can be declared abstractly, as an instance of CustomDataSet, and defined with a predicate. 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 and 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, then the state of LG1 is retracted

$$landingGearState(LG1, "lg\_retracted") \leftarrow LandingSet(?ls) \wedge landingSetState(?ls, "ls\_retracted") \quad \text{(inv1)}$$

III. Existing Approaches for the Formalization of Domain Models

In [24], an approach is proposed for the automatic extraction of domain knowledge, as *OWL* ontologies, from *Z/Object-Z (OZ)* models [25] : OZ types and classes are transformed into OWL classes. Relations and functions are transformed into OWL properties, with the *cardinality* restricted to *1* for total functions and the *maxCardinality* restricted to *1* for partial functions. OZ constants are translated into OWL individuals. Rules are also proposed for subsets and state schemas. A similar approach is proposed in [26], for the extraction of *DAML* ontologies [27] 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, we work on the opposite direction: each ontology refinement level is used to generate a formal model component and links between domain models give links between formal components.

In [28], domain is modeled by defining agents, business entities and relations between them. The paper proposes rules to translate domain models so designed into *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 [28], but they are very far from covering the extent of the SysML/KAOS domain modeling language.

Some rules for passing from an *OWL* ontology representing a domain model to *Event-B* specifications are proposed in [29], in [30] and through a case study in [4]. In [30], the proposed rules 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 transforms OWL individuals into capitalized proper names, classes into common names, and properties into active and passive verbs. Once the verbalization process has been completed, [30] proposes a set of rules for obtaining the *Event-B* specification: classes are translated to *Event-B* sets, properties are translated to relations, etc. In [29], 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* [31]: 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. The approaches in [29] and [30] require a manual transformation of the ontology before the possible application of translation rules to obtain the formal specifications: In [29], it is necessary to convert OWL ontologies into UML diagrams. In [30], the proposal requires the generation of a controlled English version of the *OWL* ontology. Furthermore, 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. In [4], the case study reveals three translation rules: each ontology class, having no individual, is modeled as an *Event-B* abstract set. If the class has individuals, then it is modeled as an enumerated set. Finally, each object property between two classes is modeled as a constant defined as a relation. Several shortcomings are common to these approaches: the provided rules do not take into account the refinement links between model parts. Furthermore, they have not been implemented or formally verified and they are provided in an informal way that does not allow the assesment of their quality and consistency. Finally, the approaches are far from covering the extent of the SysML/KAOS domain modeling language and they are only interested in static domain knowledge (they do not distinguish what gives rise to formal constants or variables).

Several works have been done on the translation of UML diagrams into B specifications such as [31], [32]. They have obviously inspired many of our rules, like those dealing with the translation of classes (concepts) and of associations (attributes and relations). 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. Since our domain models are associated with SysML/KAOS goal model refinement levels, the hierarchy between domain models is converted into refinement links between formal components. Moreover, the predicates linking the elements of concrete models to those of abstract models give gluing invariants. Taking into account links between models guarantees a better scalability, readability and reusability of rules and models. Finally, in the case of the SysML/KAOS domain modeling language, the changeability properties (properties characterising the belonging of an element to the static or dynamic knowledge, materialised with the isVariable property in classes Concept, Relation and Attribute) are first-class citizens, as well as association characteristics (such as isTransitive of the class Relation and isFunctional or the class Attribute), in order to

produce a strongly expressive formal specification. As a result, they are explicitly represented.

IV. Translation Rules from Domain Models to B System Specifications

In the following, we describe a set of rules that allow to obtain a formal specification from domain models associated with refinement levels of a SysML/KAOS goal model. The rules are fully described in [10], [11].

Table I summarises the translation rules, 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 the *abstract* qualifier is used for "without parent".

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) $landingGearState := \{LG1 \mapsto lg\_extended\ \}$
END

Fig. 6. Formalization of the Root Level of the Landing Gear System Domain Model

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) $||\ \ handleState := \{HD1 \mapsto down\ \}$
END

Fig. 7. Formalization of the First Refinement Level of the Landing Gear System Domain Model

Figures 6 and 7 represent respectively the *B System* specifications associated with the root level of the landing gear system domain model illustrated in Fig. 3 and that associated with the first refinement level domain model illustrated in Fig. 4.

A. Generation of B System Components

Any domain model that is not associated with another domain model through the *parent* association, gives a System component (line 1 of Table I). This is illustrated in Fig. 6 where the root level domain model is translated into a system named **`lg_system_ref_0`**.

A domain model associated with another one representing its parent gives a Refinement component (line 2 of Table I). This component refines the one corresponding to the parent domain model. This is illustrated in Fig. 7 where the first refinement level domain model is translated into a refinement named **`lg_system_ref_1`** refining **`lg_system_ref_0`**.

B. Generation of B System Sets

Any concept that is not associated with another one through the parentConcept association, gives an abstract set (line 3 of Table I). For example, in Fig. 6, abstract set named **`LandingGear`** appears because of Concept instance `LandingGear`.

Any instance of CustomDataSet, defined through an enumeration (instance of EnumeratedDataSet), gives a *B System* enumerated set. Otherwise, if it is defined with an instance of Predicate `P`, then it gives a constant for which the typing axiom is the result of the translation of `P`. Finally, it gives an abstract set if no typing predicate is provided. For example, in Fig. 6, the data set `{lg_extended, lg_retracted}`, defined in Fig. 3, gives the enumerated set **`DataSet_1={lg_extended, lg_retracted}`**.

Any instance of DefaultDataSet is mapped directly to a *B System* default set: NATURAL, INTEGER, FLOAT, STRING or BOOL.

C. Generation of B System Constants

Any concept associated with another one through the parentConcept association, gives a constant typed as a subset of the *B System* element corresponding to the parent concept (line 4 of Table I).

Each relation gives a *B System* constant representing the type of its corresponding element and defined as the set of relations between the *B System* element corresponding to the relation domain and the one corresponding to the relation range. Moreover, if the relation has its isVariable property set

TABLE I
SUMMARY OF THE TRANSLATION RULES

	Domain Model		B System	
Translation Of	**Element**	**Constraint**	**Element**	**Constraint**
Abstract domain model	DM	$DM \in$ DomainModel *DM* is not associated with a parent domain model	o_DM	$o\_DM \in$ System
Domain model with parent	DM PDM	$\{DM, PDM\} \subseteq$ DomainModel *DM* is associated with *PDM* through the *parent* association and *PDM* has already been translated	o_DM	$o\_DM \in$ Refinement *o_DM* refines *o_PDM*
Abstract concept	CO	$CO \in$ Concept *CO* is not associated with a parent concept	o_CO	$o\_CO \in$ AbstractSet
Concept with parent	CO PCO	$\{CO, PCO\} \subseteq$ Concept *CO* is associated with *PCO* through the *parentConcept* association and *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 $CO1$ is the *domain* of *RE* $CO2$ is the *range* of *RE* *CO1* and *CO2* have already been translated	o_RE	**IF** the *isVariable* property of *RE* is set to $FALSE$ **THEN** $o\_RE \in$ Constant **ELSE** $o\_RE \in$ Variable **END** $o\_RE \in o\_CO1 \leftrightarrow o\_CO2$ As usual, this relation becomes a *function*, an *injection*, ... according to the cardinalities of *RE*.
Attribute	AT CO DS	$CO \in$ Concept $DS \in$ DataSet $AT \in$ Attribute CO is the *domain* of *AT* DS is the *range* of *AT* *CO* and *DS* have already been translated	o_AT	**IF** the *isVariable* property of *AT* is set to $FALSE$ **THEN** $o\_AT \in$ Constant **ELSE** $o\_AT \in$ Variable **END** **IF** *isFunctional* and *isTotal* are set to $TRUE$ **THEN** $o\_AT \in o\_CO \rightarrow o\_DS$ **ELSE IF** *isFunctional* is set to $TRUE$ **THEN** $o\_AT \in o\_CO \nrightarrow o\_DS$ **ELSE** $o\_AT \in o\_CO \leftrightarrow o\_DS$ **END**
Concept changeability	CO	$CO \in$ Concept the *isVariable* property of *CO* is set to $TRUE$ *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 Ind is an individual of *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 Dva is a value of *DS* *DS* has already been translated	o_Dva	$o\_Dva \in$ Constant $o\_Dva \in o\_DS$
Relation transitivity	RE	$RE \in$ Relation the *isTransitive* property of *RE* is set to $TRUE$ *RE* has already been translated		$(o\_RE \ ; \ o\_RE) \subseteq o\_RE$
Relation reflexivity	RE CO	$RE \in$ Relation $Relation\_isReflexive(RE) = TRUE$ $Relation\_domain\_Concept(RE) = CO$ *RE* and *CO* have already been translated		$id(o\_CO) \subseteq o\_RE$ (All other optional properties of an instance of Relation are translated in the same way [11])
Relation maplets	RE $(M_j)_{j=1..n}$ $(a_j, i_j)_{j=1..n}$	$RE \in$ Relation $(M_j)_{j=1..n}$ are *maplets* of *RE* $\forall j \in 1..n, a_j$ is the antecedent of M_j $\forall j \in 1..n, i_j$ is the image of M_j *RE* and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** the *isVariable* property of *RE* is set to $FALSE$ **THEN** $o\_RE = \{(o\_a_j, o\_i_j)_{j=1..n}\}$ (Property) **ELSE** $o\_RE := \{(o\_a_j, o\_i_j)_{j=1..n}\}$ (Initialisation) **END**
Attribute maplets	AT $(M_j)_{j=1..n}$ $(a_j, i_j)_{j=1..n}$	$AT \in$ Attribute $(M_j)_{j=1..n}$ are *maplets* of *AT* $\forall j \in 1..n, a_j$ is the antecedent of M_j $\forall j \in 1..n, i_j$ is the image of M_j *AT* and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** the *isVariable* property of *AT* is set to $FALSE$ **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**

to *FALSE*, a second constant is added (line 5 of Table I). This is illustrated in Fig. 7 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))`.

Similarly to relations, each attribute gives a constant representing the type of its corresponding element and, in the case where isVariable is set to *FALSE*, to another constant (line 6 of Table I). However, when the isFunctional property is set to *TRUE*, the constant representing the type is defined as the set of functions between the *B System* element corresponding to the attribute domain and the one corresponding to the attribute range. The element corresponding to the attribute is then typed as a function. Furthermore, when isFunctional is set to *TRUE*, the isTotal property is used to assert if the function is total (isTotal=TRUE) or partial (isTotal=FALSE). For example, in Fig. 6, **`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}).

Finally, each individual (or data value) gives a constant (lines 8 and 9 of Table I). For example, in Fig. 7, the constant named **HD1** is the correspondent of the individual *HD1*.

D. Generation of B System Variables

An instance of Relation, of Concept or of Attribute, having its isVariable property set to *TRUE* gives a variable. For a concept, the variable represents the set of *B System* elements having this concept as type (line 7 of Table I). For a relation or an attribute, it represents the set of pairs between individuals (in case of relation) or between individuals and data values (in case of attribute) defined through it (lines 5 and 6 of Table I). For example, in Fig. 7, the variables named **landingSetState** and **handleState** appear because of the Attribute instances *landingSetState* and *handleState* for which the isVariable property is set to *TRUE* (Fig. 4).

E. Generation of B System Invariants and Properties

In this section, we are interested in translation rules between domain models and *B System* specifications that give *invariants* (instances of the Invariant class) or *properties* (instances of the Property class). Throughout this section, we will denote by *logic formula* (instance of the LogicFormula class) any invariant or property, knowing that a logic formula is a property when it involves only constant elements. Any other logic formula is an invariant. It should be noted that when the logic formula relates variables defined within the model and those defined within more abstract models, it is a *gluing invariant*.

When the isTransitive property of an instance of Relation *re* is set to *TRUE*, the logic formula $(re\ ;\ re) \subseteq re$ must appear in the *B System* component corresponding to the domain model, knowing that *";"* is the composition operator for relations (line 10 of Table I). For the isSymmetric property, the logic formula is $re^{-1} = re$. For the isASymmetric property, the logic formula is $(re^{-1} \cap re) \subseteq id(dom(re))$. For the isReflexive property, the logic formula is $id(dom(re)) \subseteq re$ and for the isIrreflexive property, the logic formula 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*).

An instance of DomainCardinality (respectively RangeCardinality) associated with an instance of Relation *re*, with bounds minCardinality and maxCardinality ($maxCardinality \geq 0$), gives the logic formula $\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) \in minCardinality..maxCardinality)$ (respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) \in minCardinality..maxCardinality)$).

When $minCardinality = maxCardinality$, then the logic formula 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 $maxCardinality = \infty$, then the logic formula 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 Fig. 7, logic formula `(1.9)` and `(1.10)` appear because of instances of RangeCardinality and DomainCardinality associated with the instance of Relation `LgOfHd` (Fig. 3).

The dual version of the previous rule allows the processing of instances of RangeCardinality.

Instances of RelationMaplet (respectively AttributeMaplet) associated with an instance of Relation (respectively Attribute) *RE* give rise, in the case where the isVariable property of *RE* is set to *FALSE*, to the property $RE = \{a_1 \mapsto i_1, a_2 \mapsto i_2, ..., a_j \mapsto i_j, ..., a_n \mapsto i_n\}$, where a_j designates the instance of Individual linked to the j-th instance of RelationMaplet (respectively AttributeMaplet), through the antecedent association, and i_j designates the instance of Individual (respectively DataValue) linked through the image association (line 11 of Table I). When the isVariable property of *RE* is set to *TRUE*, it is the substitution $RE := \{a_1 \mapsto i_1, a_2 \mapsto i_2, ..., a_j \mapsto i_j, ..., a_n \mapsto i_n\}$ which is rather defined in the *INITIALISATION* clause of the *B System* component (lines 12 and 13 of Table I). For example, in Fig. 7, the property `(1.11)` appears because of the association between **LG1** and **HD1** through **LgOfHd** (Fig. 4). 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).

Finally, any instance of Predicate gives a *B System* logic formula. When the predicate is an instance of GluingInvariant, the logic formula is a *B System* gluing invariant. For example, in Fig. 7, assertion `(1.21)` appears because of the gluing invariant (inv1).

F. The SysML/KAOS Domain Modeling Tool

The translation rules outlined here have been implemented within an open source tool [12]. It allows the construction of domain ontologies *(Fig. 8)* and generates the corresponding *B System* specifications (Fig. 6 and 7). It is build through *Jetbrains Meta Programming System* (MPS) [13], a tool to design domain specific languages using language-oriented programming. The SysML/KAOS domain modeling language is build using 28 MPS concepts by defining for each the properties, childrens, references, constraints, behaviours and custom editors. Each MPS concept represents a class of the SysML/KAOS domain metamodel. For each concept, the *properties* clause is used to define the attributes. The *Childrens* clause is used to define, for a concept `C`, the concepts that are parts of `C`. Finally, the *references* clause is used to define the linked concepts. For example, the MPS concept representing the class DomainModel defines all concepts representing the domain model elements as childrens and has a reference named *parentDomainModel* linking it to its parent domain model. Each

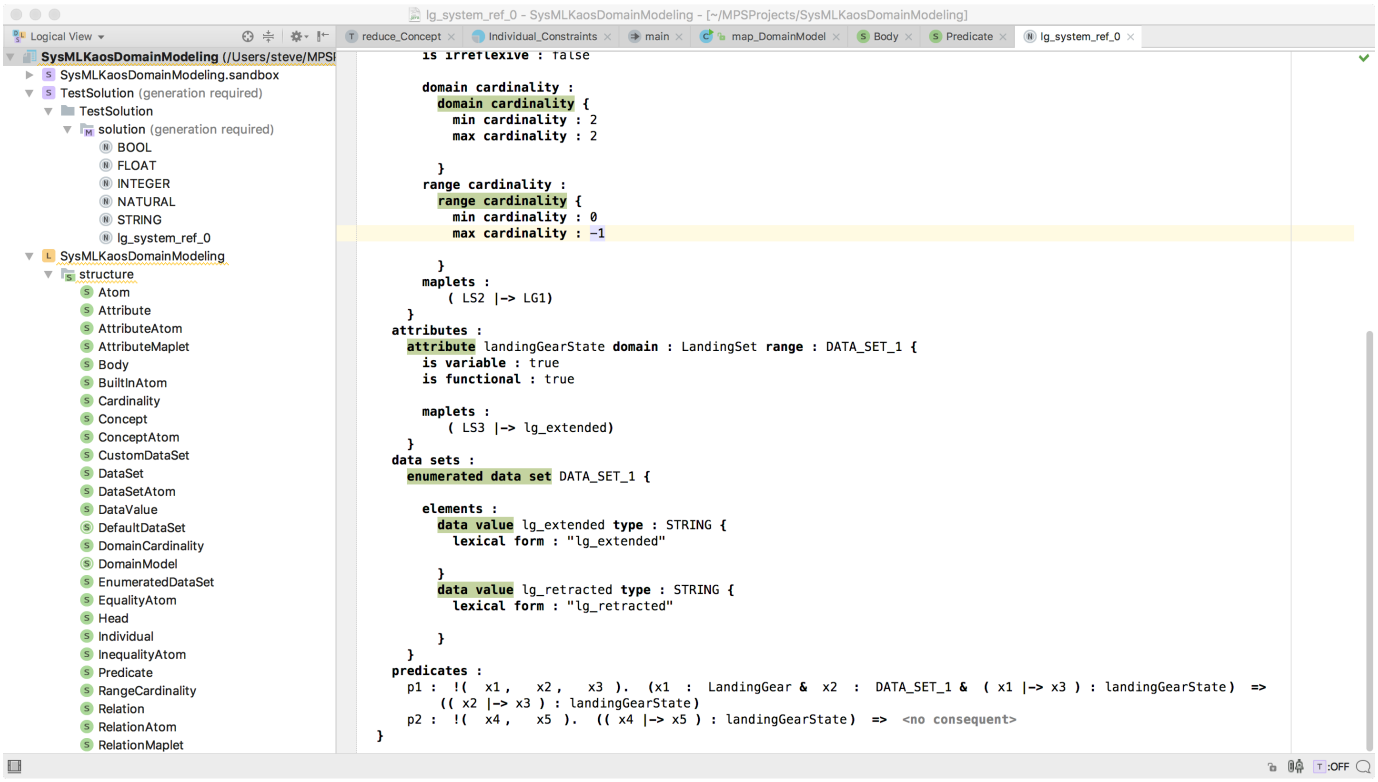

Fig. 8. Main screen of the SysML/KAOS domain modeling tool

new domain model is defined in a MPS solution using the SysML/KAOS domain modeling language. A MPS solution is an instantiation of a MPS language. We have also defined a language for the *B System* method. Thus, SysML/KAOS domain model solutions give *B System* solutions and traceability links that can be used to propagate updates performed on a solution into the paired solution. However, the update propagation feature is not currently supported by the tool and is a next step in our work.

V. Discussion

A. Formal Verification of the Translation Rules with Event-B

We have used the *Event-B* method to formally define the SysML/KAOS domain metamodel, the *B System* metamodel and the translations rules. Invariants and properties have been defined to encode the language semantics. For example, following the rules proposed in [31], [32], each class has been translated into an *Event-B* variable, typed as a subset and containing the correspondences of the class instances. Each abstract class (that is not a subclass of another class) has been translated into an Event-B abstract set. Each association or property has been translated into a variable typed as a relation. We have captured the correspondence links between instances of a class A of the SysML/KAOS domain metamodel and instances of a class B of the *B System* metamodel in a variable named *A_corresp_B* typed by the invariant $A\_corresp\_B \in A \rightarrowtail B$. An injection because each instance, on both sides, must have at most one correspondence. The injection is partial because all the elements are not translated at the same time. Each rule has been modeled as an *Event-B* event. The specification and verification have been done with *Rodin* [33], an industrial-strength tool supporting the *Event-B* method. The full specification can be found in [10], [11]. We have animated the rules with ProB [34] (Fig. 9) and we have discharged the proof obligations to ensure the absence of invariant violations and the consistency of rules. The animation of the *Event-B* specification has consisted, starting from instantiations of the SysML/KAOS domain metamodel, to simulate the translation sequence and to verify the obtained formal specification, while checking the fulfillment of the constraints, expressed with invariants and properties. For example, Figure 9 is an overview of the animation of the rules related to the translation of domain models. The top view presents an excerpt of the Event-B specification. The bottom view presents, from left to right, the current state of the system structure, the enabled operations and a summary of the performed operations. Rule `rule_1` (the rule for the translation of abstract domain models) has been used to translate the domain model `DomainModel_Set1` into the component `Component_Set1`. The same rule is enabled for the translation of `DomainModel_Set2`.

We have proved two essential properties regarding the rules. The first one is that the rules are isomorphisms (structure preserving), which guarantees that established links between elements of the ontologies are preserved between the corre-

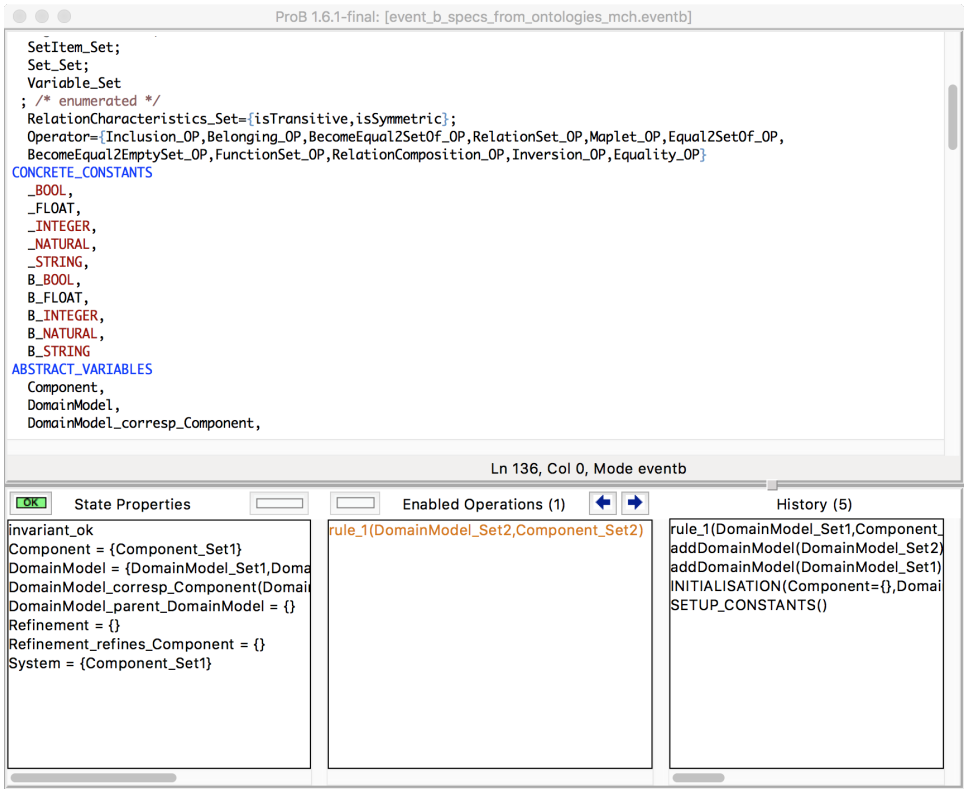

Fig. 9. Overview of the animation of the formal specification of translation rules under ProB

sponding 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. For example, to prove that for each domain model `PDM`, parent of the domain model `DM`, the *B System* component corresponding to `DM` refines the one corresponding to `PDM` and vice versa, the following invariants have been defined:

inv1: $\forall DM, PDM \cdot ((DM \in dom(parent) \land PDM = parent(DM) \land \{DM, PDM\} \subseteq dom(DomainModel\_corresp\_Component)) \Rightarrow (DomainModel\_corresp\_Component(DM) \in dom(refines) \land refines(DomainModel\_corresp\_Component(DM)) = DomainModel\_corresp\_Component(PDM)))$

inv2: $\forall o\_DM, o\_PDM \cdot ((o\_DM \in dom(refines) \land o\_PDM = refines(o\_DM) \land \{o\_DM, o\_PDM\} \subseteq ran(DomainModel\_corresp\_Component)) \Rightarrow (DomainModel\_corresp\_Component^{-1}(o\_DM) \in dom(parent) \land parent(DomainModel\_corresp\_Component^{-1}(o\_DM)) = DomainModel\_corresp\_Component^{-1}(o\_PDM)))$

The second essential property is that any domain model will always be translated into *B System* specifications, in a finite number of iterations of translation rules. To prove it, we have introduced *variants* 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 associated variant. Recall that in an *Event-B* specification, a *variant* is a natural integer or a set that each convergent event must decrease to ensure its termination in a finite number of iterations [8].

Table III summarises the key characteristics of the Rodin project corresponding to the *Event-B* specification and verification of the translation rules. The specification includes two refinement levels.

TABLE II
KEY CHARACTERISTICS OF THE *Event-B* SPECIFICATION OF TRANSLATION RULES

Characteristics	Root level	First refinement level
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**

The vast majority of proof obligations could not be automatically discharged due to the use of function operators ($\nrightarrow, \rightarrowtail, \rightarrow, \twoheadrightarrow$) in relation definitions.

B. Review of the Work Done on Case Studies

The FORMOSE approach has been used for the specification of a landing gear system [14], of a localisation software component for the *Cycab* vehicle [15] and of the *hybrid ERTMS/ETCS level 3* standard [16]. The corresponding models can be found in [17]–[19]. For each case study, we have built the goal and domain models. Then, translation rules, supported

by tools [2], [35], have been applied to obtain a *B System* specification containing the system structure and the skeleton of events. Regarding, for instance, the specification of the hybrid ERTMS/ETCS level 3 standard, the goal model includes seven refinement levels and nineteen goals. Three domain models have been defined for the root level and for the first and second refinement levels. The goal and domain models have been automatically translated into *B System* specifications and we have manually provided the body of events. The Rodin tool has been used for the verification and validation of the formal specification. Table III summarises the key characteristics related to the specification.

TABLE III
KEY CHARACTERISTICS RELATED TO THE FORMAL SPECIFICATION OF THE HYBRID ERTMS/ETCS LEVEL 3 STANDARD USING THE FORMOSE APPROACH

Refinement level	L0	L1	L2	L3	L4	L5	L6
Invariants	**4**	**11**	**13**	**4**	**6**	**5**	**9**
Proof Obligations (PO)	**20**	**40**	**50**	**13**	**5**	**5**	**14**
Automatically Discharged POs	**17**	**30**	**30**	**11**	**0**	**0**	**4**
Interactively Discharged POs	**3**	**5**	**20**	**2**	**5**	**5**	**10**

We have also specified the hybrid ERTMS/ETCS level 3 standard in a companion paper [36] using plain *Event-B*, in the traditional style. The corresponding specification includes four refinement levels and sixteen events. The FORMOSE approach allowed the decoupling between the formal specification handling difficulties and system modeling. Furthermore, it allowed us to trace the source and justify the need for each formal component and its contents, in relation with the SysML/KAOS goal and domain models. It has greatly improved the readability and reusability of the models. The approach allowed the encoding of alternatives for purpose achievement, as a result of the formalisation of goals linked with the *OR* refinement operator. The execution ordering and the refinement strategy have been enforced using proof obligations, whereas in [36] there is no proof about these aspects. The approach allowed us to better delineate the system boundaries. However, the specification obtained using the FORMOSE approach appears to be more abstract than the one obtained in [36]. It is necessary to refine it with more details from design choices. We believe that by accentuating the decomposition within the goal diagrams, it is possible to reach the same abstraction level.

VI. CONCLUSION AND FUTURE WORK

This paper focusses on the presentation of translation rules between SysML/KAOS domain models and *B System* specifications illustrated through a case study dealing with a landing gear system. The specifications thus obtained complete the formalisation of SysML/KAOS goal models by providing a description of the state of the system. The rules have been formalised and verified using *Rodin* [33], an industrial-strength tool supporting the *Event-B* method. We have proved that they are consistant and structure preserving. The full *Event-B specification* can be found in [10], [11]. A tool has been developed to support the approach [12] and it has been appraised with its usage for the specification of a landing gear system [14], of a localisation software component for the *Cycab* vehicle [15] and of the hybrid ERTMS/ETCS level 3 implementation [16]. It allowed us to quickly build the refinement hierarchy and to determine and express the safety invariants, without having to manipulate the formal specifications. The case study models can be found in [18], [19]. The tool can be used to model domain ontologies and to automatically translate them into *B System* specifications.Our work allows the complete extraction of the structural part of a *B System* specification from ontologies. We also extract the initialisation of state variables. However, it remains necessary to manually complete the body of events, which can lead to updates on the structure of the system (addition of variables and invariants).

Work in progress is aimed at studying the scalability and the maintenability of models, the first step being the propagation of updates on formal specifications within domain models. We are also working on integrating the translation rules within the open-source platform *Openflexo* [37] which federates the various contributions of the *FORMOSE* project partners [1] 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 [1] 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] ANR-14-CE28-0009, "Formose ANR project," 2017.
[2] A. Matoussi, F. Gervais, and R. Laleau, "A goal-based approach to guide the design of an abstract Event-B specification," in *ICECCS 2011*. ICS, pp. 139–148.
[3] C. Gnaho, F. Semmak, and R. Laleau, "Modeling the impact of non-functional requirements on functional requirements," ser. LNCS, vol. 8697. Springer, 2013, pp. 59–67.
[4] A. Mammar and R. Laleau, "On the use of domain and system knowledge modeling in goal-based Event-B specifications," in *ISoLA 2016, LNCS*. Springer, pp. 325–339.
[5] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, "Towards Using Ontologies for Domain Modeling within the SysML/KAOS Approach," *IEEE proceedings of MoDRE workshop, 25th IEEE International Requirements Engineering Conference*.
[6] K. Sengupta and P. Hitzler, "Web ontology language (OWL)," in *Encyclopedia of Social Network Analysis and Mining*, 2014, pp. 2374–2378.
[7] G. Pierra, "The PLIB ontology-based approach to data integration," in *IFIP 18th World Computer Congress*, ser. IFIP, vol. 156. Kluwer/Springer, 2004, pp. 13–18.
[8] J. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
[9] ClearSy, "Atelier B: B System," 2014. [Online]. Available: http://clearsy.com/
[10] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, "Event-B Specification of Translation Rules," 2017. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/SysMLKAOSDomainModelRules
[11] ——, "Formal Representation of SysML/KAOS Domain Models ," *ArXiv e-prints, cs.SE, 1712.07406*, Dec. 2017. [Online]. Available: https://arxiv.org/pdf/1712.07406.pdf

[12] ——, "SysML/KAOS Domain Modeling Tool," 2017. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser

[13] Jetbrains, "Jetbrains mps," 2017. [Online]. Available: https://www.jetbrains.com/mps/

[14] F. Boniol and V. Wiels, "The landing gear system case study," ser. ABZ. Springer, 2014.

[15] S. Sekhavat and J. H. Valadez, "The Cycab robot: a differentially flat system," in *IROS 2000*. IEEE, 2000, pp. 312–317.

[16] T. S. Hoang, M. Butler, and K. Reichl, "The Hybrid ERTMS/ETCS Level 3 Case Study," *ABZ*, pp. 1–3, 2018. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/blob/master/ABZ18_ERTMS/ERTMS_L3_Hybrid.pdf

[17] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, "SysML/KAOS Approach on the Hybrid ERTMS/ETCS Level 3 case study," 2018. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/ABZ18_ERTMS

[18] ——, "SysML/KAOS Domain Modeling Approach on Landing Gear," 2017. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/LandingGear

[19] ——, "SysML/KAOS Domain Modeling Approach on Localisation Component of Cycab," 2017. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/Tacos_Vehicle

[20] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[21] C. Gnaho and F. Semmak, "Une extension SysML pour l'ingénierie des exigences dirigée par les buts," in *28e Congrès INFORSID, France*, 2010, pp. 277–292.

[22] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, "The SysML/KAOS Domain Modeling Approach," *ArXiv e-prints, cs.SE, 1710.00903*, Sep. 2017. [Online]. Available: https://arxiv.org/pdf/1710.00903.pdf

[23] I. UL, "Owlgred home," 2017. [Online]. Available: http://owlgred.lumii.lv/

[24] H. H. Wang, D. Damljanovic, and J. Sun, "Enhanced semantic access to formal software models," in *Formal Methods and Software Engineering - ICFEM, LNCS*, vol. 6447. Springer, 2010, pp. 237–252.

[25] E. Doberkat, "The Object-Z specification language," *Softwaretechnik-Trends*, vol. 21, no. 1, 2001.

[26] J. S. Dong, J. Sun, and H. H. Wang, "Z approach to semantic web," in *Formal Methods and Software Engineering - ICFEM, LNCS*, vol. 2495. Springer, 2002, pp. 156–167.

[27] F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks, "Reference description of the DAML+ OIL ontology markup language," 2001.

[28] I. Poernomo and T. Umarov, "A mapping from normative requirements to Event-B to facilitate verified data-centric business process management," ser. CEE-SET LNCS, vol. 7054. Springer, 2009, pp. 136–149.

[29] E. Alkhammash, M. J. Butler, A. S. Fathabadi, and C. Cîrstea, "Building traceable Event-B models from requirements," *Sci. Comput. Program.*, vol. 111, pp. 318–338, 2015.

[30] Alkhammash, Eman H., "Derivation of Event-B Models from OWL Ontologies," *MATEC Web Conf.*, vol. 76, p. 04008, 2016.

[31] C. Snook and M. Butler, "UML-B: Formal Modeling and Design Aided by UML," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 92–122, Jan. 2006.

[32] R. Laleau and A. Mammar, "An overview of a method and its support tool for generating B specifications from UML notations." ICS, 2000, pp. 269–272.

[33] M. J. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, Eds., *Rigorous Development of Complex Fault-Tolerant Systems*, ser. LNCS, vol. 4157. Springer, 2006.

[34] M. Leuschel and M. J. Butler, "Prob: A model checker for B," ser. LNCS, vol. 2805. Springer, 2003, pp. 855–874.

[35] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, "The SysML/KAOS Domain Modeling Language (Tool and Case Studies)," 2017. [Online]. Available: https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master

[36] A. Mammar, M. Frappier, S. Tueno, and R. Laleau, "An Event-B Model of the ERTMS/ETCS Level 3 Standard," 2018. [Online]. Available: info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study

[37] Openflexo, "Openflexo project," 2015. [Online]. Available: http://www.openflexo.org